22

The Talk on the Street

You are a cornflakes-vert.

— Something my friend Kathy said to me in High School (and I still don't get it)

Enough descriptive hyperbole. Let's get to work.

SMBtrans is an SMB message with the `SMB_COM_TRANSACTION` command byte specified in the header. It is also the transport for all Browse Service messages. The on-the-wire layout of the body of the SMBtrans, in C-style notation, is as follows:

```
typedef struct
  {
  uchar WordCount;            /* SetupCount + 14             */
  struct                      /* SMB-layer parameters        */
    {
    ushort TotalParamCount;     /* Total param bytes to send */
    ushort TotalDataCount;      /* Total data bytes to send  */
    ushort MaxParameterCount;   /* Max param bytes to return */
    ushort MaxDataCount;        /* Max data bytes to return  */
    ushort MaxSetupCount;       /* Max setup words to return */
    ushort Flags;               /* Explained below           */
    ulong  Timeout;             /* Operation timeout         */
    ushort Reserved;            /* Unused word               */
    ushort ParameterCount;      /* Param bytes in this msg   */
    ushort ParameterOffset;     /* Param offset within SMB   */
    ushort DataCount;           /* Data bytes in this msg    */
    ushort DataOffset;          /* Data offset within SMB    */
```

```
  ushort SetupCount;          /* Setup word count          */
  ushort Setup[];             /* Setup words               */
  } Words;
ushort ByteCount;           /* Number of SMB data bytes    */
struct                      /* SMB-layer data              */
  {
  uchar Name[];               /* Transaction service name  */
  uchar Pad[];                /* Pad to word boundary      */
  uchar Parameters[];         /* Parameter bytes           */
  uchar Pad1[];               /* Pad to word boundary      */
  uchar Data[];               /* Data bytes                */
  } Bytes;
} smb_Trans_Req;
```

We can, in fact, make some sense of all that... really we can.

22.1 Making Sense of SMBtrans

As has already been pointed out, we are dealing with layered protocols and layered protocols can cause some terminology confusion. For example, earlier in the book SMB messages were described as having a header, a parameter section, and a data section (and there was a cute picture of an insect to highlight the anatomy). SMB transactions — half a protocol layer up — also have parameters and data. The terms get recycled, as demonstrated by the structure presented above in which the `Parameters[]` and `Data[]` fields are both carried within the `SMB_DATA` block (the `Bytes`) of the SMB message.

SMB transaction messages generally represent some sort of network function call. In an SMB transaction:

- The *Parameters* represent the values passed directly to the function that was called.
- The *Data* represents any indirect values, such as objects indicated by pointers passed as parameters (i.e. objects passed by reference).

That's a very general description, and it may be slightly inaccurate in practice. It works well enough in theory, though, and it provides a conceptual foothold. If you want, you can go one step further and think of the `SetupCount` and `Setup[]` fields as the transaction header.

Okay, now that we have that out of the way, here's what the SMBtrans fields are all about:

SMB Parameters

TotalParamCount

You may recall from earlier discussions that the SMBtrans transaction has the ability to carry a total payload of 64 Kbytes — potentially much more than the SMB buffer size will allow. It does this by sending zero or more secondary transaction messages which contain the additional parameters and/or data.

The `TotalParamCount` field indicates the *total* number of parameter bytes that the server should expect to receive over the course of the transaction. It may take multiple messages to get them all there.

TotalDataCount

Similar to the previous field, this indicates the total number of data bytes the server should expect to receive.

If you think about it, you will see that the theoretical SMBtrans data transfer limit is actually 2 × (216 – 1). Both the `Parameter` and `Data` lengths are 16-bit values so, in theory, SMBtrans can send 128 Kbytes (minus two bytes), which is double the 64K we've been claiming.

MaxParameterCount, MaxDataCount, and MaxSetupCount

These fields let the client inform the server of the maximum number of `Parameter[]`, `Data[]`, and `Setup[]` bytes, respectively, that the client is willing to receive in the server's reply. These are total bytes for the transaction, not per-message bytes.

A note regarding the `MaxSetupCount` field: The X/Open documentation lists this as a 16-bit field, but in the Leach/Naik CIFS draft it is given as a single byte followed by a one-byte nul pad. Because the value is given in SMB byte order (and because it will not exceed 255), either way works.

Flags

There are two flags defined in this field, but they don't appear to be used much in Browser Protocol messages.

SMBtrans flags

Bit	Bitmask	Description
15–2	`0xFFFC`	**<Reserved> (must be zero)**
1	`0x0002`	If set, this is a one-way transaction and the server should not send a response. In theory, this bit should be set in all Class 2 Mailslot messages, but in packet captures this bit always seems to be clear.
0	`0x0001`	If set, it indicates that the client wishes to disconnect from the share indicated by the `TID` field in the SMB header when the transaction completes. Mailslot messages will have a zero `TID` value, so this bit should not be set. RAP calls always use the `IPC$` share, which will have been opened using an earlier `TREE CONNECT` message. So, in theory, this bit *could* be set in RAP calls... but it was clear in all of the packets captured during testing.

`Timeout`

The documentation is scattered. The X/Open docs provide only a few words regarding this particular field. The SNIA doc has a small section (Section 3.2.9) that covers timeouts in general, and some additional information can be found in various places throughout each of those references.

The field indicates the number of milliseconds (1/1000ths of a second) that the *server* should wait for a transaction to complete. A value of zero indicates that the server should return immediately, sending an error code if it could not obtain required resources right away. A value of –1 indicates that the client is willing to have the server wait forever. The documentation doesn't make it clear, but the correct DOS error code in the case of a timeout is probably `ERRSRV/ERRtimeout` (`0x02/0x0058`).

