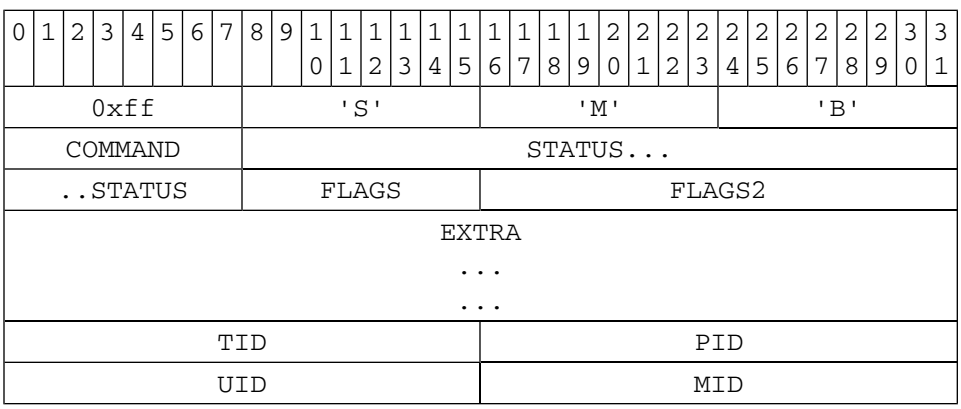
12

The SMB Header in Detail

1st rule of Oriental Cuisine:
Never look inside the eggroll.

During that first expedition into SMB territory we continually deferred, among other things, studying the finer details of the SMB header. We were trying to cover the general concepts, but now we need to dig into the guts of SMB to see how things really work. Latex gloves and lab coats required.

Let's start by revisiting the header layout. Just for review, here's what it looks like:



The first four bytes are constant, so we won't worry about those. The COMMAND field is fairly straightforward too; it's just a one byte field containing an SMB command code. The list of available codes is given in Section 5.1 of the SNIA doc. The rest of the header is where the fun lies...

12.1 The SMB_HEADER.STATUS Field Exposed

Things get interesting starting at the STATUS field. It wouldn't be so bad except for the fact that there are two possible error code formats to consider. There is the DOS and OS/2 format, and then there is the NT_STATUS format. In C language terms, the STATUS field looks something like this:

```
typedef union
  {
  ulong NT_Status;
  struct
    {
    uchar  ErrorClass;
    uchar  reserved;
    ushort ErrorCode;
    } DosError;
  } Status;
```

From the client side, one way to deal with the split personality problem is to use the DOS codes exclusively.[1] These are fairly well documented (by SMB standards), and *should* be supported by all SMB servers. Using DOS codes is probably a good choice, but there is a catch... there are some advanced features which simply don't work unless the client negotiates NT_STATUS codes.

Strange Behavior Alert

If the client negotiates Extended Security with a Windows 2000 server and also negotiates DOS error codes, then the SESSION SETUP ANDX *will fail, and return a DOS hardware error. (!?)*

1. This is exactly what jCIFS does (up through release 0.6.6 and the 0.7.0beta series). There has been a small amount of discussion about supporting the NT_STATUS codes, but it's not clear whether there is any need to change.

```
STATUS
  {
  ErrorClass = 0x03   (Hardware Error)
  ErrorCode  = 0x001F (General Error)
  }
```

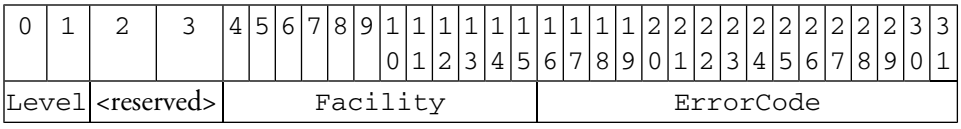
Perhaps W2K doesn't know which DOS error to return, and is guessing. The bigger question is, why does this fail at all?

The same SMB conversation with the NT_STATUS capability enabled works just fine. Perhaps, when the coders were coding that piece of code, they assumed that only clients capable of using NT_STATUS codes would also use the Extended Security feature. Perhaps that assumption came from the knowledge that all ***Windows*** *systems that could handle Extended Security would negotiate NT_STATUS. We can only guess...*

This is one of the oddities of SMB, and another fine bit of forensic SMB research by Andrew Bartlett of the Samba Team.

Another reason to support NT_STATUS codes is that they provide finer-grained diagnostics, simply because there are more of them defined than there are DOS codes. Samba has a fairly complete list of the known NT_STATUS codes, which can be found in the `samba/source/include/nterr.h` file in the Samba distribution. The list of DOS codes is in `doserr.h` in the same directory.

We have already described the structure of the DOS error codes. NT_STATUS codes also have a structure, and it looks like this:



In testing, it appears as though the `Facility` field is always set to zero (`FACILITY_NULL`) for SMB errors. That leaves us with the `Level` and `ErrorCode` fields to provide variety... and, as we have suggested, there is quite a bit of variety. Samba's `nterr.h` file lists over 500 NT_STATUS codes, while `doserr.h` lists only 99 (and some of those are repeats).

