# 10

# First Contact: Reaching the Server

Getting there is half the fun.

— Unknown

We are approaching this thing in layers. A little history, a quick introductory tour... and now this. It may seem like a bit of a diversion, but the goal in this section is to figure out how a client finds the server and initiates a connection. No, we're not dealing with SMB protocol yet, but we can't send SMB messages until we can talk to a server.

Think of a telephone call. If you want to call your cousin in New York, the first thing you need to know is the telephone number. You could ask your uncle for the number or look it up in the telephone book, or perhaps you have it written on a scrap of paper somewhere in the kitchen with your favorite tofu recipes. If you dial the wrong number you will annoy some guy in a gas station in Brooklyn. When you dial the correct number, the underlying system will go through a complex process to set up the connection so that you can start talking to your cousin (or, more likely, to the answering machine).

Similarly, if you want to connect to an SMB server you might need to resolve a NetBIOS or DNS name to an IP address. Once you have the address, you can attempt to open a session with the server.

Consider this simple SMB URL:

```
smb://server/
```

From the user's perspective, that should be enough to build an initial connection to an SMB server named "server".

From an implementation point of view, the first thing to do with this example is to parse out the "server" substring. In URI parlance, the field we are looking for is called the "host non-terminal,"[1] and it contains the name or address of the server to which we are trying to connect. Our term for the parsed-out string is "Server Identifier." Once we have extracted it, the next thing we need to know how to do is interpret it so that we can use the information to create the session.

## 10.1 Interpreting the Server Identifier

The SMB URL format supports the use of three different identifier types in the host field. We went over them briefly before. They are the IP address, DNS name, or NetBIOS name of the destination. Our next task is to figure out which is which.

Presentation is everything, and it turns out that the code for interpreting the Server Identifier is verbose and tedious. Most of the busywork for handling NetBIOS names was covered in Part I, and there are plenty of tools for dealing with IP addresses and DNS names, so to save time we will *describe* how to interpret and resolve the address (and let you write the code yourself).[2]

**It could be an IP address.**

Check the syntax of the input to determine whether it is a valid representation of an IP address. Do this test first. It is quick, and does not involve sending any queries out over the network. The `inet_aton()` function, common on Unix-like operating systems, does the job nicely for the four-byte IPv4 addresses used today.

IP version 6 (IPv6) addresses are different. They are longer, harder for a human to read, and potentially more complicated to parse out.

1. The "host" field is not really a field, but the name of a non-terminal in the BNF grammar presented in RFC 2396. That grammar has been amended to support IP version 6 (IPv6) addressing in RFC 2732. The SMB URL format adds support for the use of NetBIOS names and Scope IDs, so it is a further extension of the syntax.

2. Additional source code is available at `http://ubiqx.org/libcifs/`.

Fortunately, when used in URLs they are always contained within square brackets, as in the following example:

```
smb://[fe80::240:f4ff:fe1f:8243]/
```

The square brackets are reserved characters, used specifically for this purpose.[3] They make it easy to identify an IPv6 IP address. Once identified, the IPv6 address can be converted into its internal format by the `inet_pton()` function, which is now supported by many systems.

Note that it is, in theory, possible to register a NetBIOS name that looks exactly like an IP address. What's worse is that it might not be the same as the IP address of the node that registered it. That's nasty. Anyone who would do such a thing should have their keyboard taken away. It is probably not important to handle such situations. Defensive programming practices would suggest being prepared, but in this case the perpetrators deserve the troubles they cause for themselves.

**It could be a NetBIOS Name.**

If the Server Identifier isn't an IP address, it could be a NetBIOS name. To see if this is the case, the first step is to look for a dot ('.'). The SMB URL format does not allow unescaped dots to appear in the NetBIOS name itself, so if there is a dot character in the raw string then consider the rest of the string to be a Scope ID. For example:

```
smb://my%2Enode.scope/
```

is made up of the NetBIOS name "`MY.NODE`" and the Scope ID "`SCOPE`". (The URL escape sequence for encoding a dot is `%2E`.)

