4

The Name Service in Detail

> This is gonna hurt me
> more than it does you.
>
> — Common lie

Think of the Name Service as a database system. The data may be stored in an NBNS server (P mode), distributed across all of the participating nodes in an IP subnet (B mode), or a combination of the two (M or H mode).

Name Service messages are the transactions that maintain and utilize the NBT name-to-IP-address mapping database. These transactions fall into three basic categories:

Name Registration/Refresh
: The process by which an application adds and maintains a NetBIOS name to IP address mapping within an NBT scope.

Name Query
: The process of resolving a NetBIOS name to an IP address.

Name Release
: The process by which a NetBIOS name to IP address mapping is removed from within an NBT scope.

These three represent the lifecycle of an NBT name.

The RFCs also specify support for the NetBIOS API **Adapter Status Query** function. Implementation of the Adapter Status Query is quite similar

to that of the Name Query, so it gets lumped in with the Name Service. This is fairly reasonable, since the query packets are almost identical and the most important result of the status query is a list of names owned by the target node.

4.1 NBT Names: Once More with Feeling

Let's review what we've learned so far:

- Though the RFCs do not say so, NetBIOS names should be converted to upper case *before* they are encoded. The practice probably goes back to early IBM implementations. Converting NetBIOS names to upper case allows for comparison of the encoded string, rather than requiring that NBT names be decoded and compared using a case-insensitive function. Some existing implementations use this shortcut, and will not recognize names with encoded lower-case characters.
- The RFCs list NetBIOS names as being 16 bytes in length. It is common practice, however, to implement NetBIOS names as two subfields: a 15-byte name and a one-byte suffix. (That's what Microsoft does so everyone else has to do it too.) The suffix byte actually winds up being quite useful. The suffix byte is read as an integer in the range 0..255, so it is *not* converted to upper-case.
- If the NetBIOS name is less than 15 bytes, it must be padded. The space character (`0x20`) is the designated padding character (though there are some rare, special-case exceptions.)
- Other than length and padding, the only restriction the RFCs place on the syntax of a NetBIOS name is that it may not begin with an asterisk ('`*`').

4.1.1 *Valid NetBIOS Name Characters*

Any octet value can be encoded using the first-level mechanism. In theory, then, any eight-bit value can be part of a NetBIOS name. Keep this in mind and be prepared. There are some very strange names in use in the wild.

In practice, implementations do place some restrictions on the characters that may be used in NetBIOS names. These restrictions are implemented at the application layer, and should be considered artificial. Under Windows 9x,

for example, the "Network Identity" control panel allows only the following characters in a machine name:

Valid Windows 9x machine name characters

' '	==	0x20	'-'	==	0x2D
'!'	==	0x21	'.'	==	0x2E
'#'	==	0x23	'@'	==	0x40
'$'	==	0x24	'^'	==	0x5E
'%'	==	0x25	'_'	==	0x5F
'&'	==	0x26	'{'	==	0x7B
'\''	==	0x27 (single quote)	'}'	==	0x7D
'('	==	0x28	'~'	==	0x7E
')'	==	0x29	alphanumeric characters		

Yet the same Windows 9x system may also register the special-purpose name "\x01\x02__MSBROWSE__\x02\x01", which contains control characters as shown.

Note that the set of alphanumeric characters may include extended characters, such as 'Å' or 'Ü'. Unfortunately, these are often represented by different octet values under different operating systems, or even under different configurations of the same operating system.

Some examples:

Character	ISO Latin-1	DOS Code Page 437
'Ä'	0xC4	0x8E
'Ç'	0xC7	0x80
'É'	0xC9	0x90
'Î'	0xCE	—
'Ö'	0xD6	0x99
'Ñ'	0xD1	0xA5
'Ù'	0xD9	—

As you can see, the mapping between character sets can be a bit of a challenge — particularly since there is no standard character set for use in NBT and no mechanism for negotiating a common character set.[1]

One more thing to consider when dealing with NetBIOS name characters: Windows NT will generate a warning — and W2K an error — if the Machine Name is not also a valid DNS name. You may need to do some testing to determine which characters Windows considers valid DNS label characters.

4.1.2 *NetBIOS Names within Scope*

Under NBT, NetBIOS names exist within a *scope*. The scope is the set of all machines which can "see" the name. For B nodes, the scope is limited to the IP broadcast domain. For P nodes, the scope is limited to the set of nodes that share the same NBNS. For M and H nodes, the scope is the union of the broadcast domain and the shared NBNS.

Scope can be further refined using a *Scope ID*. The Scope ID effectively sub-divides a virtual NetBIOS LAN into separate, named vLANs. Unfortunately, few (if any) implementations actually support multiple Scope IDs so this feature is of limited practical use.

The syntax of the Scope ID matches the best-practices recommendations for DNS domain names. (Some Windows flavors allow almost any character value in a Scope ID string. Sigh.) Scope IDs should be converted to upper case before use on the wire.

Annoyance Alert

In versions of Windows 95 and '98 that we tested, the Scope ID field in the network setup control panel is greyed out if no WINS server IP address is specified. That is, you cannot enter a Scope ID if your machine is running in B mode.

You can work around this by entering the Scope ID in the right place in the registry, or by entering a (bogus) WINS server IP, entering the Scope ID, saving your changes, rebooting, reopening the network control panel, removing the WINS IP entry, saving your changes, and rebooting again.

---

1. To further complicate matters Microsoft has registered its own character sets, such as the Windows-1252 character set. Windows-1252 is a superset of ISO Latin-1. It uses octets in the range `0x80..0x9F` (normally reserved for control characters) to represent some additional display characters, such as the trademark symbol (™). This is why non-Microsoft web browsers on non-Microsoft platforms often display question marks all over the screen when they load web pages generated by Microsoft products.

The system does not seem to clear the Scope ID once it has been entered. To clear the Scope ID you must either edit the registry, or enter a (bogus) WINS server IP, clear out the Scope ID in the control panel, save your changes, reboot, reopen the network control panel, remove the WINS IP entry, save your changes, and reboot.

Windows NT behaves correctly, and does allow the entry of a Scope ID in B mode.

4.1.3 *Encoding and Decoding NBT Names*

First Level Encoding converts a 16-byte NetBIOS name into a 32-byte encoded name, and then combines it with the Scope ID. For example:

```
"EOGFGLGPCACACACACACACACACACACAAA.CAT.ORG"
```

We have chosen to call this format the *NBT Name*. Second Level Encoding is applied to the NBT name to create the on-the-wire format, which we will refer to as the *Encoded NBT Name*:

```
"\x20EOGFGLGPCACACACACACACACACACACAAA\x03CAT\x03ORG\0"
```

As previously described, the maximum length of a label in an NBT name is 63 bytes. This is because the label length field is divided into two sub-fields, the first of which is a two-bit flag field with four possible values:

00 == 0: Label Length
01 == 1: Reserved (unused)
10 == 2: Reserved (unused)
11 == 3: Label String Pointer

With both bits clear (zero), the next 6 bits are the label LENGTH. The LENGTH field is an unsigned integer with a value in the range 0..63.

0	1	2	3	4	5	6	7
0	0	LENGTH					

If both flag bits are set, however, then the next *fourteen* bits are a "Label String Pointer"; the offset at which the real label can be found.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	LABEL STRING POINTER													

Label String Pointers are used to reduce the size of Name Service messages that might otherwise contain two copies of the same NBT name. For example, a NAME REGISTRATION REQUEST message includes both a QUESTION_RECORD and an ADDITIONAL_RECORD, each of which would otherwise contain the same NBT name. Instead of duplicating the name, however, the ADDITIONAL_RECORD.RR_NAME field contains a label string pointer to the QUESTION_RECORD.QUESTION_NAME field.

Label String Pointers are a prime example of the NBT theory/practice dichotomy, and another throwback to the DNS system. As it turns out, the only Label String Pointer value ever used in NBT is 0xC00C. The reason for this is quite simple. The NBT header is a fixed size (12 bytes), and is always followed by a block that starts with an encoded NBT Name. Thus, the offset of the first name in the packet is always 12 (0x0C). Any further name field in the packet will point back to the first.

So, the rule of thumb is that the encoded NBT name will always be found at byte offset 0x000C. As a shortcut, some implementations work directly with the encoded name and only bother to decode the name when interacting with a user. Decoding, however, is fairly straightforward:

Listing 4.1: Level 2 and Level 1 decoding

```
int L2_Decode( uchar *dst,     /* Decoded name target buffer.   */
               uchar *src,     /* Encoded name source buffer.   */
               int    srcpos,  /* Start position of name.       */
               int    srcmax ) /* Size of source buffer.        */
  {
  int len;
  int pos;
  int next;

  /* Be safe. */
  dst[0] = '\0';

  /* Get encoded string length (doesn't include root label). */
  len = strlen( (char *)&(src[srcpos]) );

  /* If length is zero, return the empty string. */
  if( 0 == len )
    return( 0 );
```

```
  /* Make sure name does not exceed source buffer length. */
  if( len >= (srcmax - srcpos) )
    return( -1 );

  /* Copy source to destination skipping the first label length byte
   * (but including the terminating nul label length).
   */
  (void)memcpy( dst, &(src[srcpos+1]), len );

  /* Now find remaining label length bytes
   * and convert them to dots.
   */
  for( pos = src[srcpos];         /* Read the first label length. */
       '\0' != (next = dst[pos]); /* While label length is > 0... */
       pos += next + 1 )          /* Move one byte beyond label.  */
    {
    dst[pos] = '.';
    }

  return( --len );  /* Return string length. */
  } /* L2_Decode */

int L1_Decode( uchar *name,    /* Target.  Minimum 16 bytes. */
               uchar *src,     /* Message buffer.            */
               int    srcpos,  /* Start position of name.    */
               int    srcmax ) /* Size of source buffer.     */
  {
  int    i;
  int    suffix;
  uchar *p = &src[srcpos];

  /* Make sure we have 32 bytes worth of message to read.    */
  if( (srcmax - srcpos) < 32 )
    {
    name[0] = '\0';
    return( -1 );
    }

  /* Convert each source pair to their original octet value. */
  for( i = 0; i < 32; i++ )
    name[i/2] = ( (( (int)(p[i]) - 'A' ) << 4)
                +  ( (int)(p[++i]) - 'A' ) );
```

```
  /* Copy out suffix byte and replace with nul terminator.   */
  suffix = name[15];
  name[15] = '\0';

  /* Trim off trailing spaces, if any. */
  for( i = 14; (i >= 0) && (' ' == name[i]); i-- )
    name[i] = '\0';

  return( suffix );   /* Return the suffix value as an int.  */
  } /* L1_Decode */
```

The `L2_Decode()` function copies the encoded NBT name to the destination buffer, skipping the first label length byte and replacing internal label length bytes with the dot character. That is, given the input string:

```
"\x20EOGFGLGPCACACACACACACACACACACAAA\x03CAT\x03ORG\0"
```

it will produce the string:

```
"EOGFGLGPCACACACACACACACACACACAAA.CAT.ORG"
```

The `L1_Decode()` function decodes the First Level Encoded NetBIOS name, and hands back the suffix byte as its return value.

4.2 NBT Name Service Packets

RFC 1002 lists 17 different Name Service packet types, constructed from three basic building blocks:

- header,
- query records, and
- resource records.

These pieces are described in more detail below.

4.2.1 *Name Service Headers*

The header is an array of six 16-bit values, as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NAME_TRN_ID															
FLAGS															
QDCOUNT															
ANCOUNT															
NSCOUNT															
ARCOUNT															

Managing Name Service headers is fairly straightforward. With the exception of the FLAGS field, all of the fields are simple unsigned integers. The entire thing can be represented in memory as an array of `unsigned short int`, or whatever is appropriate in your programming language of choice.

The FLAGS field is further broken down thus:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	OPCODE				NM_FLAGS							RCODE			

Handling the bits in the FLAGS field is fairly trivial for any seasoned programmer. One simple solution is to shift the values given in RFC 1002, Section 4.2.1.1 into their absolute positions. For example, an OPCODE value of 0x7 (WACK) would be left-shifted 11 bits to align it properly in the OPCODE subfield:

```
(0x0007 << 11) = 0x3800 = 0 0111 00000000000(bin)
```

...which puts it where it's supposed to be:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	OPCODE				NM_FLAGS							RCODE			
0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0

Listing 4.2 presents NS_Header.h, a header file that will be referenced as we move forward. It provides a set of re-aligned FLAGS subfield values plus a few extra constants. These values will be covered below, when we explain how to use each of the Name Service message types.

Listing 4.2: Name Service packet header FLAGS subfield values: NS_Header.h

```
#define NBTNS_R_BIT       0x8000  /* The 'R'esponse bit   */

/* OPCODE values */
#define OPCODE_QUERY      0x0000  /* Query        (0<<11) */
#define OPCODE_REGISTER   0x2800  /* Registration (5<<11) */
#define OPCODE_RELEASE    0x3000  /* Release      (6<<11) */
#define OPCODE_WACK       0x3800  /* WACK         (7<<11) */
#define OPCODE_REFRESH    0x4000  /* Refresh      (8<<11) */
#define OPCODE_ALTREFRESH 0x4800  /* Alt Refresh  (9<<11) */
#define OPCODE_MULTIHOMED 0x7800  /* Multi-homed  (f<<11) */
#define OPCODE_MASK       0x7800  /* Mask                 */

/* NM_FLAGS subfield bits */
#define NM_AA_BIT         0x0400  /* Authoritative Answer */
#define NM_TR_BIT         0x0200  /* TRuncation flag      */
#define NM_RD_BIT         0x0100  /* Recursion Desired    */
#define NM_RA_BIT         0x0080  /* Recursion Available  */
#define NM_B_BIT          0x0010  /* Broadcast flag       */

/* Return Codes */
#define RCODE_POS_RSP     0x0000  /* Positive Response    */
#define RCODE_FMT_ERR     0x0001  /* Format Error         */
#define RCODE_SRV_ERR     0x0002  /* Server failure       */
#define RCODE_NAM_ERR     0x0003  /* Name Not Found       */
#define RCODE_IMP_ERR     0x0004  /* Unsupported request  */
#define RCODE_RFS_ERR     0x0005  /* Refused              */
#define RCODE_ACT_ERR     0x0006  /* Active error         */
#define RCODE_CFT_ERR     0x0007  /* Name in conflict     */
#define RCODE_MASK        0x0007  /* Mask                 */

/* Used to set the record count fields. */
#define QUERYREC          0x1000  /* Query Record         */
#define ANSREC            0x0100  /* Answer Record        */
#define NSREC             0x0010  /* NS Rec (never used)  */
#define ADDREC            0x0001  /* Additional Record    */
```

The NAME_TRN_ID field is the transaction ID, which should probably be handled by the bit of code that sends and receives the NBT messages. Many implementations use a simple counter to generate new transaction IDs (Samba uses a random number generator), but these should always be checked to ensure that they are not, by chance, the same as the transaction ID of a conversation initiated by some other node. Better yet, the originating node's IP address should be used as an additional key for segregating transactions.