`Level` is one of the following:

`00` == Success
`01` == Information
`10` == Warning
`11` == Error

Since the next two bits (the <reserved> bits) are always zero, the highest-order nibble will have one of the following values: `0x0`, `0x4`, `0x8`, or `0xC`. At the other end of the longword, the `ErrorCode` is read as an unsigned short (just like the DOS `ErrorCode` field).

The availability of Samba's list of NT_STATUS codes makes things easy. It took a bit of effort to generate that list, however, as most of the codes are not documented in an accessible form. Andrew Tridgell described the method below, which he used to generate a list of valid NT_STATUS codes. His results were used to create the `nterr.h` file used in Samba.

Tridge's Trick

1. *Modify the source of Samba's* `smbd` *daemon so that whenever you try to delete a file that matches a specific pattern it will return an NT_STATUS error code. (Do this on a testing copy, of course. This hack is not meant for production.) For example, return an error whenever the filename to be deleted matches "*`STATUS_CODE_HACK_FILENAME.*`*". Another thing to do is to include the specific error number as the filename extension, so that the name*

   ```
   STATUS_CODE_HACK_FILENAME.0xC000001D
   ```

   will cause Samba to return an NT_STATUS code of `0xC000001D`.

2. *Create the files on the server side first so you have something to delete. That is easily done with a shell script, such as this:*

   ```
   #!/bin/bash
   #
   i=0;j=256
   while [ $i -lt $j ]
   do
    touch `printf "STATUS_CODE_HACK_FILENAME.0xC000%.4x" $i`
    i=`expr $i + 1`
   done
   ```

   Change the values of `i` *and* `j` *to generate different ranges.*

3. *On a Windows NT or Windows 2000 system, mount the Samba share containing the generated* `STATUS_CODE_HACK*` *files. Next, open a DOS command shell and, one by one, delete the files. For each file, Samba should return the specified NT_STATUS code... and Windows will interpret the code and tell you what it means. If the code is not defined, Windows will tell you that as well.*

4. *If you capture the delete transactions using Microsoft's NetMon tool, it will show you the symbolic names that Microsoft uses for the NT_STATUS codes.*

Okay, now for the next conundrum...

Servers have it tougher than clients. Consider a server that needs to respond to one client using DOS error codes, and to another client using NT_STATUS codes. That's bad enough, but consider what happens when that server needs to query yet another server in order to complete some operation. For example, a file server might need to contact a Domain Controller in order to authenticate the user.

The problem is that, no matter which `STATUS` format the Domain Controller uses when responding to the file server, it will be the wrong format for one of the clients. To solve this problem the server needs to provide a consistent mapping between DOS and NT_STATUS codes.

Windows NT and Windows 2000 both have such mappings built-in but, of course, the details are not published (a partial list is given in Section 6 of the SNIA doc). Andrew Bartlett used a trick similar to Tridge's in order to generate the required mappings. His setup uses a Samba server running as a Primary Domain Controller (PDC), and a Windows 2000 system providing SMB file services. A third system, running Samba's `smbtorture` testing utility, acts as the client. When the client system tries to log on to the Windows server, Windows passes the login request to the Samba PDC.

The test works like this:

Andrew Bartlett's Trick

1. *Modify Samba's authentication code to reject login attempts for any username beginning with "*`0x`*". Translate the login name (e.g. "*`0xC000001D`*") into an NT_STATUS code, and return that in the* `STATUS` *field.*
2. *Configure* `smbtorture` *to negotiate DOS error codes. Aim* `smbtorture` *at the W2K SMB server and try logging in as user* `0xC0000001`*,* `0xC0000002`*... etc.*
3. *For each login attempt from the client, the Windows SMB server will receive a login failure message from the Samba PDC. Since* `smbtorture` *has requested DOS error codes, the W2K pickle-in-the-middle is forced to translate the NT_STATUS values into DOS error codes... and that's how you discover Microsoft's mapping of NT_STATUS codes to DOS error codes.*

   The test configuration is shown in Figure 12.1.

Andrew's test must be rerun periodically. The mappings have been known to change when Windows service packs are installed. See the file

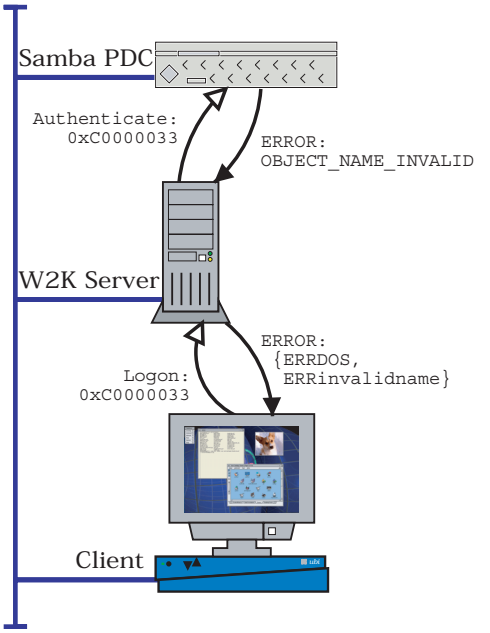


Figure 12.1: *Andrew Bartlett's test configuration*

The polite way to ask Windows for its NT_STATUS-to-DOS error code mappings.

