16

Building Your SMB Vocabulary

> ...they have weapons
> of mass confusion
> and aren't afraid to use them.
>
> — iomud on Slashdot

Looking back over our shoulders, we see that we have performed only two SMB exchanges so far: the `NEGOTIATE PROTOCOL` and the `SESSION SETUP`. There may be a `TREE CONNECT` shoved into the packet with the `SESSION SETUP` as an AndX, but we haven't really described the `TREE CONNECT` in detail.

So, although we have covered a tremendous amount of material, our progress seems rather pathetic doesn't it? What if the rest of SMB is just as tedious, verbose, and difficult?

Relax. It's not.

Certainly there are other difficulties lying in wait, but the biggest ones have already been identified and we are carefully avoiding them. If you pursue your dream of creating a complete and competitive CIFS implementation then you may, some day, need to know how things like MS-RPC and Extended Security really work inside. Fortunately, you can do without them for now.

Let's just be clear on this before we move along:

> There is a lot you can do with CIFS without implementing any of the extended sub-protocols that SMB supports, but if you want to build a complete and competitive CIFS client/server implementation you will need to go well beyond the SMB protocol itself.

That's why it has taken the Samba Team (with help from hundreds if not thousands of people across the Internet) more than ten years to make Samba the industrial-strength server system it is today. Tridge worked out the basics of NBT and SMB in a couple of weeks back in 1991, but new things keep getting tacked on to the system.

When implementing CIFS, the rule of thumb is this: *Implement as little as possible to do the job you need to do.*

The minute you cross the border into uncharted territory, you open up a whole new world to explore and discover. Sometimes, you just don't want to go there. Other times, you must.

Anyway, in the spirit of keeping things simple we will cover only a few more SMB messages, and those in much less depth than we have done so far. There really is no need to study every message, longword, bit, and string. If you've come this far, you should know how to read packet captures and interpret the message definitions in the SNIA doc. It is time to take the training wheels off and learn to *ride*.

16.1 That TREE CONNECT Thingy

We have talked a lot about the TREE CONNECT ANDX REQUEST SMB. There was even an example way back in Section 11.4 on page 188. The example looked like this:

```
SMB_PARAMETERS
  {
  WordCount       = 4
  AndXCommand     = SMB_COM_NONE (0xFF)
  AndXOffset      = 0
  Flags           = 0x0000
  PasswordLength  = 1
  }
SMB_DATA
  {
  ByteCount       = 22
  Password        = ""
  Path            = "\\SMEDLEY\HOME"
  Service         = "?????"  (yes, really)
  }
```

Notice that the TREE CONNECT includes a Password field, but that in this example the Password field is *almost* empty (it contains a nul byte).

If the server negotiates Share Level security, then the password that would otherwise be in the `SESSION_SETUP_ANDX.CaseInsensitive-Password` field will show up in the `TREE_CONNECT_ANDX.Password` field instead. The password may be plaintext, or it may be one of the response values we calculated earlier.

The `TREE_CONNECT_ANDX.Path` field is also worth mentioning. It contains the UNC pathname of the share to which the client is trying to connect. In this example, the client is attempting to access the `HOME` share on node `SMEDLEY`. Note that the `Path` will be in Unicode if negotiated.

Finally there is that weird quintuple question mark string in the `TREE_CONNECT_ANDX.Service` field. There are, as it turns out, five possible values for that field:

String	Meaning
`A:`	A filesystem share
`LPT1:`	A shared printer
`IPC`	An interprocess communications named pipe
`COMM`	A serial or other communications device
`?????`	Wildcard

It's annoying for the client to need to know the kind of share to which it is connecting, which is probably why the wildcard option is available. The server will return the service type in the `Service` field of the Response. Note that the `Service` strings are always in 8-bit ASCII characters — never Unicode.

The response (for LANMAN2.1 and above) looks like this:

```
SMB_PARAMETERS
  {
  WordCount        = 3
  AndXCommand      = <Next ANDX command>
  AndXOffset       = <Next ANDX block offset>
  OptionalSupport  = <A bitfield>
  }
```

```
SMB_DATA
  {
  ByteCount        = <variable>
  Service          = <"A:" | "LPT1:" | "IPC" | "COMM">
  NativeFileSystem = <"" | "FAT" | "NTFS">
  }
```

The example above shows the empty string, "FAT", or "NTFS" as the valid values for the NativeFileSystem field. Other values are possible. (Samba, for instance, has a configuration option that allows you to put in anything you like.) The empty string is used for the hidden IPC$ share.