The four COUNT fields indicate the number of Question and Resource Records which follow. In theory, each of these fields can contain a value in the range 0..65535. In practice, however, the count fields will contain either 0 or 1 as shown in the record layouts in RFC 1002, Section 4.2. It appears as though some implementations either ignore these fields or read them as simple booleans.

One final consideration is the byte order of NBT messages. True to its DNS roots, NBT uses network byte order (big-endian). Some microprocessors — including Alpha, MIPS, and Intel i386 family — use or can use little-endian byte order.[2] If your target system is little-endian, or if you want your code to be portable, you will need to ensure that your integers are properly converted to and from network byte order. Many systems provide the `htonl()`, `htons()`, `ntohl()`, and `ntohs()` functions for exactly this purpose.

Bizarre Twist Alert

The SMB protocol was originally built to run on DOS. DOS was originally built to run on Intel chips, so SMB is little-endian... the opposite of the NBT transport!

This next bit of code is `nbt_nsHeader.c`. It shows how to create and parse NBT Name Service headers. As with all of the code presented in this book, it is designed to be illustrative, not efficient. (We know you can do better.)

Listing 4.3: Read and write Name Service headers: `NS_Header.c`

```
#include <netinet/in.h>     /* htons(), ntohs(), etc. */

#include "NS_Header.h"      /* From Listing 4.2.      */

void Put_NS_TID( ushort hdr[], ushort TrnID )
  /* ------------------------------------------------- **
   * Store the transaction ID in the Name Service header.
   * ------------------------------------------------- **
   */
  {
  hdr[0] = htons( TrnID );
  } /* Put_NS_TID */
```

2. Big-endian byte order is also known as "normal," "intuitive," or "obvious" byte order. Little-endian is sometimes referred to as "annoying," "dysfunctional," or "stupid." These designations do not, of course, reflect any bias or preference.

```
void Put_NS_Hdr_Flags( ushort hdr[], ushort flags )
  /* ------------------------------------------------------ **
   * Store the flags in the NBT Name Service header.
   * ------------------------------------------------------ **
   */
  {
  hdr[1] = htons( flags );
  } /* Put_NS_Hdr_Flags */

void Put_NS_Hdr_Rec_Counts( ushort hdr[], int reccount )
  /* ------------------------------------------------------ **
   * Place (ushort)1 into each record count field for
   * which the matching flag bit is set in reccount.
   * ------------------------------------------------------ **
   */
  {
  ushort one;

  one = htons( 1 );

  hdr[2] = ( QUERYREC & reccount ) ? one : 0;
  hdr[3] = ( ANSREC   & reccount ) ? one : 0;
  hdr[4] = ( NSREC    & reccount ) ? one : 0;
  hdr[5] = ( ADDREC   & reccount ) ? one : 0;
  } /* Put_NS_Hdr_Rec_Counts */

ushort Get_NS_Hdr_TID( ushort hdr[] )
  /* ------------------------------------------------------ **
   * Read and return the transaction ID.
   * ------------------------------------------------------ **
   */
  {
  return( ntohs( hdr[0] ) );
  } /* Get_NS_Hdr_TID */

ushort Get_NS_Hdr_Flags( ushort hdr[] )
  /* ------------------------------------------------------ **
   * Read and return the flags field.
   * ------------------------------------------------------ **
   */
  {
  return( ntohs( hdr[1] ) );
  } /* Get_NS_Hdr_Flags */
```

```
int Get_NS_Hdr_Rec_Counts( ushort hdr[] )
  /* ------------------------------------------------------ **
   * Convert the four record count fields into a single
   * flagset.
   * ------------------------------------------------------ **
   */
  {
  int tmp = 0;

  if( hdr[2] )
    tmp |= QUERYREC;

  if( hdr[3] )
    tmp |= ANSREC;
  if( hdr[4] )
    tmp |= NSREC;
  if( hdr[5] )
    tmp |= ADDREC;

  return( tmp );
  } /* Get_NS_Hdr_Rec_Counts */
```

4.2.2 *Name Service Question Records*

The question record is also simple. It consists of an encoded NBT name (in the QUESTION_NAME field) followed by two unsigned 16-bit integer fields: the QUESTION_TYPE and QUESTION_CLASS.

The length of an encoded NBT name is at least 34 bytes, but it will be longer if a Scope ID is used, so the QUESTION_NAME field has no fixed length. There is also no padding done to align the integer fields. The QUESTION_TYPE and QUESTION_CLASS follow immediately after the QUESTION_NAME.

>= 34 bytes	2 bytes	2 bytes
QUESTION_NAME ...	QUESTION_TYPE	QUESTION_CLASS

There are only two valid values for the QUESTION_TYPE field. These are:

NB == 0x0020 indicates a standard Name Query,
NBSTAT == 0x0021 indicates a Node Status Query.

The QUESTION_CLASS field always has a value of:

IN == 0x0001 indicates the "Internet Class."

Go back and take a look at the broadcast name query example presented earlier. In that example, we hard-coded both the NBT Name Service header and the tail-end of the question record. Now that you have a clearer understanding of the fields involved, you should be able to design much more flexible code. Here's a start:

Listing 4.4a: Reading/writing question records: NS_Qrec.h

```
/* Query Type */
#define QTYPE_NB      0x0020  /* Name Query     */
#define QTYPE_NBSTAT  0x0021  /* Adapter Status */

/* Query Class */
#define QCLASS_IN     0x0001  /* Internet Class */
```

Listing 4.4b: Reading/writing question records: NS_Qrec.c

```
#include <string.h>     /* For memcpy() */
#include <netinet/in.h> /* htons(), ntohs(), etc. */

#include "NS_Qrec.h"

int Put_Qrec( uchar       *dst,
              const uchar *name,
              const uchar  pad,
              const uchar  sfx,
              const uchar *scope,
              const ushort qtype )
  /* ------------------------------------------------- **
   * Store the fully encoded NBT name in the destination
   * buffer.  Also write the QUERY_TYPE and QUERY_CLASS
   * values.
   * ------------------------------------------------- **
   */
  {
  int    len;
  ushort tmp;
  ushort qclass_in;

  qclass_in = htons( QCLASS_IN );
```

```
  /* Validate the qtype. */
  if( (QTYPE_NB != qtype)
   && (QTYPE_NBSTAT != qtype ) )
    return( -1 );

  len = L2_Encode( dst, name, pad, sfx, scope );
  if( len < 0 )
    return( len );

  tmp = htons( qtype );
  (void)memcpy( &(dst[len]), &tmp, 2 );
  len += 2;

  (void)memcpy( &(dst[len]), &qclass_in, 2 );
  return( len + 2 );
  } /* Put_Qrec */

ushort Get_Qtype( const uchar *qrec, int offset )
  /* ------------------------------------------------------ **
   * Read the QUERY_TYPE field from a query record.
   * Note that the offset parameter can be derived using
   * L2_Decode() function in Listing 4.1.  Either that,
   * or use 1+strlen( qrec ).
   * ------------------------------------------------------ **
   */
  {
  ushort tmp;

  /* Read the two bytes from the packet.
   */
  tmp  = (ushort)(qrec[offset]) * 256;
  tmp |= qrec[1+offset];

  /* Convert to host byte order and return. */
  return( ntohs( tmp ) );
  } /* Get_Qtype */
```

4.2.3 *Name Service Resource Records*

For convenience, we will break the Resource Record into three sub-parts:

- the Name section,
- the `TTL` field, and
- the Resource Data section.

The Name section has the same structure as a Query Entry record, except that the RR_NAME field may contain a 16-bit Label String Pointer instead of a complete NBT name.

2 bytes or >= 34 bytes	2 bytes	2 bytes
RR_NAME ...	RR_TYPE	RR_CLASS

The RR_TYPE field is used to indicate the type of the resource record, which has an effect on the structure of the resource data section. The available values for this field are:

```
     A == 0x0001 (not used in practice)
    NS == 0x0002 (not used in practice)
  NULL == 0x000A (not used in practice)
    NB == 0x0020
NBSTAT == 0x0021
```

The values marked as "not used in practice" are described in the RFCs, and indicated as valid values, but are never really used in modern implementations. The value of RR_TYPE will be NB except in a NODE STATUS REPLY, in which case NBSTAT is used.

As with the question record, the RR_CLASS field always has a value of:

```
IN == 0x0001
```

The TTL field follows the name section. It indicates the "Time To Live" value associated with a resource record. Each NBT name-to-IP-address mapping in the NBNS database has a TTL value. This allows records to "fade out" if they are not renewed or properly released. The TTL field is an unsigned long integer, measured in seconds. A value of zero indicates infinite TTL.

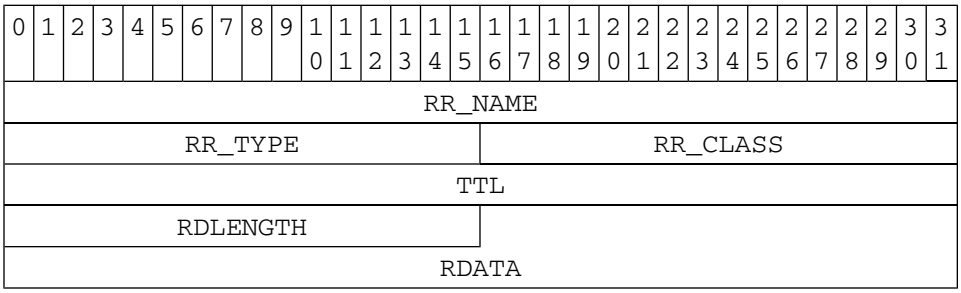
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
TTL																															

The last sub-part of the resource record is the resource data section, which is made up of two fields:

2 bytes	RDLENGTH bytes
RDLENGTH	RDATA ...

The RDLENGTH field is an unsigned 16-bit integer value indicating the length, in bytes, of the RDATA field. The structure of the contents of the RDATA field will vary from one message type to another.

The Resource Record structure, as described in Section 4.2.1.3 of RFC 1002, looks just like this:



It is always good to have some code to play with. This next set of functions can be used to manipulate Resource Records.

Listing 4.5a: Name Service Resource Records: NS_Rrec.h

```
/* Label String Pointer. */
#define LSP           0xC00C  /* Pointer to offset 12 */

/* Resource Record Type. */
#define RRTYPE_A      0x0001  /* IP Addr RR (unused)  */
#define RRTYPE_NS     0x0002  /* Name Server (unused) */
#define RRTYPE_NULL   0x000A  /* NULL RR (unused)     */
#define RRTYPE_NB     0x0020  /* NetBIOS              */
#define RRTYPE_NBSTAT 0x0021  /* NB Status Response   */

/* Resource Record Class. */
#define RRCLASS_IN    0x0001  /* Internet Class       */
```

Listing 4.5b: Name Service Resource Records: `NS_Rrec.c`

```
#include <string.h>     /* For memcpy() */
#include <netinet/in.h> /* htons(), ntohs(), etc. */

#include "NS_Rrec.h"

int Put_RRec_Name( uchar       *rrec,
                   const uchar *name,
                   const uchar  pad,
                   const uchar  sfx,
                   const uchar *scope,
                   const ushort rrtype )

  /* ------------------------------------------------- **
   * Create and store the fully qualified NBT name in the
   * destination buffer.  Also store the RR_TYPE and
   * RR_CLASS values.
   * Return the number of bytes written.
   * ------------------------------------------------- **
   */
  {
  int    len;
  ushort tmp;
  ushort rrclass_in;

  /* Validate the rrtype.
   * Note that we exclude the A, NS, and NULL RRTYPEs
   * as these are never used.
   */
  if( (RRTYPE_NB != rrtype)
   && (RRTYPE_NBSTAT != rrtype ) )
    return( -1 );

  len = L2_Encode( rrec, name, pad, sfx, scope );
  if( len < 0 )
    return( len );

  tmp = htons( rrtype );
  (void)memcpy( &(rrec[len]), &tmp, 2 );
  len += 2;

  rrclass_in = htons( RRCLASS_IN );
  (void)memcpy( &(rrec[len]), &rrclass_in, 2 );
  return( len + 2 );
  } /* Put_RRec_Name */
```

```
int Put_RRec_LSP( uchar *rrec, const ushort rrtype )
  /* ------------------------------------------------- **
   * Write a Label String Pointer (LSP) instead of an NBT
   * name.  RR_TYPE and RR_CLASS are also written.
   * Return the number of bytes written (always 6).
   * ------------------------------------------------- **
   */
  {
  ushort tmp;
  ushort lsp;
  ushort rrclass_in;

  lsp = htons( 0xC00C );
  (void)memcpy( rrec, &lsp, 2 );

  tmp = htons( rrtype );
  (void)memcpy( &(rrec[2]), &tmp, 2 );

  rrclass_in = htons( RRCLASS_IN );
  (void)memcpy( &(rrec[4]), &rrclass_in, 2 );
  return( 6 );
  } /* Put_RRec_LSP */

int Put_RRec_TTL( uchar *rrec, int offset, ulong ttl )
  /* ------------------------------------------------- **
   * Write the TTL value at rrec[offset].
   * By this point it should be obvious that functions or
   * macros for transferring long and short integers to
   * and from packet buffers would be a good idea.
   * ------------------------------------------------- **
   */
  {
  ttl = htonl( ttl );

  (void)memcpy( &(rrec[offset]), &ttl, 4 );
  return( 4 );
  } /* Put_RRec_TTL */

int Is_RRec_LSP( const uchar *rrec )
  /* ------------------------------------------------- **
   * Check the Resource Record to see if the name field
   * is actually a Label String Pointer.
   *
   * If the name is not an LSP, the function returns 0.
   * If the name is a valid LSP, the function returns 12
   * (which is the offset into the received packet at
   * which the QUERY_NAME can be found).
```

```
   * If the name contains an invalid label length, or
   * an invalid LSP, the function will return -1.
   * ------------------------------------------------- **
   */
  {
  if( 0 == (0xC0 & rrec[0]) )
    return( 0 );  /* Not an LSP */

  if( (0xC0 == rrec[0]) && (0x0C == rrec[1]) )
    return( 12 ); /* Valid LSP */

  return( -1 );   /* Bogon */
  } /* Is_RRec_LSP */

ushort Get_RR_type( const uchar *rrec, int offset )
  /* ------------------------------------------------- **
   * Read the RR_TYPE value.  The offset can be
   * determined by decoding the NBT name using the
   * L2_Decode() function from Listing 4.1.
   * ------------------------------------------------- **
   */
  {
  ushort tmp;

  /* Read the two bytes from the packet.
   */
  (void)memcpy( &tmp, &(rrec[offset]), 2 );

  /* Convert to host byte order and return. */
  return( ntohs( tmp ) );
  } /* Get_RR_type */

ulong Get_RRec_TTL( const uchar *rrec, int offset )
  /* ------------------------------------------------- **
   * Read the TTL value.
   * ------------------------------------------------- **
   */
  {
  ulong tmp;

  (void)memcpy( &tmp, &(rrec[offset]), 4 );
  return( ntohl( tmp ) );
  } /* Get_RRec_TTL */
```