The client sends a logon request to the W2K server, which forwards it to the Samba PDC. The PDC rejects the login, using the Username as the NT_STATUS code. The client requested DOS error codes, so the W2K system must translate.

`samba/source/libsmb/errormap.c` in the Samba distribution for more fun and adventure.[2]

12.2 The `FLAGS` and `FLAGS2` Fields Tell All

Most (but not all) of the bits in the older `FLAGS` field are of interest only to older servers. They represent features that have been superseded by newer features in newer servers. It would be nice if all of the old stuff would just go away

2. After all that work... Sometime around August of 2002, Microsoft posted a bit of documentation listing the DOS error codes that they have defined. Not all are used in CIFS, but it's a nice list to have. In addition, they have documented an NTDLL.DLL function that converts DOS error codes into NT_STATUS codes. (Thanks to Jeremy for finding these.)

so that we wouldn't have to worry about it. It does seem, in fact, as though this is slowly happening. (Maybe it would be better if the old stuff stayed and the new stuff had never happened. Hmmm...)

In any case, this next table presents the FLAGS bits in order of descending significance — the opposite of the order used in the SNIA doc. English speaking people tend to read from left to right and from top to bottom, so it seems logical (as this book is, more or less, written in English)[3] to transpose the left-to-right order into a top-to-bottom table.

SMB_HEADER.FLAGS

Bit	Name / Bitmask / Values	Description
7	SMB_FLAGS_SERVER_TO_REDIR 0x80 0: request 1: reply	What an awful name! On DOS, OS/2, and Windows systems, the client is built into the operating system and is called a "redirector," which is where the "SERVER_TO_REDIR" part of the name comes from. Basically, though, this is simply the reply flag.
6	SMB_FLAGS_REQUEST_BATCH_OPLOCK 0x40 0: Exclusive 1: Batch	**Obsolete.** If bit 5 is set, then bit 6 is the "batch OpLock" (aka OPBATCH) bit. Bit 6 should be clear if bit 5 is clear. In a request from the client, this bit is used to indicate whether the client wants an exclusive OpLock (0) or a batch OpLock (1). In a response, this bit indicates that the server has granted the batch OpLock. OpLocks (opportunistic locks) will be covered later. This bit is only used in the deprecated SMB_COM_OPEN, SMB_COM_CREATE, and SMB_COM_CREATE_NEW SMBs. It should be zero in all other SMBs.

3. The English language is Copyright © 1597 by William Shakespeare & Co., used by permission. All rights deserved.

SMB_HEADER.FLAGS

Bit	Name / Bitmask / Values	Description
		The SMB_COM_OPEN_ANDX SMB has a separate set of flags that handle OpLock requests, as does the SMB_COM_NT_CREATE_ANDX SMB.
5	SMB_FLAGS_REQUEST_OPLOCK 0x20 0: no OpLock 1: OpLock	**Obsolete.** This is the "OpLock" bit. If this bit is set in a request, it indicates that the client wants to obtain an OpLock. If set in the reply, it indicates that the server has granted the OpLock. OpLocks (opportunistic locks) will be covered later. This bit is only used in the deprecated SMB_COM_OPEN, SMB_COM_CREATE, and SMB_COM_CREATE_NEW SMBs. It should be zero in all other SMBs. The SMB_COM_OPEN_ANDX SMB has a separate set of flags that handle OpLock requests, as does the SMB_COM_NT_CREATE_ANDX SMB. (Sigh.)
4	SMB_FLAGS_CANONICAL_PATHNAMES 0x10 0: Host format 1: Canonical	**Obsolete.** This was supposed to be used to indicate whether or not pathnames in SMB messages were mapped to their "canonical" form. Thing is, it doesn't do much good to write a client or server that doesn't map names to the canonical form (which is basically DOS, OS/2, or Windows compatible). This bit should always be set (1).

SMB_HEADER.FLAGS

Bit	Name / Bitmask / Values	Description
3	SMB_FLAGS_CASELESS_PATHNAMES 0x08 0: case-sensitive 1: caseless	When this bit is clear (0), pathnames should be treated as case-sensitive. When the bit is set, pathnames are considered caseless. All good in theory. The trouble is that some systems assume caseless pathnames no matter what the state of this bit. Best practice on the client side is to leave this bit set (1) and always assume caseless pathnames.
2	0x04	**<Reserved> (must be zero).** ...well, sort of. This bit is clearly listed as "Reserved (must be zero)" in both the SNIA and the X/Open docs, yet the latter contains some odd references to optionally using this bit in conjunction with OpLocks. It's probably a typo. Best bet is to clear it (0) and leave it alone.
1	SMB_FLAGS_CLIENT_BUF_AVAIL 0x02 0: Not posted 1: Buffer posted	**Obsolete.** This was probably useful with other transports, such as NetBEUI. If the client sets this bit, it is telling the server that it has already posted a buffer to receive the server's response. The expired Leach/Naik Internet Draft says that this allows a "send without acknowledgment" from the server. This bit should be clear (0) for use with NBT and naked TCP transports.
0	SMB_FLAGS_SUPPORT_LOCKREAD 0x01 0: Not supported 1: Supported	**Obsolete.** If this bit is set in the SMB NEGOTIATE PROTOCOL RESPONSE, then the server supports the deprecated SMB_COM_LOCK_AND_READ and SMB_COM_WRITE_AND_UNLOCK SMBs. Unless you are implementing outdated dialects, this bit should be clear (0).