There are two bits defined in the OptionalSupport bitfield:

Bit	Meaning
SMB_SUPPORT_SEARCH_BITS 0x0001	The meaning of this bit is explained in the LANMAN2.1 documentation. Basically, it indicates that the server knows how to perform directory searches that filter out some entries based on specific file attributes — for example, the DOS archive bit, the directory attribute, etc. This is old stuff and all current implementations should support it.
SMB_SHARE_IS_IN_DFS 0x0002	This bit, if set, indicates that the UNC name is in the **D**istributed **F**ile **S**ystem (DFS) namespace. DFS is yet to be covered.

There is a note in the SNIA doc that states that some servers will leave out the OptionalSupport field even if the LANMAN2.1 or later dialect is negotiated. It does not say whether SMB_SUPPORT_SEARCH_BITS should be assumed in such cases.

16.2 SMB Echo

Here's a toy we can play with.

ECHO is really as simple as it sounds. It's sort of the SMB equivalent of ping. The client sends a packet with a data block full of bytes, and the server echoes the block back. Simple.

...but this is CIFS we're talking about.

Although the ECHO itself is simple, there are many quirks to be found in existing implementations. We will dig into this just a tiny bit to give you a taste of the kinds of problems you are likely to encounter. Let's start with a quick look at the ECHO REQUEST structure:

```
SMB_PARAMETERS
  {
  WordCount  = 1
  EchoCount  = <In theory, anything from 0 to 65535>
  }
SMB_DATA
  {
  ByteCount  = <Number of data bytes to follow>
  Bytes      = <Your favorite soup recipe?>
  }
```

The EchoCount field is a multiplier. It tells the server to respond EchoCount times. If EchoCount is zero, you shouldn't get any reply at all. If EchoCount is 9,999, then you are likely to get nine thousand, nine hundred, and ninety-nine replies. We say *likely* because of the wide variety of weirdity that can be seen in testing.

One bit of weirdation is that all of the systems that were tested would respond to an ECHO REQUEST even if no SESSION SETUP had been sent and no authentication performed. This behavior is, in fact, per design, but it means that any client that can talk to your server from anywhere can ask for EchoCount replies to a single request. (It would probably be safer for the server to send a ERRSRV/ERRnosupport error message in response to an un-authenticated ECHO REQUEST.)

Other strangisms of note:

- In testing, **Windows 9x** systems returned an "Invalid TID" error unless the TID was set to 0xFFFF. Also, these systems sent back at most a single reply, handling EchoCount as if it were a boolean.
- **Windows NT 4.0** and **Windows 2000** would try to send as many replies as specified in EchoCount. If the data block (SMB_DATA.Bytes) was very large (4K was tested) and the EchoCount very high (e.g., 20,000), the server would eventually give up and reset the connection.
- **Samba** has an upper limit of 100 repetitions. Also, Samba sends the replies fast enough that multiple replies will be batched together in a single TCP packet. (That's normal behavior for a TCP stream.)

- The **Windows NT 4.0 (Service Pack 6)** system used in testing failed to respond if the payload was greater than 4323 bytes. **Windows 2000** seems to have an upper limit of 16611 bytes, above which it resets the TCP connection.

Email

```
   From: Conrad Minshall, Apple Computer
     To: Chris Hertel
     Cc: Samba Technical Mailing List
Subject: Re: Bizarre limit alert.

I saw the same "packet drop" with an overlong WRITE_ANDX.  The
maximum buffer size an NT SP6 claims on the NEGOTIATE response
is 0x1104 (4356). This limit is not on the data, the limit
includes the SMB header (32 bytes) and the SMB command. Based
upon the size of an ECHO command I'd expect you could send
4319 bytes, not 4323, so on this topic you'll have to have the
last word... sorry.
```

No apologies. This is CIFS we're talking about.