Once the string has been parsed into its NetBIOS Name and Scope ID components, the next thing to do is to send an NBT Name Query. Always use a suffix value of `0x20`, which is the prescribed suffix for SMB services. The handling of the query depends, of course, on whether the client is a B, P, M, or H node. For anything other than a B node, the IP address of the NBNS is required. Most client implementations keep such information in some form of configuration file or database.

3. See RFC 2732 for information on the use of IPv6 addresses in URLs.

If a positive response is received, keep track of the NetBIOS name and returned IP address. You will need them in order to connect to the server.

**It could be a DNS name.**

If the Server Identifier is neither an IP address nor a NetBIOS name, try DNS name resolution. The `gethostbyname()` function is commonly used to resolve DNS names to IP addresses, but be warned that this is a blocking function. It may take quite a while for it to do its job, and your program will do nothing in the meantime.[4] That is one reason why it is typically the last thing to try.

That is how to go about determining which *kind* of Server Identifier you've been given. Isn't overloading fun? Now you see why the code for handling all of this is tedious and verbose. It really is not very difficult, though, it's just that it takes a bit of work to get it all coded up.

## 10.2 The Destination Port

Port 139 is for NBT, and port 445 is for raw TCP — good rules of thumb. Recall, though, that the NBT Session Service provides a mechanism for redirection. In addition, some security protocols use high-numbered ports to tunnel SMB connections through firewalls. That means that the use of non-standard ports should be supported on the client side.

The SMB URL allows the specification of a destination port number, like so:

```
smb://server:1928/
```

Once again, that fits into standard URI syntax. If you spend any time using a web browser, the port field should be familiar.

What this all means, however, is that the port number does not always indicate which transport should be used. Rather the opposite; if the port number is *not* specified, the default port depends upon the transport. Knowing

4. Samba's nmbd daemon spawns a separate process to handle DNS queries, just to get around this very problem.

which transport to choose is, once again, something that requires some figuring out.

## 10.3 Transport Discovery

As has been stated previously, we are only considering the NBT and naked TCP transports. Both of these are IP-based and the behavior of SMB over these two is nearly identical, so it does not seem as though separating them would be very important... but this is CIFS we're talking about.

The crux of the problem is whether or not the NBT `SESSION REQUEST` message is required. If the server is expecting correct NBT semantics, then we will need to find a valid NetBIOS name to place into the `CALLED NAME` field. This is a complicated process, involving a lot of trial-and-error. The recipe presented below is only one way to go about it. A good chef knows how to adjust the ingredients and choose seasonings to get the desired result. This is as much an art as it is a science.

### 10.3.1 *Run Naked*

Running naked is probably the easiest transport test to try first. The procedure is tasteful and dignified: simply assume that the server is expecting raw TCP transport. Open a TCP connection to port 445 on the server, but *do not* send an NBT `SESSION REQUEST` — just start sending SMB messages and see if that works. There are four possible results from this test:

1. If nothing is listening on port 445 at the server, the TCP connection will fail. If that happens, the client can fall back to using NBT on port 139.
2. If a non-SMB service is running on the destination port, one end or the other will (hopefully) figure out that the messages being exchanged are incomprehensible, and the connection will be dropped. Again, the fall-back is to try NBT on port 139.
3. The remote end may be expecting NBT transport. This *should* never happen when talking to port 445, but defensive programming practices suggest being prepared. If the server requires NBT transport then it will probably reply to the initial SMB message by sending an NBT `NEGATIVE SESSION RESPONSE`.

4. The connection might, after all, succeed.

All of the above applies if the user did not specify a non-standard port number. If the input looks more like this:

```
smb://server:2891/
```

then the option of falling back to NBT on port 139 is excluded. In addition, there is no way to guess which transport type should be used if a port number other than 139 or 445 is specified. (In theory, it is also possible to run NBT transport on port 445 and naked transport on port 139. If you catch anyone doing such a twisted thing you should probably notify the authorities.)