4.3 Conversations with the Name Service

We will now introduce a simple syntax for describing how to fill network packets. This syntax is neither standard nor rigorous, just something the author whipped up to help explain what goes into a message. If it looks like someone else's syntax (one which perhaps took long hours of study, concentration, and thought to develop) then apologies are probably in order.

Disclaimer Alert

Any resemblance to an actual syntax, living or dead, real or imaginary, is entirely coincidental.

A broadcast name query, described using our little syntax, would look like this:

```
NAME QUERY REQUEST (Broadcast)
  {
  HEADER
    {
    NAME_TRN_ID = <Set when packet is transmitted>
    FLAGS
      {
      OPCODE = 0x0
      RD     = TRUE
      B      = TRUE
      }
    QDCOUNT = 1
    }
  QUESTION_RECORD
    {
    QUESTION_NAME  = <Encoded NBT Name>
    QUESTION_TYPE  = NB (0x0020)
    QUESTION_CLASS = IN (0x0001)
    }
  }
```

Basically, the rules are these:

- If a record (a header, question record, or resource record) is not specified, it is not included in the packet. In the example above there are no resource records specified. We know from the example code that there are no resource records in a `NAME QUERY REQUEST`.

- If a field is not specified, it is zeroed. In the example above the RCODE field of the FLAGS sub-record has a value of 0x0, and the NSCOUNT field (among others) also has a value of 0.
- Comments in angle brackets are short explanations, describing what should go into the field. More complete explanations, if needed, will be found in the accompanying text.
- Comments in parentheses provide additional information, such as the value of a specified constant.
- ...and yes, each squirrelly bracket gets its own line.

It's not a particularly formal syntax, but it will serve the purpose.

4.3.1 *Name Registration*

Nodes send NAME REGISTRATION REQUEST messages when they wish to claim ownership of a name. The messages may be broadcast on the local LAN (B mode), or sent directly to an NBNS (P mode). (M and H mode are combinations of B and P modes with their own special quirks. We will get to those further on.)

A NAME REGISTRATION REQUEST message looks like this:

```
NAME REGISTRATION REQUEST
  {
  HEADER
    {
    NAME_TRN_ID = <Set when packet is transmitted>
    FLAGS
      {
      OPCODE = 0x5 (Registration)
      RD     = TRUE (1)
      B      = <TRUE for broadcast registration, else FALSE>
      }
    QDCOUNT = 1
    ARCOUNT = 1
    }
  QUESTION_RECORD
    {
    QUESTION_NAME  = <Encoded NBT name to be registered>
    QUESTION_TYPE  = NB (0x0020)
    QUESTION_CLASS = IN (0x0001)
    }
```

```
ADDITIONAL_RECORD
  {
  RR_NAME  = 0xC00C (Label String Pointer to QUESTION_NAME)
  RR_TYPE  = NB (0x0020)
  RR_CLASS = IN (0x0001)
  TTL      = <Zero for broadcast, about three days for unicast>
  RDLENGTH = 6
  RDATA
    {
    NB_FLAGS
      {
      G   = <TRUE for a group name, FALSE for a unique name>
      ONT = <Owner type>
      }
    NB_ADDRESS = <Requesting node's IP address>
    }
  }
}
```

The NAME REGISTRATION REQUEST includes both a QUESTION_RECORD and an ADDITIONAL_RECORD. In a sense, it is two messages in one. It says "Does anyone own this name?" and "I want to own this name!", both in the same packet.

The NAME REGISTRATION REQUEST gives us our first look at a Label String Pointer in its native habitat. In the packet above the QUESTION_NAME and the RR_NAME are the same name, so the latter field contains a pointer back to the former. The size of the header is constant; if there is a QUESTION_NAME in a packet it will always be found at offset 0x000C (12). The field value is 0xC00C because (as is always the case with Label String Pointers) the first two bits are set in order to indicate that the remainder is a pointer rather than a 6-bit label length. So, Label String Pointers in NBT messages always have the value 0xC00C.

The TTL field in the ADDITIONAL_RECORD provides a Time-To-Live value, in seconds, for the name. In B mode, the TTL value is not significant and is generally set to zero. In P mode, the TTL is used by the NBNS to determine when to purge old entries from the database, and is typically set to something on the order of three days in the NAME REGISTRATION REQUEST. The NBNS may override the client's request and reply with a different TTL value, which the client must accept.

The ADDITIONAL_RECORD.RDATA field is 6 bytes long (as shown in ADDITIONAL_RECORD.RDLENGTH) and contains two subfields. The first

is the NB_FLAGS field, which provides information about the name and its owner. It looks something like this:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
G	ONT		UNUSED												

The NB_FLAGS.G bit indicates whether the name is a group name or a unique name, and NB_FLAGS.ONT identifies the owner node type. ONT is a two-bit field with the following possible values:

00 == B node
01 == P node
10 == M node
11 == H node (added by Microsoft)

The ADDITIONAL_RECORD.RDATA.NB_ADDRESS holds the 4-byte IPV4 address that will be mapped to the name. This should, of course, match the address of the node registering the name.

Take a good look at the structure of the RDATA subrecord in the NAME REGISTRATION REQUEST. This is the most common RDATA format, which gives us an excuse for writing a little more code...

Listing 4.6a: RDATA Address Records: NS_RDaddr.h

```
/* RDATA NB_FLAGS. */
#define GROUP_BIT     0x8000  /* Group indicator      */
#define ONT_B         0x0000  /* Broadcast node       */
#define ONT_P         0x2000  /* Point-to-point node  */
#define ONT_M         0x4000  /* Mixed mode node      */
#define ONT_H         0x6000  /* MS Hybrid mode node  */
#define ONT_MASK      0x6000  /* Mask                 */

/* RDATA NAME_FLAGS. */
#define DRG           0x0100  /* Deregister.          */
#define CNF           0x0800  /* Conflict.            */
#define ACT           0x0400  /* Active.              */
#define PRM           0x0200  /* Permanent.           */
```

Listing 4.6b: RDATA Address Records: `NS_RDaddr.c`

```
#include <string.h>     /* For memcpy() */
#include <netinet/in.h> /* htons(), ntohs(), etc. */

#include "NS_RDaddr.h"

int Put_RDLength( uchar *rrec,
                  int    offset,
                  ushort rdlen )
  /* ------------------------------------------------- **
   * Set the value of the RDLENGTH field.
   * ------------------------------------------------- **
   */
  {
  rdlen = htons( rdlen );

  (void)memcpy( &(rrec[offset]), &rdlen, 2 );
  return( 2 );
  } /* Put_RDLength */

int Put_RD_Addr( uchar         *rrec,
                 int            offset,
                 ushort         nb_flags,
                 struct in_addr nb_addr )
  /* ------------------------------------------------- **
   * Write IP NB_FLAGS and NB_ADDRESS fields to the
   * packet buffer.
   *
   * See inet(3) on any Linux/Unix/BSD system for more
   * information on 'struct in_addr'.
   * ------------------------------------------------- **
   */
  {
  nb_flags = htons( nb_flags );
  (void)memcpy( &(rrec[offset]), &nb_flags, 2 );
  (void)memcpy( &(rrec[offset+2]), &nb_addr.s_addr, 4 );
  return( 6 );
  } /* Put_RD_Addr */

ushort Get_RDLength( const uchar *rrec, int offset )
  /* ------------------------------------------------- **
   * Read the RDLENGTH field to find out how big the
   * RDATA field is.
   * ------------------------------------------------- **
   */
```

```
  {
  ushort tmp;

  (void)memcpy( &tmp, &(rrec[offset]), 2 );
  return( ntohs( tmp ) );
  } /* Get_RDLength */

ushort Get_RD_NB_Flags( const uchar *rrec, int offset )
  /* ------------------------------------------------- **
   * Read the NB_FLAGS field from an RDATA record.
   * ------------------------------------------------- **
   */
  {
  ushort tmp;

  (void)memcpy( &tmp, &(rrec[offset]), 2 );
  return( ntohs( tmp ) );
  } /* Get_RD_NB_Flags */

struct in_addr Get_RD_NB_Addr( const uchar *rrec, int offset )
  /* ------------------------------------------------- **
   * Read the NB_ADDRESS field from an RDATA record.
   * ------------------------------------------------- **
   */
  {
  ulong          tmp;
  struct in_addr tmp_addr;

  (void)memcpy( &tmp, &(rrec[offset]), 4 );
  tmp_addr.s_addr = ntohl( tmp );
  return( tmp_addr );
  } /* Get_RD_NB_Addr */
```

4.3.1.1 Broadcast Name Registration

You've seen the basic form of `NAME REGISTRATION REQUEST` packet. When sending a broadcast registration, the following rules apply.

- The `B` bit is set.
- The `TTL` is zero.
- The `RDATA.NB_FLAGS.ONT` should never be `ONT_P`, since P nodes never register their names via broadcast.

A node sending a broadcast NAME REGISTRATION REQUEST (the *requester*) may receive a unicast NEGATIVE NAME REGISTRATION RESPONSE from another node that already claims ownership of the name (the *owner*). That is the only valid message in response to a broadcast registration.

```
NAME REGISTRATION RESPONSE (Negative)
  {
  HEADER
    {
    NAME_TRN_ID = <Must match REQUEST transaction ID>
    FLAGS
      {
      R      = TRUE (1; This is a response packet)
      OPCODE = 0x5 (Registration)
      AA     = TRUE (1)
      RD     = TRUE (1)
      RA     = TRUE (1)
      RCODE  = ACT_ERR (0x6)
      B      = FALSE (0; Message is unicast back to requester)
      }
    ANCOUNT = 1
    }
  ANSWER_RECORD
    {
    RR_NAME  = <The Encoded NBT Name>
    RR_TYPE  = NB (0x0020)
    RR_CLASS = IN (0x0001)
    TTL      = 0 (TTL has no meaning in this context)
    RDLENGTH = 6
    RDATA
      {
      NB_FLAGS
        {
        G   = <TRUE for a group name, FALSE for a unique name>
        ONT = <Owner type>
        }
      NB_ADDRESS = <Owner's IP address>
      }
    }
  }
```

When a requester receives a NEGATIVE NAME REGISTRATION RESPONSE, it is obliged to give up. Registration has failed because another node has prior — and conflicting — claim to the name. That is, the name already has an owner.

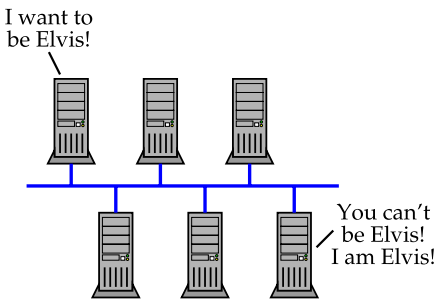


Figure 4.1a: *Broadcast unique/unique name conflict*

A unique name may not be registered if another node already owns that unique name.

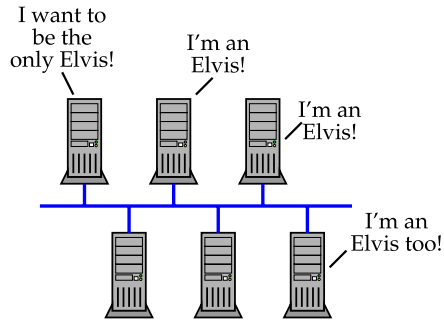


Figure 4.1b: *Broadcast unique/group name conflict*

A unique name may not be registered if the same name is registered as a group name.

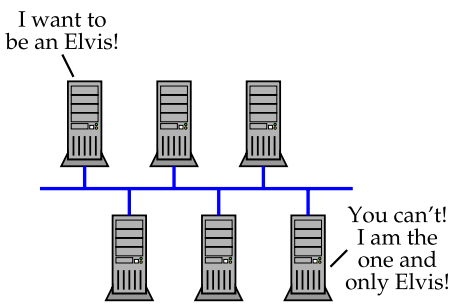


Figure 4.1c: *Broadcast group/unique name conflict*

A group name may not be registered if another node already owns the name as a unique name.