The NEGOTIATE PROTOCOL REQUEST that we dissected back in Section 11.3 on page 186 shows only the SMB_FLAGS_CANONICAL_PATHNAMES and SMB_FLAGS_CASELESS_PATHNAMES bits set, which is probably the best thing for new implementations to do. Testing with other clients may reveal other workable combinations.

Now let's take a look at the newer flags in the FLAGS2 field.

SMB_HEADER.FLAGS2

Bit	Name / Bitmask / Values	Description
15	SMB_FLAGS2_UNICODE_STRINGS 0x8000 0: ASCII 1: Unicode	If set (1), this bit indicates that string fields within the SMB message are encoded using a two-byte, little endian Unicode format. The SNIA doc says that the format is UTF-16LE but some folks on the Samba Team say it's really UCS-2LE. The latter is probably correct, but it may not matter as both formats are probably the same for the Basic Multilingual Plane. Doesn't Unicode sound like fun?[4] If clear (0), all strings are in 8-bit ASCII format (by which we actually mean 8-bit OEM character set format).
14	SMB_FLAGS2_32BIT_STATUS 0x4000 0: DOS error code 1: NT_STATUS code	Indicates whether the STATUS field is in DOS or NT_STATUS format. This may also be used to help the server guess which format the client prefers before it has actually been negotiated.
13	SMB_FLAGS2_READ_IF_EXECUTE 0x2000 0: Execute != Read 1: Execute confers Read	A quirky little bit this. If set (1), it indicates that execute permission on a file also grants read permission. It is only useful in read operations.

4. One of the reasons that the jCIFS project was started is that Java has built-in Unicode support, which solves a lot of problems. That, plus the native threading model and a few other features, made an SMB implementation in Java very tempting. Support for Unicode in a CIFS implementation is not really optional any more except, perhaps, in the simplest of client systems. Unfortunately, Unicode is way beyond the scope of this book. See the References section for some web links to get you started with Unicode.

SMB_HEADER.FLAGS2

Bit	Name / Bitmask / Values	Description
12	SMB_FLAGS2_DFS_PATHNAME 0x1000 0: Normal pathname 1: DFS pathname	This is used with the **D**istributed **F**ile **S**ystem (DFS), which we haven't covered yet. If this bit is set (1), it indicates that the client knows about DFS, and that the server should resolve any UNC names in the SMB message by looking in the DFS namespace. If this bit is clear (0), the server should not check the DFS namespace.
11	SMB_FLAGS2_EXTENDED_SECURITY 0x0800 0: Normal security 1: Extended security	If set (1), this bit indicates that the sending node understands Extended Security. We'll touch on this again when we discuss authentication.
10	0x0400	**<Reserved> (must be zero)**
9	0x0200	**<Reserved> (must be zero)**
8	0x0100	**<Reserved> (must be zero)**
7	0x0080	**<Reserved> (must be zero)**
6	SMB_FLAGS2_IS_LONG_NAME 0x0040 0: 8.3 format 1: Long names	If set (1), then any pathnames that the SMB contains are long pathnames, else the pathnames are in 8.3 format. Any new CIFS implementation really should support long names.
5	0x0020	**<Reserved> (must be zero)**
4	0x0010	**<Reserved> (must be zero)**
3	0x0008	**<Reserved> (must be zero)**
2	SMB_FLAGS2_SECURITY_SIGNATURE 0x0004 0: No signature 1: Message Authentication Code	If set, the SMB contains a **M**essage **A**uthentication **C**ode (MAC). The MAC is used to authenticate each packet in a session, to prevent various attacks.

SMB_HEADER.FLAGS2

Bit	Name / Bitmask / Values	Description
1	SMB_FLAGS2_EAS 0x0002 0: No EAs 1: Extended Attributes	Indicates that the client understands Extended Attributes. Note that the SNIA doc talks about "Extended Attributes" *and* about "Extended File Attributes." These are two completely different concepts. Extended Attributes are a feature of OS/2. They are mentioned in Section 1.1.6 (page 2) of the SNIA doc and explained in better detail on page 87. Extended File Attributes are described in Section 3.13 (page 30) of the SNIA doc. The SMB_FLAGS2_EAS bit deals with Extended Attribute support.
0	SMB_FLAGS2_KNOWS_LONG_NAMES 0x0001 0: Client wants 8.3 1: Long pathnames okay	Set by the client to let the server know that long names are acceptable in the response.