Fortunately, Windows systems (Windows 95, 98, and 2000 were tested) return an NBT `NEGATIVE SESSION RESPONSE` if they get naked semantics on an NBT service port. This makes sense, because it lets the client know that NBT semantics are required. Samba's `smbd` goes one better and simply ignores the lack of a `SESSION REQUEST` message. Samba's behavior effectively merges the two transport types and makes the distinction between them irrelevant, which simplifies things on the server side and makes life easier for the client.

The transport discovery process is illustrated using the anachronistic flowchart presented in Figure 10.1.

### 10.3.2 *Using the NetBIOS Name*

If running naked didn't work, then you will probably need to try NBT transport. Also, back in Section 10.1 we talked about the different types of Server Identifiers that most implementations support. One of those is the NetBIOS name, and it seems logical to assume that if the Server Identifier is a NetBIOS name then the transport will be NBT.

That's two good reasons to give NBT transport a whirl.

As stated earlier, the critical difference between the raw TCP and NBT transports is that NBT requires the `SESSION REQUEST`/`POSITIVE SESSION RESPONSE` exchange before the SMB messages can start flowing. The `SESSION REQUEST`, in turn, must contain a valid `CALLED NAME`. If the `CALLED NAME` is not correct, then some server implementations will reject the connection. (Windows seems to be quite picky, but Samba ignores the `CALLED NAME` field.)

Finding a valid `CALLED NAME` is easy if the Server Identifier is a NetBIOS name because, well... because there you are. The NetBIOS name *is*

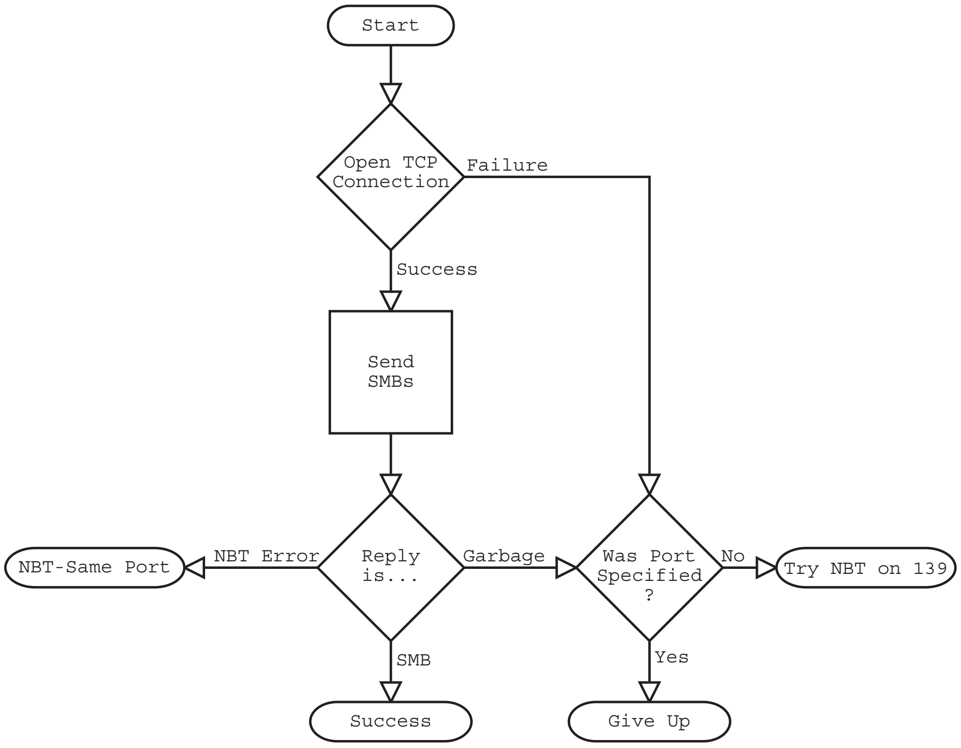

***Figure 10.1:*** *Transport discovery*

the correct CALLED NAME. Also, since the Server Identifier was resolved via an NBT Name Query, the server's IP address is known. That's everything you need.