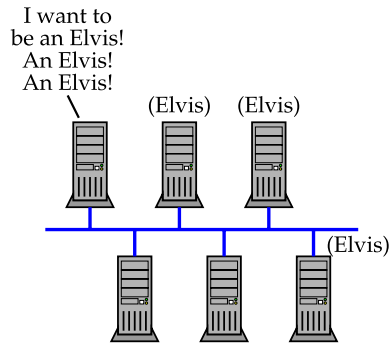


Figure 4.1d: *No conflict when joining a group*

Any node may join a group. Existing group members will not respond to the registration request.

The RCODE field of the response will be ACT_ERR (0x6), indicating that the name is in use. The RDATA field should contain the real owner's name information:

- NB_FLAGS.G indicates whether the name in use is a group or unique name,

- NB_FLAGS.ONT is the owner's node type,
- NB_ADDRESS is the owner's IP address.

Recall that the NAME REGISTRATION REQUEST contains a name query, so the ANSWER_RECORD in the reply should be constructed as it would be in a POSITIVE NAME QUERY RESPONSE. It is wrong to simply parrot back the information in the request.[3]

NEGATIVE NAME REGISTRATION RESPONSE messages are only sent if a unique name is involved.[4] Owners of a group name will not complain if a requester tries to join the group. If, however, a requester tries to register a unique name that matches an already registered group name, the members of the group will send negative responses. In a broadcast environment, a single unique name registration request can generate a large number of negative replies.

If there are no conflicts the requesting node will hear no complaints, in which case it must retry the request two more times... just to be sure. The RFCs specify a minimum timeout of 250 milliseconds between broadcast retries (Windows uses 750 ms). After the third query has timed out, the requesting node should broadcast a NAME OVERWRITE DEMAND declaring itself the victor and owner of the name. The NAME OVERWRITE DEMAND message is identical to the NAME REGISTRATION REQUEST, except that the RD bit is clear (Recursion Desired is 0).

This next program will allow you to play around with broadcast name registration. It uses functions and constants from previous listings to format a NAME REGISTRATION REQUEST and broadcast it on the local IP subnet, then it listens for and reports any replies it receives.

---

3. It is easy, but wrong, to simply copy back the information from the ADDITIONAL_RECORD of the NAME REGISTRATION REQUEST. The NEGATIVE NAME REGISTRATION RESPONSE should identify the node that currently owns the name. (...And yes, some day I may fix this in Samba.)

4. Elvis is the name of a popular clone of the venerable "vi" text editor.

Listing 4.7: A broadcast name registration

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/poll.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "NS_Header.h"
#include "NS_Qrec.h"
#include "NS_Rrec.h"
#include "NS_RDaddr.h"

#define NBT_BCAST_ADDR "255.255.255.255"

#define uchar  unsigned char
#define ushort unsigned short

int BuildRegMsg( uchar         *msg,
                 const uchar   *name,
                 struct in_addr addr )
  /* ------------------------------------------------- **
   * Create a Bcast Name Registration Message.
   *
   * This function hard-codes several values.
   * Obviously, a "real" implementation would need
   * to be much more flexible.
   * ------------------------------------------------- **
   */
  {
  ushort *hdr = (ushort *)msg;
  uchar  *rrec;
  ushort  flags;
  int     len;
  int     rr_len;

  flags = OPCODE_REGISTER | NM_RD_BIT | NM_B_BIT;

  Put_NS_TID( hdr, 1964 );
  Put_NS_Hdr_Flags( hdr, flags );
  Put_NS_Hdr_Rec_Counts( hdr, (QUERYREC | ADDREC) );
  len = 12;     /* Fixed size of header. */
```

```
  len += Put_Qrec( &msg[len],  /* Query Rec Pointer  */
                   name,       /* NetBIOS name       */
                   ' ',        /* Padding char       */
                   '\0',       /* Suffix             */
                   "",         /* Scope ID           */
                   QTYPE_NB ); /* Qtype: Name        */

  rrec = &msg[len];
  rr_len  = Put_RRec_LSP( rrec, RRTYPE_NB );
  rr_len += Put_RRec_TTL( rrec, rr_len, 0 );
  rr_len += Put_RDLength( rrec, rr_len, 6 );
  rr_len += Put_RD_Addr(  rrec, rr_len, ONT_B, addr );

  return( len + rr_len );
  } /* BuildRegMsg */

void ReadRegReply( int sock )
  /* ------------------------------------------------------ **
   * Read a reply packet, and verify that it contains the
   * expected RCODE value.
   * ------------------------------------------------------ **
   */
  {
  uchar  bufr[512];
  int    msglen;
  ushort flags;

  msglen = recv( sock, bufr, 512, 0 );
  if( msglen < 0 )
    {
    perror( "recv()" );
    exit( EXIT_FAILURE );
    }

  if( msglen < 12 )
    {
    printf( "Truncated reply received.\n" );
    exit( EXIT_FAILURE );
    }

  flags = Get_NS_Hdr_Flags( (ushort *)bufr );
```

```
  switch( RCODE_MASK & flags )
    {
    case RCODE_ACT_ERR:
      /* This is the only valid Rcode in response to
       * a broadcast name registration request.
       */
      printf( "RCODE_ACT_ERR: Name is in use.\n" );
      break;
    default:
      printf( "Unexpected return code: 0x%.2x.\n",
              (RCODE_MASK & flags) );
      break;
    }
  } /* ReadRegReply */

int OpenSocket()
  /* ------------------------------------------------------ **
   * Open the UDP socket, enable broadcast, and bind the
   * socket to a high-numbered UDP port so that we can
   * listen for replies.
   * ------------------------------------------------------ **
   */
  {
  int                s;
  int                test = 1;
  struct sockaddr_in sox;

  s = socket( PF_INET, SOCK_DGRAM, IPPROTO_UDP );
  if( s < 0 )
    {
    perror( "socket()" );
    exit( EXIT_FAILURE );
    }

  if( setsockopt( s, SOL_SOCKET, SO_BROADCAST,
                  &test, sizeof(int) ) < 0 )
    {
    perror( "setsockopt()" );
    exit( EXIT_FAILURE );
    }

  sox.sin_addr.s_addr = INADDR_ANY;
  sox.sin_family      = AF_INET;
  sox.sin_port        = 0;  /* 0 == any port */
  test = bind( s, (struct sockaddr *)&sox,
               sizeof(struct sockaddr_in) );
```

```
  if( test < 0 )
    {
    perror( "bind()" );
    exit( EXIT_FAILURE );
    }

  return( s );
  } /* OpenSocket */

void SendBcastMsg( int sock, uchar *msg, int msglen )
  /* ------------------------------------------------- **
   * Nice front-end to the sendto(2) function.
   * ------------------------------------------------- **
   */
  {
  int                result;
  struct sockaddr_in to;

  if( 0 == inet_aton( NBT_BCAST_ADDR, &(to.sin_addr) ) )
    {
    printf( "Invalid destination IP address.\n" );
    exit( EXIT_FAILURE );
    }
  to.sin_family = AF_INET;
  to.sin_port   = htons( 137 );
  result = sendto( sock, (void *)msg, msglen, 0,
                   (struct sockaddr *)&to,
                   sizeof(struct sockaddr_in) );
  if( result < 0 )
    {
    perror( "sendto()" );
    exit( EXIT_FAILURE );
    }
  } /* SendBcastMsg */

int AwaitResponse( int sock, int milliseconds )
  /* ------------------------------------------------- **
   * Wait for an incoming message.
   * One ms == 1/1000 second.
   * ------------------------------------------------- **
   */
  {
  int           result;
  struct pollfd pfd[1];
```

```
  pfd->fd     = sock;
  pfd->events = POLLIN;
  result = poll( pfd, 1, milliseconds );
  if( result < 0 )
    {
    perror( "poll()" );
    exit( EXIT_FAILURE );
    }

  return( result );
  } /* AwaitResponse */

int main( int argc, char *argv[] )
  /* ------------------------------------------------------ **
   * This program demonstrates a Broadcast NBT Name
   * Registration.
   * ------------------------------------------------------ **
   */
  {
  int             i;
  int             result;
  int             ns_sock;
  int             msg_len;
  uchar           bufr[512];
  uchar          *name;
  struct in_addr address;

  if( argc != 3 )
    {
    printf( "Usage:  %s <name> <IP>\n", argv[0] );
    exit( EXIT_FAILURE );
    }

  name = (uchar *)argv[1];
  if( 0 == inet_aton( argv[2], &address ) )
    {
    printf( "Invalid IP.\n" );
    printf( "Usage:  %s <name> <IP>\n", argv[0] );
    exit( EXIT_FAILURE );
    }

  ns_sock = OpenSocket();

  msg_len = BuildRegMsg( bufr, name, address );
```

```
for( i = 0; i < 3; i++ )
  {
  printf( "Trying...\n" );
  SendBcastMsg( ns_sock, bufr, msg_len );
  result = AwaitResponse( ns_sock, 750 );
  if( result )
    {
    ReadRegReply( ns_sock );
    exit( EXIT_FAILURE );
    }
  }
printf( "Success: No negative replies received.\n" );

/* Turn off RD bit for NAME OVERWRITE DEMAND. */
Put_NS_Hdr_Flags( (ushort *)bufr,
                  OPCODE_REGISTER | NM_B_BIT );
SendBcastMsg( ns_sock, bufr, msg_len );

close( ns_sock );
return( EXIT_SUCCESS );
} /* main */
```

The transaction ID in the NAME_TRN_ID field should be the same for all three registration attempts, for the final NAME OVERWRITE DEMAND, and for any negative response packets a remote node may care to send. All of these are part of the same transaction.

Blue Screen of Death Alert

Some OEM versions of Windows 95 had a bug that would cause the system to go into "Blue Screen of Death" mode (that is, system crash) if the NetBIOS Machine Name was in conflict. The problem was made worse by PC vendors who would ship systems with NBT turned on, all preconfigured with the same name. Customers who purchased several computers for local networks would turn them on for the first time and all but one would crash.

4.3.1.2 Unicast (NBNS) Name Registration

Unicast name registrations are subtly different from the broadcast variety.

- The B bit is cleared (zero) and the destination IP is the unicast address of the NBNS.

The message is sent "point-to-point" directly to the NBNS, rather than being broadcast on the local LAN. This is the fundamental difference between B and P modes.

- The TTL field has real meaning when you are talking to an NBNS.

  The RFCs do not specify a default TTL value. Windows systems use 300,000 seconds, which is three days, eleven hours and twenty minutes. Samba uses 259,200 seconds, which is three days even. Both of these values are ugly in hex.[5]

- The timeout between retries is longer.

  The longer timeout between retries is based on the assumption that routed links may have higher latency than the local LAN. RFC 1002 specifies a timeout value of five seconds, which is excessive on today's Internet. A client will try to register a name three times, so the total (worst case) timeout would be fifteen seconds. Samba uses a two second per-packet timeout instead, for a total of six seconds. The timeout under Windows is only 1.5 seconds per packet.

The NBNS should respond with a NAME REGISTRATION RESPONSE, which will include one of the following RCODE values:

0x0: Success

POSITIVE NAME REGISTRATION RESPONSE: You win! The NBNS has accepted the registration. Do not forget to send a refresh before the TTL expires (see Section 4.3.3 on page 98).

FMT_ERR (0x1): Format Error

The NBNS did not like your message. Something was wrong with the packet format (perhaps it was mangled on the wire).

SRV_ERR (0x2): Server failure

The NBNS is sick and cannot handle requests just now.

---

5. 3 days 00:00:00 == 259,200 seconds == 0x0003F480 (Samba),
   3 days 11:20:00 == 300,000 seconds == 0x000493E0 (Windows),
   3 days 19:01:20 == 327,680 seconds == 0x00050000.

IMP_ERR (0x4): Unsupported request error

This one is a bit of a mystery. It basically means that the NBNS does not know how to handle a request. The only clue we have to its intended usage is a poorly worded note in RFC 1002, which says:

```
Allowable only for challenging NBNS when gets an Update
type registration request.
```

Huh?

This error occurs only under odd circumstances, which will be explained in more detail later on in this section. Basically, though, an IMP_ERR should only be returned by an NBNS if it receives an unsolicited NAME UPDATE REQUEST from a client. (Be patient, we'll get there.)

RFS_ERR (0x5): Refused error

This indicates that the NBNS has made a policy decision not to register the name.

ACT_ERR (0x6): Active error

The NBNS has verified that the name is in use by another node. You can't have it.

Note that the difference between a positive and negative NAME REGISTRATION RESPONSE is simply the RCODE value.

If you get no response then it is correct to assume that the NBNS is "down." If the name cannot be registered then your node does not own it, and your application should recover as gracefully as possible. In P mode, handle a non-responsive NBNS as you would a NEGATIVE NAME REGISTRATION RESPONSE. (If the client is running in H or M mode, then it may — with caution — revert to B mode operation until the NBNS is available again.)

There are two other packet types that you may receive when registering a name with an NBNS. These are WACK and END-NODE CHALLENGE NAME REGISTRATION RESPONSE. The WACK message tells the client to wait while the NBNS figures things out. This is typically done so that the NBNS has time to send queries to another node that has claimed ownership of the requested name. A WACK looks like this:

```
WAIT FOR ACKNOWLEDGEMENT (WACK) RESPONSE
  {
  HEADER
    {
    NAME_TRN_ID = <Must match REQUEST transaction ID>
    FLAGS
      {
      R      = TRUE (1; This is a response packet)
      OPCODE = 0x7 (WACK)
      AA     = TRUE (1)
      }
    ANCOUNT = 1
    }
  ANSWER_RECORD
    {
    RR_NAME  = <The Encoded NBT Name from the request>
    RR_TYPE  = NB (0x0020; note the typo in RFC 1002, 4.2.16)
    RR_CLASS = IN (0x0001)
    TTL      = <Number of seconds to wait; 0 == Infinite>
    RDLENGTH = 2
    RDATA    = <Copy of the two-byte HEADER.FLAGS field
               of the original request>
    }
  }
```

The key field in the WACK is the TTL field, which tells the client how long to wait for a response. This is used to extend the timeout period on the client, and give the NBNS a chance to do a reality check.

Samba uses a TTL value of 60 seconds, which provides ample time to generate a proper reply. Unless it is shut down after sending the WACK message, Samba's NBNS service will always send a NAME REGISTRATION RESPONSE (positive or negative) well before the 60 seconds has elapsed. Microsoft's WINS takes a different approach, using a value of only 2 seconds. If the 2 seconds expire, however, the requesting client will simply send another NAME REGISTRATION REQUEST, and then another for a total of three tries. WINS should be able to respond within that total timeframe.

WACK messages are sent by honest, hard-working servers that take good care of their clients. In contrast, a lazy and careless NBNS server will send an END-NODE CHALLENGE NAME REGISTRATION RESPONSE. This latter response tells the client that the requested name has a registered owner, but the NBNS is not going to bother to do the work to check that the owner is still up and running and using the name.

Once again, the format of this message is so familiar that there is no need to list all of the fields. The END-NODE CHALLENGE NAME REGISTRATION RESPONSE packet is just a NAME REGISTRATION RESPONSE with:

```
               RCODE = 0x0
                  RA = 0 (Recursion Available clear)
ANSWER_RECORD.RDATA = <Information retrieved from the NBNS database>
```

The annoying thing about this packet is that the RCODE value indicates success, making it look almost exactly like a POSITIVE NAME REGISTRATION RESPONSE. *The* RA *bit must be checked to distinguish between the two message types.*

When a client receives an END-NODE CHALLENGE, its duty is to query the owner (the owner's IP address will be in the ANSWER_RECORD.RDATA.NB_ADDRESS field) to see if the owner still wants the name. If the owner does not respond, or if it replies with a NEGATIVE NAME QUERY RESPONSE, then the name is available and the requester may send a NAME UPDATE REQUEST to the NBNS. The NBNS will blindly trust the requester, change the entry, and reply with a POSITIVE NAME REGISTRATION RESPONSE. The NAME UPDATE REQUEST is the same as the unicast NAME REGISTRATION REQUEST except that the RD bit is clear (Recursion Desired is 0).

There is nothing to stop a client from skipping the name query and sending the update message to the NBNS, effectively stealing the name. This is why the RFCs use the term *non-secured* when describing this mechanism.

Terminology Turmoil Alert

*In the RFCs, the terms "*NAME UPDATE REQUEST*" and "*NAME OVERWRITE REQUEST & DEMAND*" are both used to refer to the same packet structure. These terms are interchanged somewhat randomly in the text without any explanation regarding their relationship to one another (all probably due to an editing oversight). This is confusing.*

*In this book we make a semantic distinction between the two message types, and shorten "*NAME OVERWRITE REQUEST & DEMAND*" to simply "*NAME OVERWRITE DEMAND*."*

Here's why:

The RFCs specify that a REQUEST *is a message to which a* RESPONSE *is expected. So, for example, once a* NAME REGISTRATION REQUEST *has been sent the requester must wait a reasonable period of time for a reply, and retry the request twice before giving up. A* DEMAND*, however, never generates a* RESPONSE*. It is simply sent and forgotten so there is no need to wait. Thus, the term "*NAME

OVERWRITE REQUEST & DEMAND" *is contradictory. The message is either a* REQUEST *or a* DEMAND*, but not both.*

To clear things up, we use NAME UPDATE REQUEST *to indicate the packet sent to a non-secured NBNS following a name challenge. The requester expects to receive a* POSITIVE NAME REGISTRATION RESPONSE *in reply to the* NAME UPDATE REQUEST. *In contrast, the* NAME OVERWRITE DEMAND *is sent as the last step in a successful broadcast registration, and no reply is expected.*

Again, these packets all share the same structure as the NAME REGISTRATION REQUEST. *Only the* RD *and* B *flag bits distinguish them syntactically.*

Oh... one more thing. Remember the IMP_ERR return code? It is used to indicate that an NBNS which did *not* send an END-NODE CHALLENGE is annoyed at having received a NAME UPDATE REQUEST from a client. An NBNS server should never receive unsolicited NAME UPDATE REQUESTs from clients.

4.3.1.3 M and H Node Name Registration

Mixed mode (M mode) and Hybrid mode (H mode) are both speed hacks, which combine aspects of Broadcast (B) and Point-to-Point (P) modes to short-cut Name Service operations.

M mode was designed in the days when local LAN traffic was likely to be faster than internetwork links, which were typically carried over leased lines, dial-up connections, tin cans with string, or pigeon (see RFC 1149). Since local broadcasts were both faster and more reliable than traffic to a remote NBNS, M nodes attempt B mode behavior first and try P mode behavior second.

When an M node registers a name, for example, it starts by sending a broadcast NAME REGISTRATION REQUEST. If it receives a negative response it tries no further (thus saving some time). If, however, it receives no complaints after three retries, it will attempt to register with the NBNS as a P node would. If and only if the P mode registration succeeds, the M mode will broadcast a NAME OVERWRITE DEMAND. If the unicast registration fails, the NAME OVERWRITE will not be sent and the node will not assume ownership of the name.

Hybrid mode (H mode) was introduced (probably by Microsoft) after the RFCs were published. H mode assumes that internetwork links are fast and reliable, in which case it makes sense to try P mode behavior first and revert

to B mode behavior only if the NBNS does not respond. Compared with M mode, H mode generates less broadcast traffic on local LANs.

H mode is a little trickier than M mode. A node running in H mode will attempt a unicast name registration and, if the NBNS accepts the registration, the H node will assume ownership without generating any broadcast (B mode) traffic at all. If the NetBIOS vLAN is configured properly all of the nodes within the scope will also be registering with the NBNS, thus preventing accidental name conflicts.

If the NBNS is down or unreachable, however, an H node will revert to B mode behavior and hope that no conflicts will arise when the NBNS comes back.

4.3.1.4 Registering Multi-Homed Hosts

A multi-homed host is a machine that has multiple network interfaces (physical or virtual), each with its own IP address assigned. RFCs 1001 and 1002 do not discuss handling of multi-homed hosts.

The annoying thing about multi-homed hosts in an NBT environment is that they try to register their NetBIOS names on each interface, which means multiple IP addresses per name. This is not a problem for group names because group names map to several IP addresses anyway — that's what NBT group names are all about. Unique names *are* a problem because, from the network's point of view, there is no difference between a multi-homed host and multiple machines. To an NBNS, or to B nodes on a local LAN, multiple registrations for the same name will look like a name conflict.

There are three scenarios to consider when working with multi-homed hosts.

B nodes with interfaces on separate subnets

If each IP address is on a separate IP subnet *and* the node is running in B mode then springtime returns to the cities, birds sing, and little children dance for joy. Each name-to-IP-address mapping is unique *within its NBT scope*, which is the broadcast space within the subnet, so there are no name conflicts.

The only multi-homed-specific problem that can occur in this scenario starts with a regular old-fashioned run-of-the-mill name conflict. If there is a name conflict with another node on one or more, but not all,

subnets then we have a quandary because the name is valid on some subnets, but not others. Two solutions are possible here: the multi-homed host may decide to disable the name on all interfaces (probably the best option), or just on the interfaces on which the conflict exists.

Another thing to keep in mind is that replies to name queries must return the correct IP address for the subnet, so it is important to know on which interface the query was received. This can be done by checking both the source and destination IP addresses of the original query packet. If the query is a broadcast query, then it is best to send only the IP address of the interface. Unicast queries, however, should contain a full list of the IPs registered to the name. This quirk will be examined further when we tackle P mode multi-homed registration.

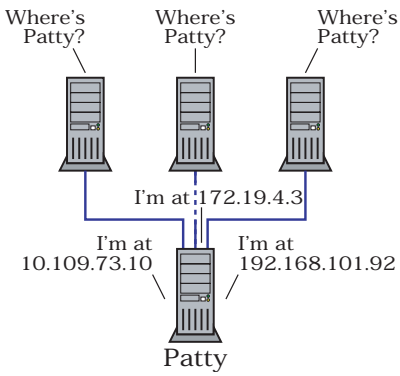


Figure 4.2: *A multi-homed B node*

Node PATTY has three interfaces, each with an IP address on a different subnet. PATTY replies to each broadcast query with the correct IP address for the subnet.

B nodes with interfaces on the same subnet

Problems occur if two or more interfaces have IP addresses on the same subnet. This is equivalent to having two or more separate nodes on the same subnet, all trying to claim the same unique name. There is no standard fix for this situation. Fortunately this configuration is rare, though it *does* occur in the wild — typically when someone tries to build a fault-tolerant or load-balanced server system. The only known work-around is

to write additional code to control which of the multi-homed interfaces "owns" a name at any given time.

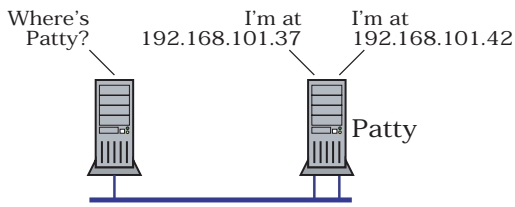


Figure 4.3: *A multi-homed B node with a shared subnet*

In this case, node PATTY has two interfaces, both connected to the same subnet. PATTY sends two correct replies to a broadcast query which looks, on the wire, exactly like a name conflict.

Multi-homed hosts and the NBNS

P mode multi-homed name registration is a circus. In P mode, a multi-homed host will send multiple registrations — one per interface — to the NBNS. Normally the NBNS would reject all but the first such registration, viewing the others as name conflicts. To get around this problem, we use a new OpCode:

0xF == multi-homed name registration.

Instead of sending normal registration requests, the host concurrently sends individual MULTI-HOMED NAME REGISTRATION REQUEST packets from each interface it wishes to register. Other than the OpCode, these are identical to normal NAME REGISTRATION REQUEST packets, though each request has its own NAME_TRN_ID (transaction ID).

The NBNS will respond to the first of these messages by sending a POSITIVE NAME REGISTRATION RESPONSE. It then sends 2-second WACK messages in reply to all the other MULTI-HOMED NAME REGISTRATION REQUEST packets it receives (all that are trying to register the same unique name). The WACK gives the NBNS extra time to process the registration.

Next, the NBNS will send a unicast NAME QUERY REQUEST to the source address of the first message it received (the one that got the POSITIVE NAME REGISTRATION RESPONSE). This is a unicast

query (the B bit is clear), so *the query response should contain the complete list of IP addresses that are allowed to share the name*.

The NBNS will then send POSITIVE NAME REGISTRATION RESPONSE messages to all of the WACKed IPs in the list, and a NEGATIVE NAME REGISTRATION RESPONSE, with an RCODE value of ACT_ERR (0x6), to any others. The NBNS finishes with a double back-flip in pike position through flaming hoops into a piece of unbuttered toast and the crowd cheers wildly.

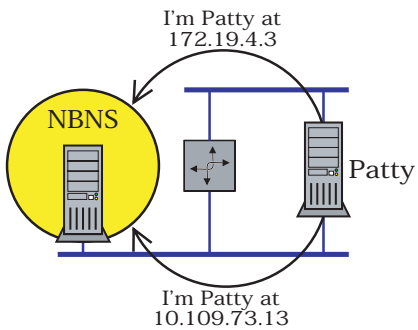


Figure 4.4: *A multi-homed P node*

Node PATTY has two interfaces, each on a separate subnet. PATTY sends separate registrations to the NBNS. Under normal circumstances, this would be handled as a name conflict.

One problem still remains, however. Consider node LANE (operating in P mode), which is trying to talk to node PATTY. The first thing LANE will do is send a NAME QUERY REQUEST to the NBNS. The NBNS has no way of knowing which IP address represents the best route between LANE and PATTY, so it must send the complete list of PATTY's IPs. LANE has to guess which IP is the best. Typically, the client will choose a destination IP by sending some sort of message (e.g., a unicast name query) to all of the listed IPs to see which one answers first. Note that in order to make this work the NBNS must keep track of all IPs associated with the NBT name registered by the multi-homed host.[6]

6. Many thanks to Monyo for providing packet captures.

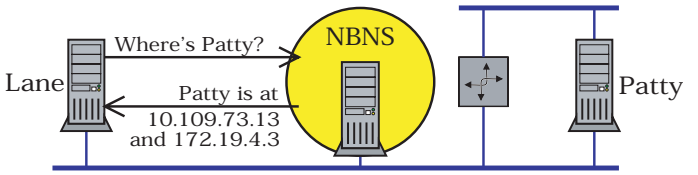


Figure 4.5: *Locating a multi-homed P node*

Node LANE gets two IPs when it asks for PATTY's address.

As you might expect, the handling of M and H mode multi-homed hosts is a fairly straightforward combination of B and P mode behavior. M and H mode name registration for single-homed hosts has already been covered.

4.3.2 *Name Query*

Each NBT node has its own local name table, which holds the list of the NetBIOS names that the node thinks it owns. NBT nodes may also register their names with a NetBIOS nameserver. Both the local name table and the NBNS database can be used to answer queries.

Name queries look like this:

```
NAME QUERY REQUEST
  {
  HEADER
    {
    NAME_TRN_ID = <Set when packet is transmitted>
    FLAGS
      {
      OPCODE = 0x0 (Query)
      RD     = <Typically TRUE (1); see discussion below>
      B      = <TRUE for broadcast queries, else FALSE (0)>
      }
    QDCOUNT = 1
    }
  QUESTION_RECORD
    {
    QUESTION_NAME  = <Encoded NBT name to be queried>
    QUESTION_TYPE  = NB (0x0020)
    QUESTION_CLASS = IN (0x0001)
    }
  }
```

As you can see from the packet description, name queries are really very simple (just as the eye of a hurricane is calm). The only fiddly bits are the B and RD flags.

- The B bit is used to distinguish between broadcast and unicast queries.

  Broadcast queries are used for name resolution within the broadcast scope, as shown by the example code presented earlier. Since P nodes are excluded from B mode scope, P nodes and the NBNS will both ignore broadcast name queries. Only local B, M, and H nodes (with the same Scope ID as the sender) will respond.

  The only valid reply to a broadcast name query is a POSITIVE NAME QUERY RESPONSE from a node that actually owns the name in question. Queries for group names may generate multiple responses, one per group member on the local LAN.

  In P mode, names are resolved to IP addresses by sending a unicast query to the NBNS, which checks the list of registered names (the NBNS database). If the name is found, the NBNS will reply with a POSITIVE NAME QUERY RESPONSE, otherwise it will send a NEGATIVE NAME QUERY RESPONSE. If the requester gets no response at all, then the NBNS is assumed to be down or unreachable.

  Unicast queries can also be used to determine whether an NBT end node owns a given NetBIOS name. All NBT node types (B, P, M, and H) will respond to unicast queries. As with queries sent to the NBNS, NBT end nodes may reply with either a positive or negative NAME QUERY RESPONSE.

  A unicast query for the wildcard name is the NBT equivalent of a `ping`.

- The RD bit is used to distinguish between two different types of unicast queries.

  In discussing the use of the B bit, above, we made a subtle distinction between resolution and verification queries. A *name resolution query* is the most familiar. It is used to convert a name to an IP address. Unicast queries sent to the NBNS are always resolution queries. A *verification query* is a unicast query sent to an NBT end node to ask whether the node is using the name in question. In order to send a verification query, the sender must already have the IP of the target NBT end node so name resolution is pointless.

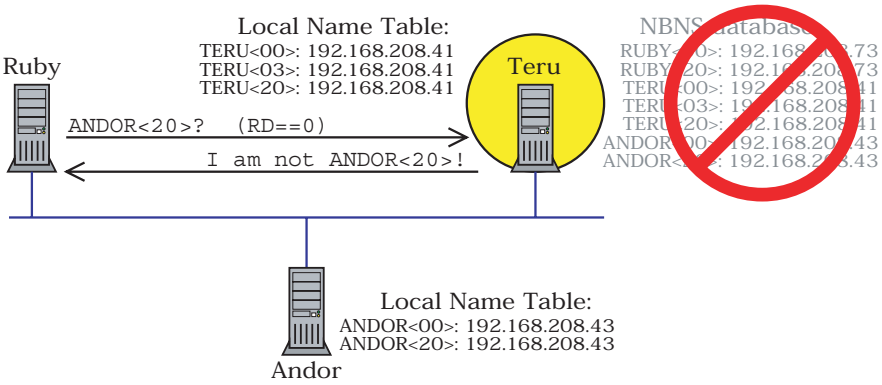


Figure 4.6a: *Verification query (`RD == FALSE`)*

`RUBY` sends a unicast query to node `TERU` asking about `ANDOR`. The `RD` bit is *clear*, so `TERU` does not check the NBNS database. It checks only the local name table and, finding no reference to the name `ANDOR<20>`, sends a `NEGATIVE NAME QUERY RESPONSE`.

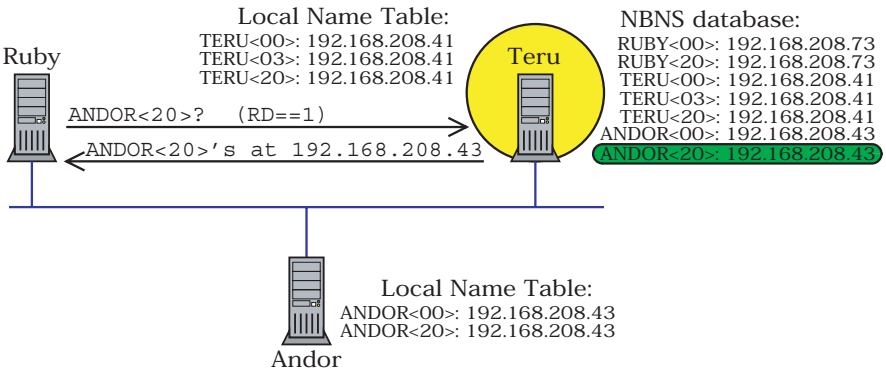


Figure 4.6b: *Verification query (`RD == TRUE`)*

`RUBY` sends a unicast query to node `TERU` asking about `ANDOR`. The `RD` bit is *set*, so `TERU` checks the NBNS database, where it finds an entry for `ANDOR<20>`. `TERU` sends a `POSITIVE NAME QUERY RESPONSE`.

Note that:

- Broadcast *name resolution queries* are always answered from data in the receiving node's local name table.

- Unicast *name resolution queries* are supposed to be answered from data in the NBNS database.
- Unicast *name verification queries* must be answered from the node's local name table — *not* the NBNS database.

So... what happens if you send a unicast query to a node that is both an NBT participant *and* the NBNS? Which kind of query is it, and which name list should be consulted?

That's where the RD bit comes in. If RD is FALSE then only the local name table is consulted, forcing a verification query. If RD is TRUE and the NBNS service is running on the receiving node, then the NBNS database may also be used to answer the query — that makes it a resolution query.

This particular problem, and its solution, are not covered in the RFCs. The diagram in RFC 1002, Section 4.2.12 shows the RD bit as always set, and this is common practice.[7] The state of the RD bit in a query message is typically ignored, and is only significant in the one case we have described: a unicast query sent to a node that is both an NBT participant and the NBNS.

In summary:

```
/* Pseudocode */
if( the B bit is TRUE )
  { /* It's a broadcast query. */
  if( the receiver is a B, M, or H node )
    {
    entry = lookup name in local name table;
    if( entry was found )
      send( POSITIVE NAME QUERY RESPONSE );
    }
  }
```

7. For example, when an NBNS is processing a multi-homed registration it *should* send name queries with the RD bit clear, yet all Windows systems that were tested set the RD bit. It may not matter, however, unless the multi-homed host is also the node running the NBNS, in which case the problem would likely be solved using internal mechanisms (because the NBNS would be sending the query to itself). The right thing to do is to send verification queries with the RD flag turned OFF.

```
else
{ /* It's a unicast query. */
entry = lookup name in local name table;
if( entry was not found & RD is TRUE & receiver is the NBNS )
  {
  entry = lookup name in NBNS database;
  }
if( entry was found )
  send( POSITIVE NAME QUERY RESPONSE );
else
  send( NEGATIVE NAME QUERY RESPONSE );
}
```

Got it? Good. Let's move on...

As with other NBT Name Service requests, if there is no response to a name query within a reasonable timeout period, the query is sent again. This happens twice for a maximum of two retries (that is, three query messages). Timeouts vary from system to system and depend upon the type of query being sent. Query timeouts should be matched to those used for name registration where possible.

Broadcast queries

Between 250 ms and 750 ms is typical. RFC 1002 specifies 250 ms.

Unicast Resolution queries

A range of 1.5 to 2 seconds is common. These queries go to the NBNS, and the expectation is that the NBNS will be able to answer quickly. RFC 1002 specifies 5 seconds.

Verification queries

Intervals of 1.5 to 5 seconds have been spotted. Once again, RFC 1002 specifies 5 seconds.

Timeout values are a balance between reliability and user annoyance. Too short, and replies will be missed. Too long, and the user goes off to make another pot of tea.

4.3.2.1 Negative Query Response

A negative response looks like this:

```
NEGATIVE NAME QUERY RESPONSE
  {
  HEADER
    {
    NAME_TRN_ID = <Same as QUERY REQUEST>
    FLAGS
      {
      R      = TRUE (1; This is a response packet)
      OPCODE = 0x0 (Query)
      AA     = TRUE (1)
      RD     = <Copy RD bit from QUERY REQUEST>
      RA     = <TRUE if the reply is from the NBNS>
      B      = FALSE (0)
      RCODE  = <Error code>
      }
    ANCOUNT = 1
    }
  ANSWER_RECORD
    {
    RR_NAME  = <The Encoded NBT Name from the request>
    RR_TYPE  = <NB (0x0020), or possibly NULL (0x000A)>
    RR_CLASS = IN (0x0001)
    TTL      = 0
    RDLENGTH = 0
    }
  }
```

RFC 1002 is inconsistent in its descriptions of the RD and RA bits as used in NAME QUERY RESPONSE messages. There is also a small issue regarding the RR_TYPE field. Let's clear things up:

- The diagram in RFC 1002, Section 4.2.14, shows RD always set in the reply. Most implementations do, in fact, set the RD bit in all NAME QUERY RESPONSE messages. To be painfully correct, however, the right thing to do is to copy the RD value from the NAME QUERY REQUEST as described in RFC 1002, Section 4.2.1.1. It really doesn't matter, though, because the RD bit is probably ignored by the node receiving the query response.
- Regarding the RA bit: There is a weird little note in RFC 1002, Section 4.2.15, which states:

  > An end node responding to a NAME QUERY REQUEST always responds with the AA and RA bits set for both the NEGATIVE and POSITIVE NAME QUERY RESPONSE packets.

That's poop. The RA bit should not be set by an end-node. Only the NBNS should set the RA bit, as explained in 4.2.1.1:

```
RA    3   Recursion Available Flag.

          Only valid in responses from a NetBIOS Name
          Server - must be zero in all other
          responses.
```

In modern usage, the RA bit *should* mean that the responding node is running the NBNS service.

- The diagram in RFC 1002, Section 4.2.14, specifies that RR_TYPE should have a value of 0x000A (NULL). In practice, the value 0x0020 (NB) is used instead (no exceptions were found in testing).

The NEGATIVE NAME QUERY RESPONSE will include an RCODE value, indicating the reason for the negative reply. RFC 1002 lists several possible RCODE values, but at least two of them — IMP_ERR and RFS_ERR — are incorrect as they are never generated in response to a query. The valid values or a NEGATIVE NAME QUERY RESPONSE are:

FMT_ERR (0x1): Format Error
: The NBNS did not like your message. Something was wrong with the packet format (perhaps it was mangled on the wire).

SRV_ERR (0x2): Server failure
: The NBNS is sick and cannot handle requests just now.

NAM_ERR (0x3): Name Error
: The requested name does not exist in the selected name table(s).

4.3.2.2 Positive Query Response

The POSITIVE NAME QUERY RESPONSE is similar to the negative response, with the following differences:

- The RCODE is 0x0 (success),
- the RR_TYPE field always has a value of 0x0020 (NB),
- the TTL field is non-zero, and
- the RDATA field contains an array of IP address information, like so:

```
POSITIVE NAME QUERY RESPONSE
  {
  HEADER
    {
    NAME_TRN_ID = <Same as QUERY REQUEST>
    FLAGS
      {
      R      = TRUE (1; This is a response packet)
      OPCODE = 0x0 (Query)
      AA     = TRUE (1)
      RD     = <Copy RD bit from QUERY REQUEST>
      RA     = <TRUE if the reply is from the NBNS>
      B      = FALSE (0)
      RCODE  = 0x0
      }
    ANCOUNT = 1
    }
  ANSWER_RECORD
    {
    RR_NAME  = <The Encoded NBT Name from the request>
    RR_TYPE  = NB (0x0020)
    RR_CLASS = IN (0x0001)
    TTL      = <Time To Live>
    RDLENGTH = <6   number of entries>
    RDATA
      {
      ADDR_ENTRY[]
        {
        NB_FLAGS
          {
          G   = <TRUE for a group name, FALSE for a unique name>
          ONT = <Owner type>
          }
        NB_ADDRESS = <Owner's IP address>
        }
      }
    }
  }
```

If the packet is sent by the NBNS, the `TTL` field will contain the number of seconds until the entry's Time-To-Live expires (the remaining TTL). End nodes responding to verification queries will typically use the default `TTL` value which, as we described earlier, is something around 3 days.

4.3.2.3 The Redirect Name Query Response

The RFCs provide a mechanism whereby one NBNS can redirect a client to another NBNS. That is, the NBNS can return a message saying "I don't know, ask someone else."

No living examples of this mechanism have been seen in the wild. It is probably extinct. Fossil remains may be found in RFC 1001, Section 15.1.5.3, and RFC 1002, Section 4.2.15.

4.3.2.4 A Simple Name Query Revisited

Remember Listing 3.3? In that example we provided code for generating a simple broadcast name query. Listing 4.8 provides an updated version which is a bit more flexible. In particular, the `BuildQuery()` function takes several parameters, allowing you to customize the query you want to send. The program mainline, as given, sends only broadcast queries. It can, however, be easily hacked to create a more versitile command-line tool. This new version also listens for replies.

Listing 4.8: Broadcast name query revisited

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/poll.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "NS_Header.h"
#include "NS_Qrec.h"

#define uchar  unsigned char
#define ushort unsigned short

int BuildQuery( uchar         *msg,
                const int      bcast,
                const int      rdbit,
                const uchar   *name,
                const uchar    pad,
                const uchar    suffix,
                const uchar   *scope,
                const ushort   qtype )
```

```
  /* ------------------------------------------------- **
   * Create a name query.
   *
   * This is much more flexible than the registration
   * example in Listing 4.7.  There are also a lot more
   * parameters.  :-)
   * ------------------------------------------------- **
   */
  {
  ushort *hdr = (ushort *)msg;
  ushort  flags;
  int     len;

  /* RD always set if B is set. */
  if( bcast )
    flags = NM_RD_BIT | NM_B_BIT;
  else
    flags = rdbit ? NM_RD_BIT : 0;

  Put_NS_TID( hdr, 1964 );
  Put_NS_Hdr_Flags( hdr, flags );
  Put_NS_Hdr_Rec_Counts( hdr, QUERYREC );
  len = 12;     /* Fixed size of header. */

  len += Put_Qrec( &msg[len], /* Query Rec Pointer */
                   name,      /* NetBIOS name      */
                   pad,       /* Padding char      */
                   suffix,    /* Suffix            */
                   scope,     /* Scope ID          */
                   qtype );   /* Query type        */

  return( len );
  } /* BuildQuery */

void ReadQueryReply( int sock )
  /* ------------------------------------------------- **
   * Read the query reply message(s).
   * ------------------------------------------------- **
   */
  {
  uchar  bufr[512];
  int    msglen;
  ushort flags;

  msglen = recv( sock, bufr, 512, 0 );
```

```
  if( msglen < 0 )
    {
    perror( "recv()" );
    exit( EXIT_FAILURE );
    }

  if( msglen < 12 )
    {
    printf( "Truncated reply received.\n" );
    exit( EXIT_FAILURE );
    }

  flags = Get_NS_Hdr_Flags( (ushort *)bufr );
  switch( RCODE_MASK & flags )
    {
    case RCODE_POS_RSP:
      printf( "Positive Name Query Response.\n" );
      break;
    case RCODE_FMT_ERR:
      printf( "RCODE_FMT_ERR: Format Error.\n" );
      break;
    case RCODE_SRV_ERR:
      printf( "RCODE_SRV_ERR: Server Error.\n" );
      break;
    case RCODE_NAM_ERR:
      printf( "RCODE_NAM_ERR: Name Not Found.\n" );
      break;
    default:
      printf( "Unexpected return code: 0x%.2x.\n",
              (RCODE_MASK & flags) );
      break;
    }

  } /* ReadQueryReply */

int main( int argc, char *argv[] )
  /* ------------------------------------------------- **
   * This program demonstrates a Broadcast NBT Name Query.
   * ------------------------------------------------- **
   */
  {
  int                i;
  int                result;
  int                ns_sock;
  int                msg_len;
  uchar              bufr[512];
  uchar             *name;
```

```
if( argc != 2 )
  {
  printf( "Usage:  %s <name>\n", argv[0] );
  exit( EXIT_FAILURE );
  }

name = (uchar *)argv[1];

ns_sock = OpenSocket();

msg_len = BuildQuery( bufr,  /* Target buffer.   */
                      1,     /* Broadcast true.  */
                      1,     /* RD bit true.     */
                      name,  /* NetBIOS name.    */
                      ' ',   /* Padding (space). */
                      '\0',  /* Suffix (0x00).   */
                      "",    /* Scope ("").      */
                      QTYPE_NB ); /* Query type. */

for( i = 0; i < 3; i++ )
  {
  printf( "Trying...\n" );
  SendBcastMsg( ns_sock, bufr, msg_len );
  result = AwaitResponse( ns_sock, 750 );
  if( result )
    {
    do
      {
      /* We may get multiple replies. */
      ReadQueryReply( ns_sock );
      } while( AwaitResponse( ns_sock, 750 ) );
    exit( EXIT_SUCCESS );
    }
  }
printf( "No replies received.\n" );

close( ns_sock );
return( EXIT_FAILURE );
} /* main */
```