The ECHO SMB may be one of those things that get coded up just because they're in the documentation and they seem easy. It also appears as though ECHO hasn't been tested much. Certainly, the more it is stressed, the more variation can be seen. There is, however, something to note in the last example in the above list and in the message from Conrad: Once you know what you're looking at, you will find common themes that appear and reappear across a given implementation. These common themes are derived from common internals, and they can provide many clues about the inner workings of the implementation.

Another fine point highlighted by our quick look at the ECHO SMB is that TCP is designed to carry *streams* of data — not discrete packets. This can be seen in the results of the tests against Samba, in which multiple replies were contained in a single TCP packet. At the other extreme, several TCP packets are needed to transfer a single ECHO if it has a very large data payload. As a result, a single read operation may or may not return one and only one complete SMB message.

Oversimplification Alert

The `RecvTimeout()` *function (provided way back in Listing 10.1) makes the assumption that one complete SMB message will be returned per call to the* `recv()` *function. That's a weak assumption. It works well enough for the simple testing we have done so far, but it is not sufficient for a real SMB implementation.*

A better version of `RecvTimeout()` *would verify the received data length against the NBT* `SESSION_MESSAGE.LENGTH` *field value to ensure that only one message is read at a time, and that the complete message is read before it is returned.*

16.3 Readin', Writin', and 'Rithmatic

Here is a quick run-down on some of the basic essentials of SMB.

`OPEN_ANDX`

This SMB is discussed in examples given throughout the SNIA doc, but there is no actual writeup given there. That's because it was labeled as "obsolescent" in the Leach/Naik CIFS draft. The `NT_CREATE_ANDX` SMB is now considered the more fashionable choice. Servers must still support the `OPEN_ANDX` SMB, however, and there are certainly clients that still send it (even under the NT LM 0.12 dialect).

It's times like these that the earlier documentation comes in really handy.

The `OPEN_ANDX` SMB is used to gain access to a file for further processing (reading, writing, that sort of thing). The open file is identified by a `FID` (**F**ile **ID**). The `FID`, of course, is returned by a successful `OPEN_ANDX` call.

`READ_ANDX`

It seems fairly obvious. This one lets you read blocks of data from a file (or device) on the server. The `READ_ANDX` request supports 64-bit file offsets if the `OffsetHigh` field is present (if it is present, the `WordCount` will be 12).

An oddity of the `READ_ANDX` is the `MaxCountHigh` field, which is only used if the `CAP_LARGE_READX` capability has been set. `MaxCountHigh` is an unsigned long (four bytes) that is supposed to

hold the upper 16 bits (two bytes) of the unsigned short (two byte) `MaxCount` field. Two problems with this:

1. Why use a 32-bit field to hold 16 bits worth of data?
2. Even with `CAP_LARGE_READX` set, the maximum SMB large read is 64K. That should fit into the `MaxCount` field with no need for `MaxCountHigh`.

   Play with it and see what happens. Should be interesting.

`WRITE_ANDX`

Allows writing to a file or device. This SMB can also be extended by two words to include an `OffsetHigh` field, thus providing 64-bit offsets. There is also a `DataLengthHigh` field that is comparable to the `MaxCountHigh` from the `READ_ANDX`. In this case, though, the `DataLengthHigh` field is given as an unsigned short. That's only two bytes, which makes more sense.

`SEEK_ANDX`

This one may be considered deprecated. Newer clients probably don't need to send the `SEEK_ANDX`, but servers may need to support it just in case.

Email

```
From: Charles Caldarale
  To: jCIFS Mailing List

SMB_COM_SEEK is a useless SMB, since all of the read and write
functions require a file relative address. It's not surprising
it wasn't used; it would have been a waste of network bandwidth
if it had been sent.

- Chuck
```

See also the SNIA doc's comments regarding this SMB.

`FLUSH`

The `SMB_COM_FLUSH` has nothing to do with plumbing. It is sent by the client to ask the server to write all data and metadata for an open file

(specified by its FID) to disk. If a FID value of 0xFFFF is given, the server is being asked to flush all open files relative to the TID.