There is one small problem with this scenario that could cause a little trouble: some NBNS servers can be configured to pass NetBIOS name queries through to the DNS system, which means that the DNS — not the NBNS — may have resolved the name to an IP address. That would mean that we have a false positive and the Server Identifier is not, in fact, a NetBIOS name. If that happens, you could wind up trying to make an NBT connection to a system that isn't running NBT services. (The opposite of the "run naked" test described above.)

Detecting an SMB service that wants naked transport is not as clean and easy as detecting one that wants NBT. In testing, a Windows 2000 system running naked TCP transport did not respond at all to an NBT SESSION REQUEST, and the client timed out waiting for the reply. This problem is neatly avoided if naked transport is attempted before NBT transport. Since Samba considers the `SESSION REQUEST` optional, this kind of transport confusion is not an issue when talking to a Samba server.

### 10.3.3 *Reverse Mapping a NetBIOS Name*

Reverse mapping is the last, desperate means for finding a workable NetBIOS `CALLED NAME` so that a valid `SESSION REQUEST` can be sent. Reverse mapping is also quite common. Your code will need to try this technique if naked transport didn't work and the Server Identifier was a DNS name or IP address — a situation which is not unusual.

As stated before, there is no *right* way to do reverse mapping. Fortunately, there are a few almost-right ways to go about it. Here they are:

**Try a Node Status query.**

Send an NBT `NODE STATUS QUERY` to the server. If it responds, run through the list of returned names looking for a unique name with a suffix byte value of `0x20`. Try using that name as the `CALLED NAME` when setting up the session. If there are multiple names with a suffix value of `0x20`, try them in series until you get a `POSITIVE SESSION RESPONSE` (or until they all fail).

Stop laughing. It gets better.

**Try using the generic `CALLED NAME`.**

This kludge was introduced in Windows NT 4.0 and has been adopted by many other implementations. It is fairly common, but not universal.

The generic `CALLED NAME` is `*SMBSERVER<20>` (that is, "`*SMBSERVER`" with a suffix byte value of `0x20`). Think of it as an alias, allowing you to connect to the SMB server without knowing its "real," registered NetBIOS name. The `*SMBSERVER<20>` name starts with an asterisk, which is against the rules, so it is never registered with the NBT Name Service. If you send a unicast Name Query for this name, the destination node should always send a `NEGATIVE NAME QUERY RESPONSE` in reply (assuming that it is actually running NBT).

A bit awkward but it does work... sometimes. Now for the *coup de gras*.

**Try using the DNS name.**

Try using the first label of the DNS name (the hostname of the server) as the `CALLED NAME`. If you were given an IP address you will need to do a reverse DNS lookup to get a name to play with (we suggested earlier that the DNS name might come in handy). As always, use a suffix byte value of `0x20`.

If the first label doesn't work, try the first two labels (retaining the dot) and so on until you have a string that is longer than 15 bytes, at which point you give up.

Yes, there are implementations which actually do this.

If none of those options worked, then it is finally time to send an error message back to the user explaining that the Server Identifier is no good.

***Ignorance is Bliss Omission Alert***

*We have not fully discussed IPv6.*

*As it currently stands, NBT doesn't work with IPv6. All of the IP address fields in the NBT messages are four-byte fields, but IPv6 addresses are longer. There has been talk of NetBIOS emulation over IPv6, but if such a thing ever happens (unlikely) it will take a while before the proposal is worked out and accepted.*

*Unfortunately, when it comes to SMB over IPv6 the author is clueless. It is probably just like SMB over naked transport, except that the addresses are IPv6 addresses.*

## 10.4 Connecting to the Server

We are still dealing with the transport layer and haven't actually seen any SMBs yet. It is, however, finally time for some code. Listing 10.1 handles the basics of opening the connection with an SMB server. It is example code so, of course, it takes a few shortcuts. For instance, it completely sidesteps Server Identifier interpretation and transport discovery (that is, everything we just covered).