The sweet and chewey center of a POSITIVE NAME QUERY RESPONSE is the RDATA section, which contains an array of address entries. In most cases there will be only one entry, but a group name or a multi-homed host name may have several associated IP addresses. The contents of the ADDR_ENTRY records should be fairly familiar by now, so we won't dwell on

them. Here are some quick functions which can be used to display the IP addresses and NB_FLAGS of an ADDR_ENTRY array:

Listing 4.9: Listing ADDR_ENTRY records

```
#include "NS_RDaddr.h"

int Find_RDLength( uchar *msg )
  /* ------------------------------------------------------ **
   * Calculate the offset of the RDLENGTH field within a
   * POSITIVE NAME QUERY RESPONSE.
   * ------------------------------------------------------ **
   */
  {
  int len;

  len = 12                          /* Length of the header */
      + strlen( &msg[12] ) + 1      /* NBT Name length      */
      + 2 + 2 + 4;                  /* Type, Class, & TTL   */
  return( len );
  } /* Find_RDLength */

void List_Addr_Entry( uchar *msg )
  /* ------------------------------------------------------ **
   * This function nicely prints the contents of an
   * RDATA.ADDR_ENTRY[] array.
   * ------------------------------------------------------ **
   */
  {
  ushort numIPs;
  ushort flags;
  int    offset;
  int    i;

  offset = Find_RDLength( msg );
  numIPs = Get_RDLength( msg, offset ) / 6;
  offset += 2;  /* Move past the RDLENGTH field. */

  for( i = 0; i < numIPs ; i++, offset += 6 )
    {
    /* Read the NB_FLAGS field. */
    flags = Get_RD_NB_Flags( msg, offset );

    /* If there are more than one, number the entries. */
    if( numIPs > 1 )
      printf( "ADDR_ENTRY[%d]: ", i );
```

```
    /* Print the IP address. */
    printf( "%d.%d.%d.%d\t",
            msg[offset+2], msg[offset+3],
            msg[offset+4], msg[offset+5] );

    /* Group or Unique. */
    if( GROUP_BIT & flags )
      printf( "<Group>\t" );
    else
      printf( "<Unique>\t" );

    /* Finally, the owner node type. */
    switch( ONT_MASK & flags )
      {
      case ONT_B: printf( "<B-node>\n" ); break;
      case ONT_P: printf( "<P-node>\n" ); break;
      case ONT_M: printf( "<M-node>\n" ); break;
      case ONT_H: printf( "<H-node>\n" ); break;
      }
    }
  } /* List_Addr_Entry */
```

4.3.3 *Name Refresh*

Name refresh has two purposes. The first is to remind the NBNS that the client exists, thus ensuring that the name entry in the NBNS database does not expire. The second is to rebuild the NBNS database in the event of an NBNS crash. `NAME REFRESH REQUEST` messages are not needed in B mode since each node keeps track of its own names.

```
NAME REFRESH REQUEST
  {
  HEADER
    {
    NAME_TRN_ID = <Set when packet is transmitted>
    FLAGS
      {
      OPCODE = <0x8 or 0x9> (Refresh)
      RD     = FALSE (0)
      B      = FALSE (0)
      }
    QDCOUNT = 1
    ARCOUNT = 1
    }
```

```
QUESTION_RECORD
  {
  QUESTION_NAME  = <Encoded NBT name to be refreshed>
  QUESTION_TYPE  = NB (0x0020)
  QUESTION_CLASS = IN (0x0001)
  }
ADDITIONAL_RECORD
  {
  RR_NAME  = 0xC00C (Label String Pointer to QUESTION_NAME)
  RR_TYPE  = NB (0x0020)
  RR_CLASS = IN (0x0001)
  TTL      = <Client's default TTL value (3 days)>
  RDLENGTH = 6
  RDATA
    {
    NB_FLAGS
      {
      G   = <TRUE for a group name, FALSE for a unique name>
      ONT = <Owner type>
      }
    NB_ADDRESS = <Requesting node's IP address>
    }
  }
}
```

This message is almost identical to the unicast NAME REGISTRATION REQUEST, with a few small exceptions. Note, in particular, the following:

OPCODE

The NAME REFRESH REQUEST packet uses the Refresh OpCode. Due to a typo in RFC 1002, the OPCODE values 0x8 and 0x9 are considered equivalent and both mean NAME REFRESH REQUEST. 0x8 is more commonly used.

RD

The RD field is set to FALSE, which is a little strange since the NAME REFRESH REQUEST deals directly with the NBNS.

TTL

The TTL field typically contains the client's default TTL value — the same value used in the NAME REGISTRATION REQUEST. Once again, the NBNS has the right to override the client's TTL value in the TTL field of the response.

RDATA

The RDATA should match the data stored by the NBNS. If not, the NBNS will treat the request as if it were a registration request. If the refresh RDATA conflicts with the existing data, the NBNS may need to send a query to validate the older information in its database.

From watching packets on the wire,[8] it seems that Windows systems use the following formula to determine how frequently a refresh message should be sent:

```
Refresh_Time = minimum( 40 minutes, (TTL/2) )
```

Based on the above formula, and considering that the default TTL value used by most clients is about three days, Windows NBNS clients typically send NAME REFRESH REQUEST messages every 40 minutes. This is a fairly high frequency, and it suggests a general lack of faith in the stability of the NBNS.[9]

The NBNS handles a NAME REFRESH REQUEST in exactly the same manner as it handles a NAME REGISTRATION REQUEST. There is little reason to distinguish between the two message types. Indeed, there is no multi-homed variant of the refresh message so multi-homed hosts perform the refresh operation by sending MULTI-HOMED NAME REGISTRATION REQUEST messages.

4.3.4 *Name Release*

Both B and P nodes (and their hybrid offspring, the M and H nodes) send NAME RELEASE messages to announce that they are giving up ownership of a name.

---

8. Many thanks to Jean François for all of his work on WINS behavior and TTL gymnastics.

9. Microsoft may be assuming that the NBNS service is being provided by their own WINS implementation. Samba's NBNS, which is part of the nmbd daemon, periodically writes the contents of its database to a file called wins.dat. The wins.dat file is re-read at startup, and any non-expired names are placed back into the database. This prevents data loss due to a system restart. Samba sends refreshes every TTL/2 seconds, and there have been reports of Samba server names "disappearing" from WINS databases following a Windows system crash. It is likely that newer versions of Samba (V3.0 and beyond) will use Microsoft's formula for calculating name refresh time.

A NAME RELEASE sent in B mode is a NAME RELEASE DEMAND, as no response is expected. Any node receiving the release message will flush the released name from its local cache (if it has one[10]). In P mode, the release message sent by a node is a NAME RELEASE REQUEST, and it is always unicast to the NBNS. The message structure is the same in both cases:

```
NAME RELEASE REQUEST or NAME RELEASE DEMAND
  {
  HEADER
    {
    NAME_TRN_ID = <Set when packet is transmitted>
    FLAGS
      {
      OPCODE = 0x6 (Release)
      B      = <FALSE (0) for REQUEST, TRUE (1) for DEMAND>
      }
    QDCOUNT = 1
    ARCOUNT = 1
    }
  QUESTION_RECORD
    {
    QUESTION_NAME  = <Encoded NBT name to be released>
    QUESTION_TYPE  = NB (0x0020)
    QUESTION_CLASS = IN (0x0001)
    }
  ADDITIONAL_RECORD
    {
    RR_NAME  = 0xC00C (Label String Pointer to QUESTION_NAME)
    RR_TYPE  = NB (0x0020)
    RR_CLASS = IN (0x0001)
    TTL      = 0 (zero)
    RDLENGTH = 6
    RDATA
      {
      NB_FLAGS
        {
        G   = <TRUE for a group name, FALSE for a unique name>
        ONT = <Owner type>
        }
      NB_ADDRESS = <Releasing node's IP address>
      }
    }
  }
```

10. Windows systems typically cache resolved names for about seven minutes. Use the `nbtstat -c` command from the DOS prompt to see the cache contents.

4.3.4.1 Name Release Response

The NBNS will always respond to a NAME RELEASE REQUEST. The response packet looks like this:

```
NAME RELEASE RESPONSE
  {
  HEADER
    {
    NAME_TRN_ID = <Must match REQUEST transaction ID>
    FLAGS
      {
      R      = TRUE (1; This is a response packet)
      OPCODE = 0x6 (Release)
      AA     = TRUE (1)
      RCODE  = <See discussion>
      B      = FALSE (0)
      }
    ANCOUNT = 1
    }
  ANSWER_RECORD
    {
    RR_NAME  = <The Released Name, encoded as usual>
    RR_TYPE  = NB (0x0020)
    RR_CLASS = IN (0x0001)
    TTL      = 0 (TTL has no meaning in this context)
    RDLENGTH = 6
    RDATA    = <Same as request packet>
    }
  }
```

Possible values for RCODE are:

0x0: Success

POSITIVE NAME RELEASE RESPONSE. The name entry has been removed from the NBNS database.

FMT_ERR (0x1): Format error

Something got messed up, and the NBNS couldn't understand the request.

SRV_ERR (0x2): Server failure

The NBNS is sick and cannot handle requests just now.

NAM_ERR (0x3): Name error

The name does not exist in the NBNS database or, if the name exists, the NB_FLAGS did not match (so it's not really the same name).

RFS_ERR (0x5): Refused error

The NBNS has made a policy decision not to release the name. For some reason, the end node that sent the request does not have authority to remove it.

ACT_ERR (0x6): Active error

The name was found in the database, but the NB_ADDRESS field did not match. Another node owns the name, so your node may not release it.

4.3.5 *Node Status*

The Node Status Request operation goes by many names: "Node Status Query," "Adapter Status Query," "NBSTAT," etc. This NBT message is used to implement the old NetBIOS Adapter Status command, which was used to retrieve information from LAN Adapter cards (LANAs, in PC Network terms).

```
NODE STATUS REQUEST
  {
  HEADER
    {
    NAME_TRN_ID = <Set when packet is transmitted>
    FLAGS
      {
      OPCODE = 0x0 (Query)
      B      = FALSE (0)
      }
    QDCOUNT = 1
    }
  QUESTION_RECORD
    {
    QUESTION_NAME  = <Encoded NBT name to be queried>
    QUESTION_TYPE  = NBSTAT (0x0021)
    QUESTION_CLASS = IN (0x0001)
    }
  }
```

Note that these queries are sent from one end node to another. The NBNS is never involved. This is because the NBNS itself is not connected to an NBT

virtual LAN Adapter. The NBNS is part of the infrastructure that *creates* the NetBIOS virtual LAN. Only the end nodes are actually *members of* the LAN.

4.3.5.1 Node Status Response

The response is not as simple as the query. The format of the reply depends upon the type of card and/or virtual adapter used to build the network. In the old days, different implementations of NetBIOS were built on top of different LANAs, or emulated on top of a variety of underlying transport protocols. Each implementation kept track of its own set of status information, so the reply to the Adapter Status command was vendor-specific.