Some of the flags are used to modify the interpretation of the SMB message, while others are used to negotiate features. Some do both. It may take some experimentation to find the safest way to handle these bits. Implementations are not consistent, so new code must be fine-tuned.

You may need to refer back to these tables as we dig further into the details. Note that the constant names listed above may not match those in the SNIA doc, or those in other docs or available source code. There doesn't seem to be a lot of agreement on the names.

12.3 EXTRA! EXTRA! Read All About It!

Um, actually we are going to delay covering the EXTRA field yet again. EXTRA.PidHigh will be thrown in with the PID field, and EXTRA.Signature will be handled as part of authentication.

12.4 TID and UID: Separated at Birth?

It would seem logical that the [V]UID and TID fields would be somehow related. Both are assigned and managed by the server, and we said before that the SESSION SETUP (where the logon occurs) is supposed to happen *before* the TREE CONNECT.

Well, put all that aside and pay attention to this little story...

Storytime

Once upon a time there were many, many magic kingdoms taking up office space in cities and towns around the world. In each of these magic kingdoms there were lots of overpaid advisors called VeePees. The VeePees were all jealous of one another, but they were more jealous of the underpaid wizards in the IT department who had power over the data and could work spells and make the numbers come out all right.

Then, one day, evil marketing magicians appeared and convinced the VeePees that they could steal all of the power away from the wizards of IT and have it for themselves. To do this, the only thing the VeePees would need was a magic box called a PeeCee (the name appealed to the VeePees). PeeCees, of course, were not cheap but the lure of power was great and the marketing magicians knew that the VeePees had control of the budget.

Soon, the wizards of IT discovered that their supplies of mag-tapes and 8-inch floppies were dwindling, and that no one had bothered to update the service contracts on their VAXes. Worse, the VeePees started taunting them, saying "We don't need you any more. We have spreadsheets." The wise wizards of IT smiled quietly, went back to their darkened cubicles, and entertained themselves by implementing EMACS in TECO macro language. They did not seem at all surprised when the VeePees showed up asking questions like "What happens if I format C-colon?" and "Should I Abort, Retry, or — um — Fail?" The wizards understood what the VeePees did not: With power there must be equal measures of knowledge and understanding, otherwise the power will consume the data — and the user.

The marketing magicians, seeing that their golden goose was molting, came up with a bold plan. They conjured up a LAN system and connected it to a shiny new file-server, which they gave to the IT wizards. At first, the wizards were delighted by the wonderful new server and the beautiful strands of network cable running all over the kingdom. They quickly realized, however, that they had been tricked. The client/server architecture had effectively separated authority from responsibility, and the wizards were left with only the latter.

...and so it is unto this very day. The VeePees and their minions have their PeeCees and hold the power of the data, but they remain under the influence of the sinister marketing magicians. The wizards of IT are still underpaid, have little or no say when decisions are made, and are held responsible and told to clean up the mess whenever anything goes wrong. A wholly dysfunctional arrangement.

So what the purplebananafish does this have to do with TIDs and UIDs? Well, see, it's like this...

Early corporate LANs, such as those in our story, were small and self-contained. The driving goal was to make sure that the data was available to everyone in the office who could legitimately claim to need access. Security was not considered a top priority, so PC OSes (e.g. DOS) did not support complicated minicomputer features like user-based authentication. Given the environment, it is not surprising that the authentication system originally built into SMB was (by today's standards) quite primitive. Passwords, if they were used at all, were assigned to shares — not users — and everyone who wanted to access the same share would use the same password.

This early form of SMB authentication is now known as "Share Level" security. It does not include the concept of user accounts, so the `UID` field is always zero. The password is included in the `TREE CONNECT` message, and a valid `TID` indicates a successfully authenticated connection. In fact, though the `UID` field is listed in the SMB message format layout described in the ancient `COREP.TXT` scrolls, *it is not mentioned again anywhere else in that document*. There is no mention of a `SESSION SETUP` message either.

There are some interesting tricks that add a bit of flexibility to Share Level security. For example, a single share may have multiple passwords assigned, each granting different access rights. It is fairly common, for instance, to assign separate passwords for read-only vs. read/write access to a share.

Another interesting fudge is often used to provide access to user home directories. The server (which, in this case, understands user-based authentication even if the protocol and/or client do not) simply offers usernames as share names. When a user connects to the share matching their username, they give their own login password. The server then checks the username/password pair using its normal account validation routines. Thus, user-based authentication can be mapped to Share Level security (see Figure 12.2).

Figure 12.2: *User-based authentication via Share Level security*

Each share name maps to a username (Chico, Groucho, etc.). The server will accept the user's logon password as the `TREE CONNECT` password.

Share Level security, though still used, is considered deprecated. It has been replaced with "User Level" security which, of course, makes use of username/password instead of sharename/password pairs.