**Listing 10.1:** Opening a session with an SMB server

```
#include <stdio.h>
#include <errno.h>
#include <stdarg.h>
#include <stdlib.h>

#include <sys/poll.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* NBT Session Service Packet Type Codes
 */
#define SESS_MSG           0x00
#define SESS_REQ           0x81
#define SESS_POS_RESP      0x82
#define SESS_NEG_RESP      0x83
#define SESS_RETARGET      0x84
#define SESS_KEEPALIVE     0x85

/* NBT Session Service Error Codes
 */
#define ErrNLCalled        0x80
#define ErrNLCalling       0x81
#define ErrCalledNotPrsnt  0x82
#define ErrInsResources    0x83
#define ErrUnspecified     0x8F

ushort nbt_GetShort( uchar *src, int offset )
  /* ---------------------------------------------------- **
   * Read two bytes from an NBT message and convert them
   * to an unsigned short int.
   * Note that we read the bytes in NBT byte order, which
   * is the opposite of SMB byte order.
   * ---------------------------------------------------- **
   */
  {
  ushort tmp;

  tmp = src[offset];
  tmp = (tmp << 8) | src[offset+1];

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

```
void Fail( char *fmt, ... )
  /* ---------------------------------------------------- **
   * This function formats and prints an error to stdout,
   * then exits the program.
   * A nice quick way to abandon ship.
   * ---------------------------------------------------- **
   */
  {
  va_list ap;

  va_start( ap, fmt );
  (void)fprintf( stdout, "Error: " );
  (void)vfprintf( stdout, fmt, ap );
  exit( EXIT_FAILURE );
  } /* Fail */

void NegResponse( uchar *bufr, int len )
  /* ---------------------------------------------------- **
   * Negative Session Response error reporting.
   *
   * The Negative Session Response message should always
   * be five bytes in length.  The final byte (bufr[4])
   * contains the error code.
   * ---------------------------------------------------- **
   */
  {
  if( len < 5 )
    Fail( "Truncated Negative Session Response.\n" );

  printf( "Negative Session Response: " );

  switch( bufr[4] )
    {
    case ErrNLCalled:
      printf( "Not listening on Called Name.\n" );
      break;
    case ErrNLCalling:
      printf( "Not listening *for* Calling Name.\n" );
      break;
    case ErrCalledNotPrsnt:
      printf( "Called Name not present.\n" );
      break;
    case ErrInsResources:
      printf( "Insufficient resources on server.\n" );
      break;
    case ErrUnspecified:
      printf( "Unspecified error.\n" );
      break;
    default:
      printf( "Unknown error.\n" );
      break;
    }
  } /* NegResponse */

void Retarget( uchar *bufr, int result )
  /* ---------------------------------------------------- **
   * This function is called if we receive a RETARGET
   * SESSION RESPONSE from the server.  The correct thing
   * to do would be to retry the connection, using the
   * returned information.  This function simply reports
   * the retarget response so that the user can manually
   * retry.
   * ---------------------------------------------------- **
   */
  {
  if( result < 10 )
    Fail( "Truncated Retarget Session Response.\n" );

  printf( "Retarget Session Response: " );
  printf( "IP = %d.%d.%d.%d, ",
          bufr[4], bufr[5], bufr[6], bufr[7] );
  printf( "Port = %d\n", nbt_GetShort( bufr, 8 ) );
  } /* Retarget */

int MakeSessReq( uchar *bufr,
                 uchar *Called,
                 uchar *Calling )
  /* ---------------------------------------------------- **
   * Create an NBT SESSION REQUEST message.
   * ---------------------------------------------------- **
   */
  {
  /* Write the header.
   */
  bufr[0] = SESS_REQ;
  bufr[1] = 0;
  bufr[2] = 0;
  bufr[3] = 68;         /* 2x34 bytes in length. */
  /* Copy the Called and Calling names into the buffer.
   */
  (void)memcpy( &bufr[4],  Called,  34 );
  (void)memcpy( &bufr[38], Calling, 34 );