`ParameterCount`

The number of bytes in the `Parameter[]` block of this message. Keep in mind that this may be lower than the `TotalParamCount`

value. If it is, then the rest of the parameters will follow in secondary transaction messages.

ParameterOffset

The offset from the beginning of the SMB message at which the parameter block starts.

DataCount

The number of bytes in the `Data[]` block of this message. Once again, this may be lower than the `TotalDataCount` value. If so, the rest of the data will follow in additional messages.

DataOffset

The offset from the beginning of the SMB message at which the data block starts.

SetupCount

The number of setup *words*. As with the `MaxSetupCount` field, `SetupCount` is presented as an unsigned short in the X/Open document but is given as an unsigned byte followed by a one-byte pad in the Leach/Naik draft.

Setup[]

An array of 16-byte values used to "set up" the transaction (the transaction, not the function call). This might be considered the header portion of the transaction.

SMB Data

Name[]

The name of the Named Pipe or Mailslot to which the transaction is being sent (for example, "`\PIPE\LANMAN`").

Parameters[]

The marshalled parameters.

Data[]

The marshalled data. A little later on, we will carefully avoid explaining how the parameters and data get packaged.

`Pad` and `Pad1`

Some (but not all) clients and servers will add padding bytes (typically, but not necessarily, nul) to force word or longword alignment of the `Parameters[]` and `Data[]` sections. That really messes things up. You must:

- Be sure to use `ByteCount` to figure out how large the `SMB_DATA` section really is.
- Use `ParameterOffset` and `ParameterCount` to figure out where the transaction parameters begin and how many bytes there are.
- Use `DataOffset` and `DataCount` to figure out where the transaction data begins and how many bytes there are.

Gotta love this stuff...

There is a lot more information in both the X/Open documentation and the Leach/Naik CIFS drafts. For some reason, specific details regarding SMBtrans were left out of the SNIA doc, although there is a discussion of Mailslots and Named Pipes (and the other transaction types are covered). All of the listed docs do explain how secondary transaction messages may be used to transfer `Setup[]`, `Parameter[]`, and/or `Data[]` blocks that are larger than the allowed SMB buffer size.

There are also some warnings given in the SNIA doc regarding variations in implementation. It seems you need to be careful with CIFS (no surprise there). See the last paragraph of Section 3.15.3 in the SNIA doc if'n your curious.

...but now it's time for some code.

Listing 22.1 is a bit dense, but it does a decent job of putting together an SMBtrans message from parts. It doesn't fill in the NBT or SMB headers, but there are code examples and descriptions elsewhere in the book that cover those issues. What it does do is provide a starting point for managing SMBtrans transactions, particularly those that might exceed the server's SMB buffer limit and need to be fragmented.

Listing 22.1: SMBtrans messages

```
typedef struct
  {
  ushort  SetupCount;        /* Setup word count          */
  ushort *Setup;             /* Setup words               */
  ushort  Flags;             /* 0x1=Disconnect;0x2=oneway */
  ulong   Timeout;           /* Server timeout in ms      */
  ushort  MaxParameterCount; /* Max param bytes to return */
  ushort  MaxDataCount;      /* Max data bytes to return  */
  ushort  MaxSetupCount;     /* Max setup words to return */
  ushort  TotalParamCount;   /* Total param bytes to send */
  ushort  TotalDataCount;    /* Total data bytes to send  */
  ushort  ParamsSent;        /* Parameters already sent   */
  ushort  DataSent;          /* Data already sent         */
  uchar  *Name;              /* Transaction service name  */
  uchar  *Parameters;        /* Parameter bytes           */
  uchar  *Data;              /* Data bytes                */
  } smb_Transaction_Request;

int SetStr( uchar *dst, int offset, char *src )
  /* ------------------------------------------------- **
   * Quick function to copy a string into a buffer and
   * return the *total* length, including the terminating
   * nul byte.  Does *no* limit checking (bad).
   * Input:  dst    - Destination buffer.
   *         offset - Starting point within destination.
   *         src    - Source string.
   * Output: Number of bytes transferred.
   * ------------------------------------------------- **
   */
  {
  int i;

  for( i = 0; '\0' != src[i]; i++ )
    dst[offset+i] = src[i];
  dst[offset+i] = '\0';

  return( i+1 );
  } /* SetStr */

int smb_TransRequest( uchar                    *bufr,
                      int                       bSize,
                      smb_Transaction_Request *Request )
  /* ------------------------------------------------- **
   * Format an SMBtrans request message.
   * ------------------------------------------------- **
   */
```

```
{
int    offset = 0;
int    keep_offset;
int    bcc_offset;
int    result;
int    i;

/* See that we have enough room for the SMB-level params:
 * Setup + 14 bytes of SMB params + 2 bytes for Bytecount.
 */
if( bSize < (Request->SetupCount + 14 + 2) )
  Fail( "Transaction buffer too small.\n" );

/* Fill the SMB-level parameter block.
 */
bufr[offset++] = (uchar)(Request->SetupCount + 14);
smb_SetShort( bufr, offset, Request->TotalParamCount );
offset += 2;
smb_SetShort( bufr, offset, Request->TotalDataCount );
offset += 2;
smb_SetShort( bufr, offset, Request->MaxParameterCount);
offset += 2;
smb_SetShort( bufr, offset, Request->MaxDataCount );
offset += 2;
smb_SetShort( bufr, offset, Request->MaxSetupCount );
offset += 2;
smb_SetShort( bufr, offset, (Request->Flags & 0x0003) );
offset += 2;
smb_SetLong(  bufr, offset, Request->Timeout );
offset += 4;
smb_SetShort( bufr, offset, 0 );       /* Reserved word */
offset += 2;
keep_offset = offset;   /* Remember ParamCount location */
offset += 8;               /* Skip ahead to SetupCount. */
smb_SetShort( bufr, offset, Request->SetupCount );
offset += 2;
for( i = 0; i < Request->SetupCount; i++ )
  {
  smb_SetShort( bufr, offset, Request->Setup[i] );
  offset += 2;
  }

/* Fill the SMB-level data block...
 * We skip the ByteCount field until the end.
 */
bcc_offset = offset;  /* Keep the Bytecount offset. */
offset += 2;
```

```
/* We need to have enough room to specify the
 * pipe or mailslot.
 */
if( strlen( Request->Name ) >= (bSize - offset) )
  Fail( "No room for Transaction Name: %s\n",
        Request->Name );

/* Start with the pipe or mailslot name.
 */
offset += SetStr( bufr, offset, Request->Name );

/* Now figure out how many SMBtrans parameter bytes
 * we can copy, and copy them.
 */
result = bSize - offset;
if( result > Request->TotalParamCount )
  result = Request->TotalParamCount;
Request->ParamsSent = result;
if( result > 0 )
  (void)memcpy( &bufr[offset],
                Request->Parameters,
                result );
/* Go back and fill in Param Count and Param Offset.
 */
smb_SetShort( bufr, keep_offset, result );
keep_offset += 2;
smb_SetShort( bufr, keep_offset, SMB_HDR_SIZE + offset );
keep_offset += 2;
offset += result;

/* Now figure out how many SMBtrans data bytes we
 * can copy, and copy them.
 */
result = bSize - offset;
if( result > Request->TotalDataCount )
  result = Request->TotalDataCount;
Request->DataSent = result;
if( result > 0 )
  (void)memcpy( &bufr[offset],
                Request->Data,
                result );
/* Go back and fill in Data Count and Data Offset.
 */
smb_SetShort( bufr, keep_offset, result );
keep_offset += 2;
smb_SetShort( bufr, keep_offset, SMB_HDR_SIZE + offset );
```

```
  keep_offset += 2;          /* not really needed any more */
  offset += result;

  /* Go back and fill in the byte count.
   */
  smb_SetShort( bufr, bcc_offset, offset - (bcc_offset+2) );

  /* Done.
   */
  return( offset );
  } /* smb_TransRequest */
```