Under User Level security, the SESSION SETUP is performed as the authentication step *before* any TREE CONNECT requests may be sent. If the logon succeeds, the server will assign a valid (non-zero) UID. Subsequent TREE CONNECT attempts can use the UID as an authentication token when requesting access to a share. If User Level security is in use, the password field in the TREE CONNECT message will be blank.

So, with User Level security, the client must authenticate to get a valid UID, and then present the UID to gain access to shares. Thing is, more than one UID may be generated within a single connection, and the UID used to connect to the share does not need to be the same as the one used to access files within the share.

12.5 PID and MID Revealed

Simply put:

- a PID identifies a client process,
- a [PID, MID] pair identifies a thread within a process.

That's the idea, anyway. The client provides values for these fields when it sends a request to the server, and the server is supposed to echo the values back in the response. That way, the client can match the reply to the original request.

Some systems (such as Windows and OS/2) multiplex all of the SMB traffic between a client and a server over a single TCP connection. If the client OS is multi-tasking there may be several active SMB sessions running concurrently, so there may be several requests outstanding at any given time. The SMB conversations are all intertwined, so the client needs a way to sort out the replies and hand them off to the correct thread within the correct process (see Figure 12.3).

The PID field is also used to maintain the semantics of local file I/O. Think about a simple program, like the one in Listing 12.1 which opens a file in read-only mode and dumps the contents. Consider, in particular, the call to the open() function, which returns a file descriptor. File descriptors are

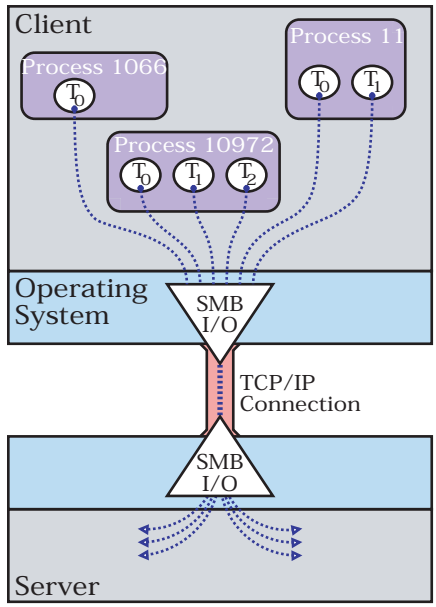


Figure 12.3: *Multiplexing SMB over a single TCP connection*

Instead of opening individual TCP connections, one per process, some systems multiplex all SMB traffic to a given server over a single connection. (T_0, T_1, etc. are threads within a process.)

maintained on a per-process basis — that is, each process has its own private set. The descriptor is an integer used by the operating system to identify an internal record that keeps track of lots of information about the open file, such as:

- Is the file open for reading, writing, or both?
- What is the current file pointer offset within the file?
- Do we have any locks on the file?

Listing 12.1: Quick dump

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#define bSIZE 1024

int main( int argc, char *argv[] )
  /* ------------------------------------------------------ **
   * Copy file contents to stdout.
   * ------------------------------------------------------ **
   */
  {
  int      fd_in;
  int      fd_out;
  ssize_t  count;
  char     bufr[bSIZE];

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

  fd_in = open( argv[1], O_RDONLY );
  if( fd_in < 0 )
    {
    perror( "open()" );
    exit( EXIT_FAILURE );
    }
  fd_out = fileno( stdout );

  do
    {
    count = read( fd_in, bufr, bSIZE );
    if( count > 0 )
      count = write( fd_out, bufr, (size_t)count );
    } while( bSIZE == count );

  if( count < 0 )
    {
    perror( "read()/write()" );
    exit( EXIT_FAILURE );
    }

  (void)close( fd_in );
  exit( EXIT_SUCCESS );
  } /* main */
```

Now take all of that and stretch it out across a network. The files physically reside on the server and information about locks, offsets, etc. must be kept on the server side. The process that has opened the files, however, resides on the client and all of the file status information is relevant within the context of that process. That brings us back to what we said before: The `PID` identifies a client process. It lets the server keep track of client context, and associate it correctly with the right customer when the requests come rolling in.

Further complicating things, some clients support multiple threads running within a process. Threads share context (memory, file descriptors, etc.) with their sister threads within the same process, but each thread may generate SMB traffic all on its own. The `MID` field is used to make sure that server replies get back to the thread that sent the request. The server really doesn't do much with the `MID`. It just echoes it back to the client so, in fact, the client could make whatever use it wanted of the `MID` field. Using it as a thread identifier is probably the most practical thing to do.

There is an important rule which the client should obey with regard to the `MID` and `PID` fields: only one SMB request should ever be outstanding per [`PID`, `MID`] pair per connection. The reason for this rule is that the client will generally need to know the result of a request before sending the next request, especially if an error occurred. The problems which might result should this rule be broken probably depend upon the server, but defensive programming practices would suggest avoiding trouble.

12.5.1 `EXTRA.PidHigh` *Dark Secrets Uncovered*

Earlier on we promised to cover the `EXTRA.PidHigh` field. Well, a promise is a promise...

The `PidHigh` field is supposed to be a `PID` extension, allowing the use of 32-bit rather than 16-bit values as process identifiers. As with all extensions, however, there is the basic problem of backward compatibility.

In this case, trouble shows up if (and only if) the client supports 32-bit process IDs but the server does not. In that situation, the client must have a mechanism for mapping 32-bit process IDs to 16-bit values that can fit into the `PID` field. It doesn't need to be an elaborate mapping scheme, and it is unlikely that there will be 64K client processes talking to the same server at the same time, so it should be a simple problem to solve.

Since that mapping mechanism needs to be in place in order for the client to work with servers that don't support the `PidHigh` field, there's no reason to use 32-bit process IDs at all. In testing, it appears as though the `PidHigh` field is, in fact, always zero (except in some obscure security negotiations that are still not completely understood). Best bet, leave it zero.

12.6 SMB Header Final Report

Code...

The next Listing 12.2 provides support for reading and writing SMB message headers. Most of the header fields are simple integer values, so we can use the `smb_Set*()` and `smb_Get*()` functions from Listing 11.1 to move the data in and out of the header buffer. To make subsequent code easier to read, we provide a set of macros with nice clear names to front-end the function calls and assignments that are actually used.

Listing 12.2a: SMB Header [De]Construction: `MB_Header.h`