  /* Return the total message length.
   */
  return( 72 );
  } /* MakeSessReq */

int RecvTimeout( int    sock,
                 uchar *bufr,
                 int    bsize,
                 int    timeout )
  /* ---------------------------------------------------- **
   * Attempt to receive a TCP packet within a specified
   * period of time.
   * ---------------------------------------------------- **
   */
  {
  int           result;
  struct pollfd pollfd[1];

  /* Wait timeout/1000 seconds for a message to arrive.
   */
  pollfd->fd      = sock;
  pollfd->events  = POLLIN;
  pollfd->revents = 0;
  result = poll( pollfd, 1, timeout );

  /* A result less than zero is an error.
   */
  if( result < 0 )
    Fail( "Poll() error: %s\n", strerror( errno ) );

  /* A result of zero is a timeout.
   */
  if( result == 0 )
    return( 0 );

  /* A result greater than zero means a message arrived,
   * so we attempt to read the message.
   */
  result = recv( sock, bufr, bsize, 0 );
  if( result < 0 )
    Fail( "Recv() error: %s\n", strerror( errno ) );
  /* Return the number of bytes received.
   * (Zero or more.)
   */
  return( result );
  } /* RecvTimeout */

void RequestNBTSession( int sock,
                        uchar *Called,
                        uchar *Calling )
  /* ---------------------------------------------------- **
   * Send an NBT SESSION REQUEST over the TCP connection,
   * then wait for a reply.
   * ---------------------------------------------------- **
   */
  {
  uchar bufr[128];
  int   result;

  /* Create the NBT Session Request message.
   */
  result = MakeSessReq( bufr, Called, Calling );

  /* Send the NBT Session Request message.
   */
  result = send( sock, bufr, result, 0 );
  if( result < 0 )
    Fail( "Error sending Session Request message: %s\n",
          strerror( errno ) );

  /* Now wait for and handle the reply (2 seconds).
   */
  result = RecvTimeout( sock, bufr, 128, 2000 );
  if( result == 0 )
    {
    printf( "Timeout waiting for NBT Session Response.\n" );
    return;
    }

  switch( *bufr )
    {
    case SESS_POS_RESP:
      /* We got what we wanted. */
      printf( "Positive Session Response.\n" );
      return;
    case SESS_NEG_RESP:
      /* Report an error. */
      NegResponse( bufr, result );
      exit( EXIT_FAILURE );
    case SESS_RETARGET:
      /* We've been retargeted. */
      Retarget( bufr, result );
      exit( EXIT_FAILURE );
    default:
      /* Not a response we expected. */
      Fail( "Unexpected response from server.\n" );
      break;
    }
  } /* RequestNBTSession */

int OpenTCPSession( struct in_addr dst_IP, ushort dst_port )
  /* ---------------------------------------------------- **
   * Open a TCP session with the specified server.
   * Return the connected socket.
   * ---------------------------------------------------- **
   */
  {
  int                sock;
  int                result;
  struct sockaddr_in sock_addr;

  /* Create the socket.
   */
  sock = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP );
  if( sock < 0 )
    Fail( "Failed to create socket(); %s.\n",
          strerror( errno ) );

  /* Connect the socket to the server at the other end.
   */
  sock_addr.sin_addr   = dst_IP;
  sock_addr.sin_family = AF_INET;
  sock_addr.sin_port   = htons( dst_port );
  result = connect( sock,
                    (struct sockaddr *)&sock_addr,
                    sizeof(struct sockaddr_in) );
  if( result < 0 )
    Fail( "Failed to create socket(); %s.\n",
          strerror( errno ) );

  return( sock );
  } /* OpenTCPSession */
```

```
int main( int argc, char *argv[] )
  /* ---------------------------------------------------- **
   * Program mainline.
   * Parse the command-line input and open the connection
   * to the server.
   * ---------------------------------------------------- **
   */
  {
  uchar          Called[34];
  uchar          Calling[34];
  struct in_addr dst_addr;
  int            dst_port = 139;
  int            sock;