The `smb_Transaction_Request` structure in the listing differs from the wire-format version. The former is designed to keep track of a transaction while it is being built and until it has been completely transmitted. With a little more code, it should be able to compose secondary transaction messages too. Fortunately, all of the Browse Service requests are small enough to fit into a typical SMB buffer, so you shouldn't have to worry about sending secondary SMB transaction messages. At least not right away. On the other hand, a Browse Server's reply to a `NetServerEnum2` call can easily exceed the SMB buffer size so you may need to know how to rebuild a fragmented response. With that in mind, we will explain how multi-part messages work when we cover `NetServerEnum2`.

It is probably worth noting, at this point, just how many layers of abstraction we're dealing with. If you look at a packet capture of an `NetServerEnum2` request, you'll find that it is way the heck down at the bottom of a large pile:

```
Ethernet II
+ IP
  + TCP
    + NBT Session Service
      + SMB (SMB_COM_TRANSACTION)
        + SMB Pipe Protocol
          + Microsoft Windows Remote Administration Protocol
            + NetServerEnum2
```

It sure is getting deep around here...

All those layers make things seem more complicated than they really are, but if we chip away at it one small workable piece at a time it will all be easier to understand.

22.2 Browse Service Mailslot Messages

The vast bulk of the Browser Protocol consists of Mailslot messages. These are also relatively simple, which is why we are starting with them instead of RAP. Still, there are a lot of layers to go through in order to get a Mailslot message out onto the wire. Let's get chipping...

The NBT layer

Browser Mailslot messages are transported by the NBT Datagram Service, which was covered in Chapter 5 on page 115. We will ignore most of the fields at the NBT layer, since their values are host specific (things like source IP address and **S**ending **N**ode **T**ype). The important fields, from our persective, are:

```
NBT_Datagram
  {
  MsgType         = <unicast, multicast, broadcast, etc.>
  SourceName      = <NBT Source Name>
  DestinationName = <NBT Destination Name>
  UserData        = <The SMBTrans Message>
  }
```

The values assigned to the `SourceName` and `DestinationName` fields may be written as *machine*`<xx>` or *workgroup*`<yy>`, where *machine* and *workgroup* are variable and dependent upon the environment.

The `SourceName` field will contain the name registered by the service sending the request. In practice, this is either the Mailslot Service name (*machine*`<00>`) or the Server Service name (*machine*`<20>`). Both have been seen in testing. In most cases it does not matter.

`UserData` will be indicated indirectly by detailing the SMBtrans and Mailslot layers.

The SMB layer

The NBT Datagram Service is connectionless, and Class 2 Mailslots don't send replies. At the SMB level, however, the header fields are used to maintain state or return some sort of error code or security token. Such values have no meaning in a Mailslot message, so almost all of the SMB header fields are pointless in this context. Only the first five bytes are actually used. That would be the `"\xffSMB"` string and the one byte command code, which is always `SMB_COM_TRANSACTION` (`0x25`).

The rest are all zero. We will not bother to specify the contents of the SMB header in our discussion.

The SMBtrans layer

The SMBtrans transaction fields will be filled in via the `smb_Transaction_Request` structure from Listing 22.1. That way you can map the discussions directly to the code.