```
/* SMB Headers are always 32 bytes long.
 */
#define SMB_HDR_SIZE 32

/* FLAGS field bitmasks.
 */
#define SMB_FLAGS_SERVER_TO_REDIR       0x80
#define SMB_FLAGS_REQUEST_BATCH_OPLOCK  0x40
#define SMB_FLAGS_REQUEST_OPLOCK        0x20
#define SMB_FLAGS_CANONICAL_PATHNAMES   0x10
#define SMB_FLAGS_CASELESS_PATHNAMES    0x08
#define SMB_FLAGS_RESERVED              0x04
#define SMB_FLAGS_CLIENT_BUF_AVAIL      0x02
#define SMB_FLAGS_SUPPORT_LOCKREAD      0x01
#define SMB_FLAGS_MASK                  0xFB

/* FLAGS2 field bitmasks.
 */
#define SMB_FLAGS2_UNICODE_STRINGS      0x8000
#define SMB_FLAGS2_32BIT_STATUS         0x4000
#define SMB_FLAGS2_READ_IF_EXECUTE      0x2000
#define SMB_FLAGS2_DFS_PATHNAME         0x1000
#define SMB_FLAGS2_EXTENDED_SECURITY    0x0800
```

```
#define SMB_FLAGS2_RESERVED_01          0x0400
#define SMB_FLAGS2_RESERVED_02          0x0200
#define SMB_FLAGS2_RESERVED_03          0x0100
#define SMB_FLAGS2_RESERVED_04          0x0080
#define SMB_FLAGS2_IS_LONG_NAME         0x0040
#define SMB_FLAGS2_RESERVED_05          0x0020
#define SMB_FLAGS2_RESERVED_06          0x0010
#define SMB_FLAGS2_RESERVED_07          0x0008
#define SMB_FLAGS2_SECURITY_SIGNATURE   0x0004
#define SMB_FLAGS2_EAS                  0x0002
#define SMB_FLAGS2_KNOWS_LONG_NAMES     0x0001
#define SMB_FLAGS2_MASK                 0xF847

/* Field offsets.
 */
#define SMB_OFFSET_CMD      4
#define SMB_OFFSET_NTSTATUS 5
#define SMB_OFFSET_ECLASS   5
#define SMB_OFFSET_ECODE    7
#define SMB_OFFSET_FLAGS    9
#define SMB_OFFSET_FLAGS2  10
#define SMB_OFFSET_EXTRA   12
#define SMB_OFFSET_TID     24
#define SMB_OFFSET_PID     26
#define SMB_OFFSET_UID     28
#define SMB_OFFSET_MID     30

/* SMB command codes are given in the
 * SNIA doc.
 */

/* Write a command byte to the header buffer.
 */
#define smb_hdrSetCmd( bufr, cmd ) \
        (bufr)[SMB_OFFSET_CMD] = (cmd)

/* Read a command byte; returns uchar.
 */
#define smb_hdrGetCmd( bufr ) \
        (uchar)((bufr)[SMB_OFFSET_CMD])

/* Write a DOS Error Class to the header buffer.
 */
#define smb_hdrSetEclassDOS( bufr, Eclass ) \
        (bufr)[SMB_OFFSET_ECLASS] = (Eclass)
```

```
/* Read a DOS Error Class; returns uchar.
 */
#define smb_hdrGetEclassDOS( bufr ) \
        (uchar)((bufr)[SMB_OFFSET_ECLASS])

/* Write a DOS Error Code to the header buffer.
 */
#define smb_hdrSetEcodeDOS( bufr, Ecode ) \
        smb_SetShort( bufr, SMB_OFFSET_ECODE, Ecode )

/* Read a DOS Error Code; returns ushort.
 */
#define smb_hdrGetEcodeDOS( bufr ) \
        smb_GetShort( bufr, SMB_OFFSET_ECODE )