  /* Check for the correct number of arguments.
   */
  if( argc < 3 || argc > 4 )
    {
    printf( "Usage:  %s <NAME> <IP> [<PORT>]\n",
            argv[0] );
    exit( EXIT_FAILURE );
    }

  /* Encode the destination name.
   */
  if( '*' == *(argv[1]) )
    (void)L2_Encode( Called, "*SMBSERVER", 0x20, 0x20, "" );
  else
    (void)L2_Encode( Called, argv[1], 0x20, 0x20, "" );

  /* Create a (bogus) Calling Name.
   */
  (void)L2_Encode( Calling, "SMBCLIENT", 0x20, 0x00, "" );

  /* Read the destination IP address.
   * We could do a little more work and resolve
   * the Called Name, but that would add a lot
   * of code to the example.
   */
  if( 0 == inet_aton( argv[2], &dst_addr ) )
    {
    printf( "Invalid IP.\n" );
    printf( "Usage:  %s <NAME> <IP> [<PORT>]\n",
            argv[0] );
    exit( EXIT_FAILURE );
    }
  /* Read the (optional) port number.
   */
  if( argc == 4 )
    {
    dst_port = atoi( argv[3] );
    if( 0 == dst_port )
      {
      printf( "Invalid Port number.\n" );
      printf( "Usage:  %s <NAME> <IP> [<PORT>]\n",
              argv[0] );
      exit( EXIT_FAILURE );
      }
    }

  /* Open the session.
   */
  sock = OpenTCPSession( dst_addr, dst_port );

  /* Comment out the next call for raw TCP.
   */
  RequestNBTSession( sock, Called, Calling );

  /* ** Do real work here. ** */

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

The code in Listing 10.1 provides an outline for setting up the session via NBT or raw TCP. With that step behind us, we won't have to deal with the details of the transport layer any longer. Let's run through some code highlights quickly and put all that transport stuff behind us.

**Transport**

The program does not attempt to discover which transport to use. As written, it assumes NBT transport. To try naked transport, simply comment out the call to `RequestNBTSession()` in `main()`.

**The command line**

Because we are shamelessly avoiding presenting code that interprets Server Identifiers, the example program makes the user do all of the work. The user must enter the NetBIOS name and IP address of the server. Entering a destination port number is optional.

The name entered on the command line will be used as the `CALLED NAME`. If the input string begins with an asterisk, the generic `*SMBSERVER<20>` name will be used instead.

#### The `CALLING NAME` (NBT source address)

The program inserts `SMBCLIENT<00>` as the `CALLING NAME`.

In a correct implementation, the name should be the client's NetBIOS Machine Name (which is typically the same as the client's DNS hostname) with a suffix byte value if `0x00`.

The contents of the `CALLING NAME` field are not particularly significant. According to the expired Leach/Naik CIFS Internet Draft, the same name from the same IP address is supposed to represent the same client... but you knew that. Samba can make use of the `CALLING NAME` via a macro in the `smb.conf` configuration file. The macro is used for all sorts of things, including generating per-client log files.

#### Transporting SMBs

A key feature of this program is the line within `main()` which reads:

```
/* ** Do real work here. ** */
```

That's where the SMB stuff is supposed to happen. At that point in the code, the session has been established on top of the transport layer and it is time to start moving those Server Message Blocks.

Use the program above as a starting point for building your own SMB client utility. Add a parser capable of dissecting the UNC or SMB URL format, and then code up Server Identifier resolution and transport discovery, as described above. When you have all of that put together, you will have completed the foundation of your SMB client.