For example, if the `Data` block contains ten bytes, the `TotalDataCount` would be filled in like so:

```
smb_Transaction_Request
  {
  TotalDataCount = 10
  }
```

The `SetupCount` and `Setup` fields are constant across all Browser Mailslot messages. The values are specified here so that they don't have to be specified for every message:

```
SetupCount = 3
Setup[]
  {
  0x0001, (Mailslot Opcode   = Mailslot Write)
  0x0001, (Transact Priority = Normal)
  0x0002, (Mailslot Class    = Unreliable/Bcast)
  }
```

Finally, any remaining fields (the values of which have not been otherwise specified or explicitly ignored) should be assumed zero (NULL, nul, nada, non, nerp, nyet, etc.). For example, the `MaxParameterCount`, `MaxDataCount`, and `MaxSetupCount` fields will not be listed because they are always zero in Class 2 Mailslot messages.

The Mailslot Layer

Browser Mailslot messages are carried in the `Data[]` block of the SMBtrans message. They each have their own structure, which will be described using C-style notation.

A few more general notes about Mailslot messages before we forge ahead...

- Browser Mailslots don't use the SMBtrans `Parameters[]` block, so the `TotalParamCount` is always zero.

- The `Mailslot OpCode` in the `Setup[]` field is set to `0x0001`, which indicates a `Mailslot Write` operation. There are no other operations defined for Mailslots, which kinda makes it pointless. This field has nothing to do with the `OpCode` contained within the Mailslot message itself (described below), which identifies the Browse Service function being performed.
- The `Transact Priority` in the `Setup[]` is supposed to contain a value in the range 0..9, where 9 is the highest. The X/Open docs say that if two messages arrive (practically) simultaniously, the higher priority message should be processed first. The SNIA doc says that this field is ignored. The latter is probably correct, but it doesn't matter much. Most of the captures taken in testing show a priority value of 0 or 1.
- The `Mailslot Class`, also in the `Setup[]`, should always contain `0x0002`, indicating a Class 2 Mailslot. The SNIA doc says that this field is ignored too.[1]

Yet one more additional general note regarding Mailslot messages: the first byte of the `Data` block is always an `OpCode` indicating which of the `\MAILSLOT\BROWSE` (or `\MAILSLOT\LANMAN`) functions is being called. Here's a list of the available functions:

OpCode	Function
1	HostAnnouncement
2	AnnouncementRequest
8	RequestElection
9	GetBackupListRequest
10	GetBackupListResponse
11	BecomeBackupRequest
12	DomainAnnouncement
13	MasterAnnouncement
14	ResetBrowserState
15	LocalMasterAnnouncement

1. It is possible that Class 1 Mailslots are not used. At all.

The next step is to describe each of those functions.

Let's get to it...

22.2.1 *Announcement Request*

The `AnnouncementRequest` frame is fairly simple, so it's a good place to start. The message structure (carried in the `smb_Trans_Req.Bytes.Data` section) looks like this:

```
struct
  {
  uchar  OpCode;
  uchar  Unused;
  uchar *ResponseName;
  } AnnouncementRequest;
```

which means that the `AnnouncementRequest` frame is made up of an `OpCode`, an unused byte, and a character string. (The unused byte may have been reserved for use as a flags field at one time.)

The following values are assigned:

```
NBT_Datagram
  {
  MsgType         = 0x11 (DIRECT_GROUP DATAGRAM)
  SourceName      = machine<00>
  DestinationName = workgroup<00>
  }
smb_Transaction_Request
  {
  TotalDataCount  = 3 + strlen( ResponseName )
  Name            = "\MAILSLOT\BROWSE"
  Data
    {
    OpCode        = 0x02 (AnnouncementRequest)
    ResponseName  = <NetBIOS machine name, no suffix>
    }
  }
```

This frame may also be sent to the `MSBROWSE<01>` name to request responses from LMBs for foreign workgroups.

The `TotalDataCount` is calculated by adding:

- one byte for the OpCode,
- one for the Unused byte,
- the string length of the ResponseName field, and
- one byte for the ResponseName nul terminator.

Don't forget those string terminators.

There is no direct reply to this request, so the SourceName and ResponseName fields in the packet are ignored. Providers that receive this message are expected to broadcast a HostAnnouncement frame (described next) to re-announce their services. They are supposed to wait a random amount of time between 0 and 30 seconds before sending the announcement, to avoid network traffic congestion. In testing, however, many systems ignored this message.

Under the older LAN Manager style browsing, a similar message was sent to the \MAILSLOT\LANMAN Mailslot. The LAN Manager AnnounceRequest and Announce frame formats are described in Section 5.3.3 of the X/Open doc *IPC Mechanisms for SMB*.

22.2.2 *Host Announcement*

The HostAnnouncement is a bit more complicated than the AnnouncementRequest. Here's its structure:

```
struct
  {
  uchar  Opcode;
  uchar  UpdateCount;
  ulong  Periodicity;
  uchar *ServerName;
  uchar  OSMajorVers;
  uchar  OSMinorVers;
  ulong  ServerType;
  uchar  BroMajorVers;
  uchar  BroMinorVers;
  ushort Signature;
  uchar *Comment;
  } HostAnnouncement;
```

...and here's how it all pans out:

```
NBT_Datagram
  {
  MsgType          = 0x11 (DIRECT_GROUP DATAGRAM)
  SourceName       = machine<00>
  DestinationName  = workgroup<1D>
  }
smb_Transaction_Request
  {
  TotalDataCount   = 18 + strlen( ServerName + Comment )
  Name             = "\MAILSLOT\BROWSE"
  Data
    {
    OpCode         = 0x01 (HostAnnouncement)
    UpdateCount    = <Incremented after each announcement>
    Periodicity    = <Time until next announcement, in ms>
    ServerName     = <NetBIOS machine name, no suffix>
    OSMajorVers    = 4 <Windows OS version to mimic>
    OSMinorVers    = 5 <Windows OS point version to mimic>
    ServerType     = <Discussion below>
    BroMajorVers   = 15
    BroMinorVers   = 1
    Signature      = 0xaa55
    Comment        = <Server description, max 43 bytes>
    }
  }
```

That needs a once-over.