NT_CREATE_ANDX

This SMB is used to open, create, or overwrite a file or directory. It offers a myriad of options for file attributes, file sharing, security, etc. As the "NT" in the name implies, the NT_CREATE_ANDX SMB is closely tied to the feature set offered by Windows NT filesystem calls. Here's where you start needing to know more about Windows itself.

One problem with complex calls such as this is that the number of permutations gets to be very high, and it quickly becomes very difficult to test them all.[1] There are various reports describing combinations of values that can cause a Windows NT client or server to go BSOD (**B**lue **S**creen **O**f **D**eath). Have fun with your testing.

There is yet another version of this SMB known as the NT_TRANSACT_CREATE, which is implemented as a sub-command of the SMB_COM_NT_TRANSACTION SMB. It is used to apply Extended Attributes (EAs) or Security Descriptors (SDs) to a file or directory.

CLOSE

All good things must come to an end. Close the file, say goodnight, sing one more song, and get some rest.

Remember earlier when we talked about SMB messages as if we were dissecting some strange, new species of multi-legged critter? Well, we've moved beyond Entomology, Invertebrate Zoology, Taxonomy, and such. We're now studying really complex stuff like Sociology, Psychology, and Numismatics, and we get to put the little critters into Skinner boxes and see how they react to various stimuli. It's important research, and there are all sorts of interesting things to discover.

Consider, for example, the SMB_COM_COPY command. It's supposed to allow you to copy a file from one location on the server to another location.

---

1. I vaguely remember a presentation given by David Korn, author of the Korn Shell (ksh), regarding AT&T's UWIN project. At the end of the presentation there was some discussion regarding the differences between standard Posix APIs and Win32 APIs. It was pointed out that there were hundreds or possibly thousands of permutations of parameter values that could be passed to the Posix open() function. The permutations for the equivalent Win32 function, it was reported, was on the order of millions. How the heck do you test all those possibilities?

That saves the client from having to read the data over the wire and write it back again. A good idea, eh? Unfortunately, no one seems to be able to get it to work — at least, not against Windows servers. There has been some limited success in the laboratory...

Email

```
   From: Greg McCain
     To: Chris Hertel
Subject: CIFS and SMB_COPY

Chris,

I found that smb_copy will in fact copy a file iff:
 - the src file is in the root of the share
 - you do not specify the full path to the file src and dest files
in the smb_copy command. Instead, just specify the names of the
files (this is out of spec.).

The resulting destination file will be named like the source
file, minus the first character. It will NOT be named as specified
in the dest parameter. Hence "smb_copy wanda -> fred" results in
a second file "anda" in the root of the share.

This works on the .NET server RC1 and Windows 2000 servers
that I've tried. Hope it helps.
```

SMB is an old protocol, and it has gotten sloppy over the years. As you work your way through the SMB messages, implementing first the easy ones and then the more difficult ones, keep this thought in mind: *It's not your fault.*

Say it to yourself now: "It's not my fault."

Very good.

That will prevent you from getting frustrated and doubting your own skill. It's really not your fault.

16.4 Transaction SMBs

We are going to blast through this, so you'd better get your running shoes on.

The purpose of the Transaction SMBs is to carry specialized sub-protocols. Examples include the **R**emote **A**dministration **P**rotocol (RAP) and Microsoft's

implementation of DCE/RPC (MS-RPC). There are other, more esoteric sets of calls as well. We will play with some of them when we get to the Browse Service.

Think of these sub-protocols as sets of function calls that are stretched across the network. As suggested in Figure 16.1, a function call is made on the client side and the parameters and data are packed up and shoved across the network. The call is then completed at the remote end and the results (if any) are packed up and shoved back. In CIFS jargon, that's called a *transaction*.