/* Write an NT_STATUS code.
 */
#define smb_hdrSetNTStatus( bufr, nt_status ) \
     smb_PutLong( bufr, SMB_OFFSET_NTSTATUS, nt_status )

/* Read an NT_STATUS code; returns ulong.
 */
#define smb_hdrGetNTStatus( bufr ) \
        smb_GetLong( bufr, SMB_OFFSET_NTSTATUS )

/* Write FLAGS to the header buffer.
 */
#define smb_hdrSetFlags( bufr, flags ) \
  (bufr)[SMB_OFFSET_FLAGS] = (flags)

/* Read FLAGS; returns uchar.
 */
#define smb_hdrGetFlags( bufr ) \
        (uchar)((bufr)[SMB_OFFSET_FLAGS])

/* Write FLAGS2 to the header buffer.
 */
#define smb_hdrSetFlags2( bufr, flags2 ) \
        smb_SetShort( bufr, SMB_OFFSET_FLAGS2, flags2 )

/* Read FLAGS2; returns ushort.
 */
#define smb_hdrGetFlags2( bufr ) \
        smb_GetShort( bufr, SMB_OFFSET_FLAGS2 )
```

```
/* Write the TID.
 */
#define smb_hdrSetTID( bufr, TID ) \
        smb_SetShort( bufr, SMB_OFFSET_TID, TID )

/* Read the TID; returns ushort.
 */
#define smb_hdrGetTID( bufr ) \
        smb_GetShort( bufr, SMB_OFFSET_TID )

/* Write the PID.
 */
#define smb_hdrSetPID( bufr, PID ) \
        smb_SetShort( bufr, SMB_OFFSET_PID, PID )

/* Read the PID; returns ushort.
 */
#define smb_hdrGetPID( bufr ) \
        smb_GetShort( bufr, SMB_OFFSET_PID )

/* Write the [V]UID.
 */
#define smb_hdrSetUID( bufr, UID ) \
        smb_SetShort( bufr, SMB_OFFSET_UID, UID )

/* Read the [V]UID; returns ushort.
 */
#define smb_hdrGetUID( bufr ) \
        smb_GetShort( bufr, SMB_OFFSET_UID )

/* Write the MID.
 */
#define smb_hdrSetMID( bufr, MID ) \
        smb_SetShort( bufr, SMB_OFFSET_MID, MID )

/* Read the MID; returns ushort.
 */
#define smb_hdrGetMID( bufr ) \
        smb_GetShort( bufr, SMB_OFFSET_MID )

/* Function prototypes.
 */
```

```
int smb_hdrInit( uchar *bufr, int bsize );
  /* ------------------------------------------------- **
   * Initialize an empty header structure.
   * Returns -1 on error, the SMB header size on success.
   * ------------------------------------------------- **
   */

int smb_hdrCheck( uchar *bufr, int bsize );
  /* ------------------------------------------------- **
   * Perform some quick checks on a received buffer to
   * make sure it's safe to read.  This function returns
   * a negative value if the SMB header is invalid.
   * ------------------------------------------------- **
   */
```

Listing 12.2b: SMB Header [De]Construction: `MB_Header.c`

```
#include "smb_header.h"

const char *smb_hdrSMBString = "\xffSMB";

int smb_hdrInit( uchar *bufr, int bsize )
  /* ------------------------------------------------- **
   * Initialize an empty header structure.
   * Returns -1 on error, the SMB header size on success.
   * ------------------------------------------------- **
   */
  {
  int i;

  if( bsize < SMB_HDR_SIZE )
    return( -1 );

  for( i = 0; i < 4; i++ )
    bufr[i] = smb_hdrSMBString[i];
  for( i = 4; i < SMB_HDR_SIZE; i++ )
    bufr[i] = '\0';

  return( SMB_HDR_SIZE );
  } /* smb_hdrInit */
```

```
int smb_hdrCheck( uchar *bufr, int bsize )
  /* ------------------------------------------------- **
   * Perform some quick checks on a received buffer to
   * make sure it's safe to read.  This function returns
   * a negative value if the SMB header is invalid.
   * ------------------------------------------------- **
   */
  {
  int i;

  if( NULL == bufr )
    return( -1 );

  if( bsize < SMB_HDR_SIZE )
    return( -2 );

  for( i = 0; i < 4; i++ )
    if( bufr[i] != smb_hdrSMBString[i] )
      return( -3 );

  return( SMB_HDR_SIZE );
  } /* smb_hdrCheck */
```

The `smb_hdrInit()` and `smb_hdrCheck()` functions are there primarily to ensure that the SMB headers are reasonably sane. They check for things like the buffer size, and ensure that the "`\xffSMB`" string is included correctly in the header buffer.

Note that none of these functions or macros handle the reading and writing of the four-byte session header, though that would be trivial. The `SESSION MESSAGE` header is part of the transport layer, not SMB. It is handled as a simple network-byte-order longword; something from the NBT Session Service that has been carried over into naked transport. (We covered all this back in Chapter 6 on page 129 and Section 8.2 on page 150.)