The announcement is broadcast at the IP level, but the `DestinationName` is the local LMB name so the message should only be picked up by the Local Master. Other nodes could, in theory, listen in and keep their own local Browse List copies up-to-date.

The Leach/Naik Browser Draft says that the `UpdateCount` should be zero and should be ignored by recipients. In practice, it appears that many systems increment that counter for each `HostAnnouncement` frame that they send. No harm done.

The `Periodicity` field announces the amount of time, in milliseconds, that the sender plans to wait until it sends another `HostAnnouncement` frame. As described earlier, the initial period is one minute, but it doubles for each announcement until it would exceed 12 minutes, at which point it is pegged at 12 minutes. In theory, the LMB should remove a host from the Browse List if it has not heard an update from that host after 3 periods have elapsed. In practice, some systems get this value wrong so the LMB should wait 36 minutes.

The `ServerType` field is a complex bitmap. We will dissect it later, as it is also used by the `NetServerEnum2` RAP call.

The Browser version number (15.1) and the `Signature` field are specified in the Leach/Naik Browser draft. Some Windows systems (particularly the Windows 9x family) use a Browser version number of 21.4. No one, it seems, knows why and it doesn't appear that there are any protocol differences between the two versions.

22.2.3 *Election Request*

The `RequestElection` frame is used to start or participate in a Browser Election. It looks like this:

```
struct
  {
  uchar  Opcode;
  uchar  Version;
  ulong  Criteria;
  ulong  UpTime;
  ulong  Reserved;
  uchar *ServerName;
  } RequestElection;
```

In its simplest form, the `RequestElection` can be filled in with zeros. This gives it the lowest possible election criteria. All Potential Browsers in the same workgroup on the same LAN will be able to out-bid the zero-filled request, so a full-scale election will ensue as all Potential Browsers are eligible to participate.

```
NBT_Datagram
  {
  MsgType         = 0x11 (DIRECT_GROUP DATAGRAM)
  SourceName      = machine<00>
  DestinationName = workgroup<1E>
  }
smb_Transaction_Request
  {
  TotalDataCount  = 15
  Name            = "\MAILSLOT\BROWSE"
  Data
    {
    OpCode        = 0x08 (RequestElection)
    }
  }
```

In testing, it was discovered that some Potential Browsers are willing to receive `RequestElection` frames on just about any registered NetBIOS name, including the `MSBROWSE<01>` name.

Once the election gets going, the particpants will all try to out-vote their competition. The details of the election process are convoluted, so they will be set aside for just a little while longer. In the meantime, here is a complete election message, with election criteria filled in.

```
NBT_Datagram
  {
  MsgType         = 0x11 (DIRECT_GROUP DATAGRAM)
  SourceName      = machine<00>
  DestinationName = workgroup<1E>
  }
smb_Transaction_Request
  {
  TotalDataCount  = 15 + strlen( ServerName )
  Name            = "\MAILSLOT\BROWSE"
  Data
    {
    OpCode        = 0x08 (RequestElection)
    Version       = 1
    Criteria      = <Another complex bitmap>
    UpTime        = <Time since last reboot, in milliseconds>
    ServerName    = <NetBIOS machine name, no suffix>
    }
  }
```

The `Criteria` bitmap will be covered along with the election details. Basically, though, it is read as an unsigned long integer and higher values "win."

22.2.4 *Get Backup List Request*

Another simple one. The message looks like this:

```
struct
  {
  uchar OpCode;
  uchar ReqCount;
  ulong Token;
  } GetBackupListRequest;
```

The Ethereal Network Protocol Analyzer and its many authors should be given a good heaping helping of appreciation just about now. The primary reference for the Browse Service data structures is the expired Leach/Naik Browser Internet Draft, but that document was a draft and is now expired. It cannot be expected that it will be completely accurate. It doesn't include the `ReqCount` field in its description, and it lists the `Token` as an unsigned short. That doesn't match what's on the wire. Thankfully, Ethereal knows better.

```
NBT_Datagram
  {
  MsgType         = 0x11 (DIRECT_GROUP DATAGRAM)
  SourceName      = machine<00>
  DestinationName = workgroup<1D>
  }
smb_Transaction_Request
  {
  TotalDataCount  = 6
  Name            = "\MAILSLOT\BROWSE"
  Data
    {
    OpCode        = 0x09 (GetBackupListRequest)
    ReqCount      = <Number of browsers requested>
    Token         = <Whatever>
    }
  }
```

The `ReqCount` field lets the LMB know how large a list of Backup Browsers the client (the Consumer) would like to receive.

The `Token` field is echoed back by the LMB when it sends the `GetBackupListResponse`. Echoing back the `Token` is supposed to let the Consumer match the response to the request. This is necessary because the `SourceName` in the `GetBackupListResponse` is generally the LMB's machine name, so there is nothing in the response that indicates the workgroup. If the Consumer is trying to query multiple workgroups it could easily lose track.