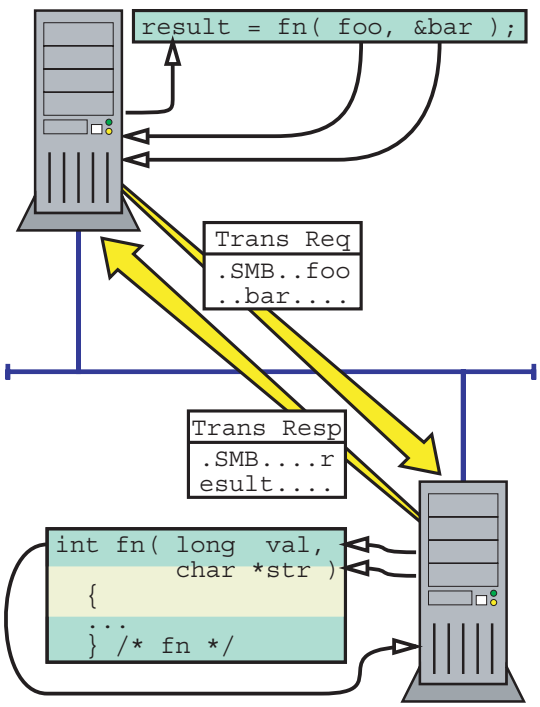

Figure 16.1: *Remote Procedure Call via transaction*

- Software on the client calls the function `fn()`.
- The parameters and pass-by-reference data are packed into an SMB Transaction and sent to the server.
- The server processes the function call.
- If results are expected, the server packs the return value(s) and any pass-by-reference data into a reply transaction.

Transactions are designed to be able to transfer more data than the limit imposed by the negotiated buffer size. They do so by fragmenting the payload. The protocol for sending large **P**rotocol **D**ata **U**nits (PDUs) is described in a variety of documents, but here is a quick run-down:

1. A *primary* Transaction SMB is sent. It includes the total expected size of the transaction (so that the server can prepare to receive the data). It also contains as much of the data as will fit in a single SMB message. If everything fits, skip to step 4.
2. The server sends back an interim response. If the interim response contains an error code then the transaction will be aborted. Otherwise, it is a signal telling the client to continue. The `WORDCOUNT` and `BYTECOUNT` fields are both zero in this message (it's a disembodied header).
3. The client sends as many *secondary* Transaction SMBs as necessary to complete the transaction request.
4. The server executes the called function.
5. The server sends as many response messages as necessary to return the results. In some cases the request does not generate results, and no response is required.

There are three primary Transaction SMBs:

```
SMB_COM_TRANSACTION  == 0x25
SMB_COM_TRANSACTION2 == 0x32
SMB_COM_NT_TRANSACT  == 0xA0
```

Those are really long names, so folks on the various mailing lists tend to shorten them to "SMBtrans," "Trans2," and "NTtrans," respectively. Each of these also has a matching secondary:

```
SMB_COM_TRANSACTION_SECONDARY  == 0x26
SMB_COM_TRANSACTION2_SECONDARY == 0x33
SMB_COM_NT_TRANSACT_SECONDARY  == 0xA1
```

There is very little difference between these three transaction types, except that the NTtrans SMB has 32-bit fields where the other two have 16-bit fields. That means that NTtrans can handle a lot more data (that is, much larger transactions). Besides that, the real difference between these three is the set of functions that are traditionally carried over each.

The SNIA doc and the Leach/Naik CIFS draft provide examples of transactions that use Trans2 and NTtrans. Calls that use SMBtrans are documented elsewhere. Places to look include Luke's book (*DCE/RPC over SMB*), the Leach/Naik Browser and RAP Internet Drafts, and the X/Open documentation (particularly *IPC Mechanisms for SMB*). These (as you already know) are listed in the References section.

16.4.1 *Mailslots and Named Pipes*

Just to simplify things even further, SMBtrans supports yet another layer of abstraction.

Mailslots and Named Pipes are used to access specific sets of remote functions. For example, the "LANMAN" pipe (which is identified as `\PIPE\LANMAN`) is always used for RAP calls.

Named Pipes are two-way inter-process communications channels. Once opened, they can be read from or written to as if they were files. In contrast, Mailslots are used for one-way, connectionless communications.

...and this is where something unexpected happens. Mailslot messages are sent using SMBs transported via the *NBT Datagram Service*. You'll have to see it to believe it, but that is easily arranged. All you need to do is grab a packet capture of port 138 on an active LAN, one with a few local servers that announce themselves to the working Network Neighborhood. If you don't like to wait, reboot something. A Windows 9x system that offers shares will do nicely.

This topic will be revisited in Part III on page 335. If you want to do some extra-curricular reading, the X/Open *IPC Mechanisms for SMB* document is recommended.