The RFC authors developed their own reply structure, probably based in part on existing samples. The `NODE STATUS RESPONSE` looks like this:

```
NODE STATUS RESPONSE
  {
  HEADER
    {
    NAME_TRN_ID = <Same as request ID.>
    FLAGS
      {
      R      = TRUE (1)
      OPCODE = 0x0 (Query)
      AA     = TRUE (1)
      }
    ANCOUNT = 1
    }
  ANSWER_RECORD
    {
    RR_NAME  = <The queried name, copied from the request>
    RR_TYPE  = NBSTAT (0x0021)
    RR_CLASS = IN (0x0001)
    TTL      = 0 (TTL has no meaning in this context)
    RDLENGTH = <Total length of following fields>
    RDATA
      {
      NUM_NAMES = <Number of NODE_NAME[] entries>
      NODE_NAME[]
        {
        NETBIOS_NAME = <16-octet NetBIOS name, unencoded>
        NAME_FLAGS   = <See discussion below>
        }
      STATISTICS = <See discussion below>
      }
    }
  }
```

This packet will need some tearing apart.

The RDATA.NUM_NAMES field is one octet in length. The RDATA.NODE_NAME array represents the responding node's local name table: the list of names the end node believes it owns. Each entry in the array contains a NETBIOS_NAME field and a NAME_FLAGS field.

The NETBIOS_NAME field is 16 bytes in length. The 16-byte name includes the suffix byte and any required padding, and is *not* encoded. The wildcard name (an asterisk followed by 15 nul bytes) is never included in the name list, which contains only registered names.

The listed NetBIOS names all exist within the same NBT scope. The Scope ID will have been sent as part of the original query, and will be stored as part of the RR_NAME field in the reply. Recall that the empty string, "", is a valid Scope ID.

Along with each NETBIOS_NAME there is a NAME_FLAGS field, which provides name status information. It looks like this:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
G	ONT		DRG	CNF	ACT	PRM	UNUSED								

The above is the same as an NB_FLAGS field with four extra bits defined.

DRG: Deregister

When an end node starts the process of releasing a name, it sets this flag. The name will continue to exist in the node's local name table until the name is released.

CNF: Conflict

We have not fully described the Name Conflict condition yet. To put it simply, if two nodes believe they both own the same name (and at least one node claims that the name is unique) then the two nodes are in conflict. One of them has to lose. The loser sets the CNF bit in its local name table and gives up using the disputed name.

ACT: Active

This bit should always be set in the NODE STATUS RESPONSE packets. If, for some strange reason, the end node stores inactive names in its local name table, these are not reported.

PRM: Permanent

According to the RFCs, every NBT end node should register a permanent name. This flag identifies that name. In practice, however, most implementations do not bother with a permanent name and this flag is not used.

These flag values are displayed by Samba's `nmblookup` program. For example:

shell

```
$ nmblookup -S zathras#20
querying zathras on 192.168.101.255
192.168.101.15 zathras<20>
Looking up status of 192.168.101.15
    ZATHRAS         <00> -          B <CONFLICT> <ACTIVE>
    UBIQX           <00> - <GROUP> B <ACTIVE>
    ZATHRAS         <03> -          B <ACTIVE>
    ZATHRAS         <20> -          B <ACTIVE>
    UBIQX           <1e> - <GROUP> B <ACTIVE>
```

The above shows that all of the names are `ACTIVE`, as they should be. The name `ZATHRAS<00>`, however, has been disabled due to a name conflict. From the column of `B`'s, it is apparent that Zathras is operating in B mode.

Now let's take a look at the `RDATA.STATISTICS` field.

This is where things really fall apart. Microsoft's `STATISTICS` blob is quite different from what is specified in the RFCs, and most likely for good reason. At the time the RFCs were published, Microsoft already had at least one NetBIOS implementation. Over time they built a few others, and they had software written to use those implementations. It probably made more sense to stick with familiar layouts than adopt the new one specified in the RFCs.

Fortunately, the data in the `STATISTICS` record is not particularly interesting, and current systems often fill most of it with zeros anyway. Only the first six bytes are commonly used now. Windows systems will attempt to place an Ethernet MAC address into this space. Samba leaves it zero filled.

Buglet Alert

The NBT Name Service listens on port 137, but queries may originate from any UDP port number. Such is the nature of UDP. Programs like Samba's `nmblookup` *utility will open a high-numbered UDP port (something above 1023) in order to send a query. The reply should be sent back to that same port.*

In early versions of Windows 95, however, the source port in `NODE STATUS REQUEST` *messages was ignored. The* `NODE STATUS RESPONSE` *message was sent to UDP port 137 — the wrong port. As a result, the node that sent the query might never hear the reply.*

Time for another chunk of code. Listing 4.10 sends a `NODE STATUS REQUEST` message and then parses and displays the reply. As usual, it uses and builds upon functions presented in previous listings.

Listing 4.10: Node Status Request

```
void Hex_Print( uchar *src, int len )
  /* ------------------------------------------------------ **
   * Print len bytes of src.  Escape any non-printing
   * characters.
   * ------------------------------------------------------ **
   */
  {
  int i;

  for( i = 0; i < len; i++ )
    {
    if( isprint( src[i] ) )
      putchar( src[i] );
    else
      printf( "\\x%.2x", src[i] );
    }
  } /* Hex_Print */

void SendMsg( int             sock,
              uchar          *msg,
              int             msglen,
              struct in_addr address )
  /* ------------------------------------------------------ **
   * Send a message to port UDP/137 at the
   * specified IP address.
   * ------------------------------------------------------ **
   */
```

```
  {
  int                result;
  struct sockaddr_in to;

  to.sin_addr   = address;
  to.sin_family = AF_INET;
  to.sin_port   = htons( 137 );
  result = sendto( sock, (void *)msg, msglen, 0,
                   (struct sockaddr *)&to,
                   sizeof(struct sockaddr_in) );
  if( result < 0 )
    {
    perror( "sendto()" );
    exit( EXIT_FAILURE );
    }
  } /* SendMsg */

void ReadStatusReply( int sock )
  /* ------------------------------------------------------ **
   * Read the Node Status Response message, parse the
   * NODE_NAME[] entries, and print everything in a
   * readable format.
   * ------------------------------------------------------ **
   */
  {
  uchar  bufr[1024];
  ushort flags;
  int    msglen;
  int    offset;
  int    num_names;
  int    i;

  /* Read the message. */
  msglen = recv( sock, bufr, 1024, 0 );
  if( msglen < 0 )
    {
    perror( "recv()" );
    exit( EXIT_FAILURE );
    }

  /* Find start of RDATA (two bytes beyond RDLENGTH). */
  offset = 2 + Find_RDLength( bufr );

  /* The NUM_NAMES field is one byte long. */
  num_names = bufr[offset++];

  /* Now go through and print each name entry. */
```

```
  for( i = 0; i < num_names; i++, offset += 18 )
    {
    flags = (bufr[offset+16] << 8) | bufr[offset+17];

    printf( "NODE_NAME[%d]: ", i );
    Hex_Print( &bufr[offset], 15 );
    printf( "<%.2x>\t", bufr[offset+15] );

    /* Group or Unique. */
    printf( "[%c", ( GROUP_BIT & flags ) ? 'G' : 'U' );

    /* The owner node type. */
    switch( ONT_MASK & flags )
      {
      case ONT_B: printf( ",B" ); break;
      case ONT_P: printf( ",P" ); break;
      case ONT_M: printf( ",M" ); break;
      case ONT_H: printf( ",H" ); break;
      }

    /* Additional flags */
    if( DRG & flags )
      printf( ",DRG" );
    if( CNF & flags )
      printf( ",CNF" );
    if( ACT & flags )
      printf( ",ACT" );
    if( PRM & flags )
      printf( ",PRM" );

    printf( "]\n" );
    }

  /* Windows systems will also send the MAC address. */
  printf( "MAC: %.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
          bufr[offset], bufr[offset+1], bufr[offset+2],
          bufr[offset+3], bufr[offset+4], bufr[offset+5] );
  } /* ReadStatusReply */

int main( int argc, char *argv[] )
  /* ------------------------------------------------- **
   * NBT Node Status Request.
   * ------------------------------------------------- **
   */
```

```
{
int             i;
int             result;
int             ns_sock;
int             msg_len;
uchar           bufr[512];
struct in_addr address;

if( argc != 2 )
  {
  printf( "Usage:  %s <IP>\n", argv[0] );
  exit( EXIT_FAILURE );
  }

if( 0 == inet_aton( argv[1], &address ) )
  {
  printf( "Invalid IP.\n" );
  printf( "Usage:  %s <IP>\n", argv[0] );
  exit( EXIT_FAILURE );
  }

ns_sock = OpenSocket();

msg_len = BuildQuery( bufr,  /* Target buffer.   */
                      0,     /* Broadcast false. */
                      0,     /* RD bit false.    */
                      "*",   /* NetBIOS name.    */
                      '\0',  /* Padding (space). */
                      '\0',  /* Suffix (0x00).   */
                      "",    /* Scope ("").      */
                      QTYPE_NBSTAT );

for( i = 0; i < 3; i++ )
  {
  printf( "Sending NODE STATUS query to %s...\n", argv[1] );
  SendMsg( ns_sock, bufr, msg_len, address );
  result = AwaitResponse( ns_sock, 750 );
  if( result )
    {
    ReadStatusReply( ns_sock );
    exit( EXIT_SUCCESS );
    }
  }
printf( "No replies received.\n" );
close( ns_sock );
return( EXIT_FAILURE );
} /* main */
```

4.3.6 *Name Conflict Demand*

The name conflict demand is a simple message. It looks exactly like the NEGATIVE NAME REGISTRATION RESPONSE that we covered earlier, except that the RCODE field contains CFT_ERR (0x7).

To review:

```
NAME CONFLICT DEMAND
  {
  HEADER
    {
    NAME_TRN_ID = <Whatever you like>
    FLAGS
      {
      R      = TRUE (1)
      OPCODE = 0x5 (Registration)
      AA     = TRUE (1)
      RD     = TRUE (1)
      RA     = TRUE (1)
      RCODE  = CFT_ERR (0x7)
      B      = FALSE (0)
      }
    ANCOUNT = 1
    }
  ANSWER_RECORD
    {
    RR_NAME  = <An NBT name owned by the target node>
    RR_TYPE  = NB (0x0020)
    RR_CLASS = IN (0x0001)
    TTL      = 0
    RDLENGTH = 6
    RDATA
      {
      NB_FLAGS
        {
        G   = <TRUE for a group name, FALSE for a unique name>
        ONT = <Owner type>
        }
      NB_ADDRESS = <Owner's IP address>
      }
    }
  }
```

Once you've got NAME REGISTRATION RESPONSE packets coded up this one will be easy. The question is, what does it do?

The NAME CONFLICT DEMAND is sent whenever the NBNS or an end node discovers a name conflict somewhere on the NBT network. The goal is to make the offending node aware of the fact that it has stolen another node's name. An NBNS might send one of these if it finds an inconsistency in its database, possibly as a result of synchronizing with another NBNS.[11] An end node will send a NAME CONFLICT DEMAND if it gets conflicting replies to a NAME QUERY REQUEST, working under the assumption that the first response is the correct one.

When a node receives a NAME CONFLICT DEMAND it is supposed to disable the offending name. Any existing connections that were made using that name are unaffected, but the node will no longer respond to name queries for the disabled name, nor will it allow the disabled name to be used for new connections. It's as if the name no longer exists.

There is an obvious security problem with this behavior. An evildoer can easily disable a name on, say, a file server or other important node. That alone could cause a Denial of Service condition but the evildoer can go further by registering the same name itself, thus assuming the identity of the disabled node. For this reason, Samba and most (but not all) Windows systems ignore NAME CONFLICT DEMAND messages.

4.3.6.1 Name Release Demand Revisited

There are actually two messages that can be used to force a node to give up a name. In addition to the NAME CONFLICT DEMAND, there is the NAME RELEASE DEMAND. You may recall that a node operating in B (or M or H) mode will broadcast a release announcement when it wants to release one of its own names. The same message can be unicast to another node to force the node to give up a name it holds.

---

11. Microsoft's WINS servers can be configured to replicate with one another, simultaneously distributing the database for greater reliability and increasing the risk of conflicts and other corruption. WINS replication takes place over TCP port 42, should you care to observe. The replication protocol is fairly straightforward and has been untangled. There are plans to add WINS replication support to Samba sometime after version 3.0 is released.

```
NAME RELEASE DEMAND (unicast)
  {
  HEADER
    {
    NAME_TRN_ID = <Set when packet is transmitted>
    FLAGS
      {
      OPCODE = 0x6 (Release)
      B      = FALSE (0)
      }
    QDCOUNT = 1
    ARCOUNT = 1
    }
  QUESTION_RECORD
    {
    QUESTION_NAME  = <Encoded NBT name to be released>
    QUESTION_TYPE  = NB (0x0020)
    QUESTION_CLASS = IN (0x0001)
    }
  ADDITIONAL_RECORD
    {
    RR_NAME  = 0xC00C (Label String Pointer to QUESTION_NAME)
    RR_TYPE  = NB (0x0020)
    RR_CLASS = IN (0x0001)
    TTL      = 0 (zero)
    RDLENGTH = 6
    RDATA
      {
      NB_FLAGS
        {
        G   = <TRUE for a group name, FALSE for a unique name>
        ONT = <Target node's owner type>
        }
      NB_ADDRESS = <Target node's IP address>
      }
    }
  }
```

As with the `NAME CONFLICT DEMAND`, most (but not all) systems ignore this message. Play around... see what you find.

4.4 Enough Already

We could dig deeper. We could provide finer detail. We could, for instance, discuss the design, implementation, care, and feeding of a full-scale NBNS...

but not now. It's getting late and we still have a lot of NBT ground to cover. Go ahead and take a quick break. Hug the spouse, make a fresh pot of tea, visit the facilities, scratch the dog, and then we'll move on to the Datagram Service.

When you get back, we will start by overstating one of the key points of this section: that the purpose of the Name Service is to create a virtual NetBIOS LAN on top of a TCP/IP (inter)network.