22.2.5 *Get Backup List Response*

This message is sent in response (but not as a reply) to a `GetBackupListRequest`. The structure is fairly straightforward:

```
struct
  {
  uchar  OpCode;
  uchar  BackupCount;
  ulong  Token;
  uchar *BackupList;
  } GetBackupListResponse;

NBT_Datagram
  {
  MsgType         = 0x10 (DIRECT_UNIQUE DATAGRAM)
  SourceName      = machine<00>
  DestinationName = <Source name from the request>
  }
smb_Transaction_Request
  {
  TotalDataCount  = 7 + <length of BackupList>
  Name            = "\MAILSLOT\BROWSE"
  Data
    {
    OpCode        = 0x0A (GetBackupListResponse)
    BackupCount   = <Number of browser names returned>
    Token         = <Echo of the request Token>
    BackupList    = <List of Backup Browsers, nul-delimited>
    }
  }
```

At the IP level this message is unicast, and at the NBT level it is sent as a `DIRECT_UNIQUE DATAGRAM`. This is the closest thing to a Mailslot "reply" that you'll see.

The `Token` is a copy of the `Token` that was sent in the `GetBackup-List Request` that triggered the response. The `BackupCount` value represents the number of names listed in the `BackupList` field, which may be less than the number requested.

The `BackupList` will contain a string of nul-*delimited* substrings. For example, you might get something like this:

```
Data
  {
  OpCode        = 0x0A (GetBackupListResponse)
  BackupCount   = 2
  Token         = 0x61706C65
  BackupList    = "STEFFOND\0CONRAD"
  }
```

which indicates that nodes STEFFOND and CONRAD are both Backup Browsers (and one of them may also be the LMB) for the workgroup. Oh... that string is, of course, nul-terminated as well. Note that you can't use a normal strlen() call to calculate the length of the BackupList. It would just return the length of the first name.

22.2.6 *Local Master Announcement*

The LocalMasterAnnouncement is broadcast by the Local Master Browser. Other nodes, particularly Backup Browsers, can listen for this message and use it to keep track of the whereabouts of the LMB service. If the Local Master Browser for a workgroup hears another node announce itself as the LMB for the same workgroup, then it can call for a new election.

This message is also used to end a Browser Election. The winner declares itself by sending a LocalMasterAnnouncement frame.

The LocalMasterAnnouncement is identical in structure to the HostAnnouncement frame except for its OpCode:

```
smb_Transaction_Request
  {
  Data
    {
    OpCode = 0x0F (LocalMasterAnnouncement)
    }
  }
```

The Leach/Naik draft says that LMBs do not need to send HostAnnouncement frames because the LocalMasterAnnouncement accomplishes the same thing. The real reason that the LMB doesn't need to send HostAnnouncement frames is that HostAnnouncement frames are sent *to the LMB*, and there's no reason for an LMB to announce itself to itself.

22.2.7 *Master Announcement*

The MasterAnnouncement is sent by the LMB to the DMB to let the DMB know that the LMB exists. The message contains the OpCode field and the SMB Server Service name of the LMB. The Server Service name will be registered with the NBNS, so the DMB will be able to look it up as needed.

```
struct
  {
  uchar  OpCode;
  uchar *ServerName;
  } MasterAnnouncement;

NBT_Datagram
  {
  MsgType         = 0x10 (DIRECT_UNIQUE DATAGRAM)
  SourceName      = machine<00>
  DestinationName = workgroup<1B>
  }
smb_Transaction_Request
  {
  TotalDataCount  = 2 + strlen( ServerName )
  Name            = "\MAILSLOT\BROWSE"
  Data
    {
    OpCode        = 0x0D (MasterAnnouncement)
    ServerName    = <NetBIOS machine name, no suffix>
    }
  }
```

When the DMB receives a `MasterAnnouncement`, it should perform a `NetServerEnum2` synchronization with the LMB. It should also keep track of remote LMBs in its workgroup and periodically (every 15 minutes) synchronize Browse Lists with them. Likewise, an LMB will periodically query the DMB. This is how the Browse List is propagated across multiple subnets.

Note that this message is unicast. A broadcast datagram would not reach a remote DMB.

22.2.8 *Domain Announcement*

The `DomainAnnouncement` has the same structure as the `HostAnnouncement` and `LocalMasterAnnouncement` frames. The difference is in the content.

The `DomainAnnouncement` is sent to the `MSBROWSE<01>` name, so that all of the foreign LMBs on the subnet will receive it. Instead of the NetBIOS machine name, the `ServerName` field contains the workgroup name. The NetBIOS machine name is also reported, but it is placed into the `Comment` field.

```
NBT_Datagram
  {
  MsgType         = 0x11 (DIRECT_GROUP DATAGRAM)
  SourceName      = machine<00>
  DestinationName = "\01\02__MSBROWSE__\02<01>"
  }
smb_Transaction_Request
  {
  TotalDataCount  = 18 + strlen( ServerName + Comment )
  Name            = "\MAILSLOT\BROWSE"
  Data
    {
    OpCode        = 0x0C (DomainAnnouncement)
    UpdateCount   = <Incremented after each announcement>
    Periodicity   = <Time until next announcement, in ms>
    ServerName    = <NetBIOS workgroup name, no suffix>
    OSMajorVers   = 4 <Windows OS version to mimic>
    OSMinorVers   = 5 <Windows OS point version to mimic>
    ServerType    = <Discussion below>
    BroMajorVers  = 15
    BroMinorVers  = 1
    Signature     = 0xaa55
    Comment       = <LMB NetBIOS machine name, no suffix>
    }
  }
```

A note of caution on this one. Some Windows systems send what appears to be garblage in the `BroMajorVers`, `BroMinorVers`, and `Signature` fields. Ethereal compensates by combining these three into a single longword which it calls "Mysterious Field."

22.2.9 *Become Backup Request*

This message is sent by the LMB when it wants to promote a Potential Browser to Backup Browser status.

```
struct
  {
  uchar  OpCode;
  uchar *BrowserName;
  } BecomeBackupRequest;
```

```
NBT_Datagram
  {
  MsgType          = 0x11 (DIRECT_GROUP DATAGRAM)
  SourceName       = machine<00>
  DestinationName = workgroup<1E>
  }
smb_Transaction_Request
  {
  TotalDataCount  = 2 + strlen( BrowserName )
  Name             = "\MAILSLOT\BROWSE"
  Data
    {
    OpCode         = 0x0B (BecomeBackupRequest)
    BrowserName    = <NetBIOS machine name of promoted node>
    }
  }
```

The message is an NBT multicast datagram sent to all Potential Browsers in the workgroup. The `BrowserName` field contains the name of the node that is being promoted (no suffix byte). That node will respond by sending a new `HostAnnouncement` frame and obtaining a fresh copy of the Browse List from the LMB. The newly promoted Backup Browser should refresh its Browse List copy every 15 minutes.

22.2.10 *The Undocumented Reset*

It is difficult to find documentation on this message — it's not written up in the Leach/Naik draft — but there is some information hiding around the web if you dig a little... and, of course, we're describing it here.

Big things come in small packages. Here's the `ResetBrowserState` frame:

```
struct
  {
  uchar OpCode;
  uchar Command;
  } ResetBrowserState;
```

Not much to it, but it can have an impact. This is how it's filled in:

```
NBT_Datagram
  {
  MsgType         = 0x11 (DIRECT_GROUP DATAGRAM)
  SourceName      = machine<00>
  DestinationName = workgroup<1D>
  }
smb_Transaction_Request
  {
  TotalDataCount  = 2
  Name            = "\MAILSLOT\BROWSE"
  Data
    {
    OpCode        = 0x0E (ResetBrowserState)
    Command       = <Bitfield - see below>
    }
  }
```

The `ResetBrowserState` message can mess with a Local Master Browser's mind. There are three bits defined for the `Command` field, and here's what they do:

`ResetBrowserState` *command bits*

Bit	Name / Bitmask	Description
7–3	`0xF8`	<Reserved> (must be zero)
2	`RESET_STATE_STOP` `0x04`	Tells the Local Master Browser not to be a browser any more. The LMB will de-register its `<1D>` and `<1E>` names and sulk in a corner. Many implementations ignore this command, even if they respect the others. DMBs should never accept this command.
1	`RESET_STATE_CLEAR_ALL` `0x02`	Causes the LMB to clear its Browse List and start over.
0	`RESET_STATE_STOP_MASTER` `0x01`	Causes the LMB to demote itself to a Backup Browser. This will, eventually, cause a new election (which may be won by the very same system).

22.2.11 *It's All in the Delivery*

Would a little more code be useful?

The code gets rather dull at this level because all we are really doing is packing and unpacking bytes. Unfortunately, that's what network protocols are all about. Not very glamorous, is it?

Listing 22.2 packs a `RequestElection` message into a byte block so that it can be handed to the `smb_TransRequest()` function via the `smb_Transaction_Request` structure. Sending election requests to a busy LAN can be kinda fun... and possibly a little disruptive.

Listing 22.2: SMBtrans messages

```
#define BROWSE_REQUEST_ELECTION 0x08

static smb_Transaction_Request TReqs[1];

static const ushort MailSlotSetup[3]
                    = { 0x0001, 0x0001, 0x0002 };
static const uchar *MailSlotName
                    = "\\MAILSLOT\\BROWSE";

int ElectionRequest( uchar *bufr,
                     int    bSize,
                     ulong  Criteria,
                     ulong  Uptime,
                     uchar *ServerName)
  /* ------------------------------------------------------ **
   * Marshal an Election Request record.
   *
   * Returns the number of bytes used.
   * ------------------------------------------------------ **
   */
  {
  size_t len;
  uchar  buildData[32];

  /* Initialize the TReqs block.
   */
  (void)memset( TReqs, 0, sizeof(smb_Transaction_Request) );
  TReqs->SetupCount = 3;
  TReqs->Setup      = MailSlotSetup;
  TReqs->Name       = MailSlotName;
```

```
/* Build the Browser message in 'buildData'. */
(void)memset( buildData, '\0', 32 );
buildData[0] = BROWSE_REQUEST_ELECTION;
len = 15;

/* If the ServerName is empty, assume that the
 * request is for a zero-filled election message.
 * Otherwise, fill in the rest of the message.
 */
if( NULL != ServerName && '\0' != *ServerName )
  {
  buildData[1] = 1;                          /* Version.  */
  smb_SetLong( buildData, 2, Criteria );     /* Criteria. */
  smb_SetLong( buildData, 6, Uptime );       /* Uptime.   */
                               /* Skip 4 reserved bytes. */

  /* Copy the ServerName, and make sure there's a nul.
   * Count the nul in the total.
   */
  (void)strncpy( &(buildbufr[15]), ServerName, 15 );
  bufr[31] = '\0';
  len += 1 + strlen( &(buildbufr[15]) );
  }

/* Finish filling in the transaction request structure.
 */
TReqs->TotalDataCount = (ushort)len;
TReqs->Data           = buildData;

/* Write the transaction into the buffer.
 * Return the transaction message size.
 */
len = smb_TransRequest( bufr, bSize, TReqs );
return( len );
} /* ElectionRequest */
```

22.3 RAPture

Understand this at the outset: Examining a function of the RAP protocol is like studying the runic carvings on the lid of Pandora's box. They might just be large friendly letters... or they could be the manufacturer's warning label.

We are *not* going to open the box.

The `NetServerEnum2` function can be implemented without having to fully understand the inner workings of RAP, so there really is no need. If you want to, you can rummage around in the RAP functions by reading through Appendix B of the X/Open book *Protocols for X/Open PC Interworking: SMB, Version 2*. After that, there is yet again another additional further Leach/Naik draft already. You can find the Leach/Naik *CIFS Remote Administration Protocol Preliminary Draft* under the filename `cifsrap2.txt` on Microsoft's FTP server. It is definitely a draft, but it provides a lot of good information if you read it carefully. One more resource a die-hard RAP-per will want to check is *Remoted Net API Format Strings*, which is an email message that was sent to Microsoft's CIFS mailing list by Paul Leach. It provides details on the formatting of RAP messages. All of these sources are, of course, listed in the References section.

One of the downsides of RAP, from our perspective, is that it defines *yet another layer* of parameters and data... and there's a heap, too.

Gotta love layers.

RAP provides a formula for marshalling its parameters, data, and heap, passing them over a network connection, and then passing the results back again. A complete RAP implementation would most likely automate the marshalling and unmarshalling process, and the human eye would never need to see it. That would be overkill in our case, so we're stuck doing things the easy way — by hand.

RAP functions are sent via a Named Pipe, not a Mailslot, so the whole communications process is different. Like the Mailslot-based functions, RAP functions are packed into an SMBtrans transaction, but that's just about all that's really the same. The steps which must be followed in order to execute a RAP call are:

- Open a TCP session.
  - NBT Session Request.
    - SMB Negotiate Protocol.
    - SMB Session Setup.
    - SMB Tree Connect (to `\\`**`machine`**`\IPC$`).
      - RAP call and reply.

- SMB Tree Disconnect (optional).
- SMB Logoff (optional).

- Close TCP session.

You can see all of this very clearly in a packet capture. Having a sniff handy as you read through this section is highly recommended, by the way. Don't forget to listen on 139/TCP instead of (or in addition to) 138/UDP.

22.3.1 `NetServerEnum2` *Request*

You can generate a `NetServerEnum2` exchange in a variety of ways. For example, you can refresh the server list in the Windows Network Neighborhood or use the jCIFS `List.java` utility with the URL "`smb://`**`workgroup`**`/`". The request, as displayed by the packet sniffer, should look something like this:

```
+ Transmission Control Protocol
+ NetBIOS Session Service
+ SMB (Server Message Block Protocol)
  SMB Pipe Protocol
- Microsoft Windows Lanman Remote API Protocol
    Function Code: NetServerEnum2 (104)
    Parameter Descriptor: WrLehDz
    Return Descriptor: B16BBDz
    Detail Level: 1
    Receive Buffer Length: 65535
    Server Type: 0xffffffff
    Enumeration Domain: WORKGROUP
```

The `Descriptor` fields are a distinctive feature of RAP requests. These are the cryptic runes of which we spoke earlier. They are format strings, used to define the structure of the parameters and data being sent as well as that expected in the reply. They can be used to automate the packing and unpacking of the packets, or they can be stuffed into the packet as constants with no regard to their meaning. The latter is the simpler course. With that in mind, here is the (simplified, but still correct) C-style format of a `NetServerEnum2` request:

```
struct
  {
  ushort RAPCode;
  uchar *ParamDesc;
  uchar *DataDesc;
  struct
    {
    ushort InfoLevel;
    ushort BufrSize;
    ulong  ServerType;
    uchar *Workgroup;
    } Params;
  } NetServerEnum2Req;
```

So, given the above structure, the `NetServerEnum2` request is filled in as shown below. Note that, at the SMBtrans-level, there are no `Setup[]` words, the `Data[]` section is empty, and all of the above structure is bundled into the `Parameter[]` block.

```
smb_Transaction_Request
  {
  TotalParamCount   = 27 + strlen( Workgroup )
  MaxParameterCount = 8
  MaxDataCount      = <Size of the reply buffer>
  Name              = "\PIPE\LANMAN"
  Data
    {
    RAPCode         = 104 (0x0068)
    ParamDesc       = "WrLehDz"
    DataDesc        = "B16BBDz"
    RAP_Params
      {
      InfoLevel     = 1 <See below>
      BufrSize      = <Same as MaxDataCount>
      ServerType    = <See below>
      Workgroup     = <Name of the workgroup to list>
      }
    }
  }
```

A few of those fields need a little discussion.

`TotalParamCount`

The value 27 includes three short integers, one long integer, two constant strings (with lengths of 8 bytes each), and one nul byte to terminate the `Workgroup` field. That adds up to 27 bytes.

MaxDataCount and BufrSize

Samba allocates the largest size buffer it can (64 Kbytes minus one byte) to receive the response data. Other clients seem to have trouble with a 64K buffer, and will subtract a few bytes from the size. 64K minus 360 bytes has been seen, and jCIFS uses 64K minus 512 bytes.

Email

```
From: Allen, Michael B
  To: jcifs@samba.org

I think I just made it up. I found 0xFFFF would result in
errors. I never really investigated why.
```

InfoLevel

There are two `InfoLevels` available: 0 and 1. Level 0 is not very interesting. Note that if you want to try level 0, you will need to change the `DataDesc` string as well.

ServerType

There are two common values used in the request message. They are:

```
SV_TYPE_DOMAIN_ENUM == 0x80000000
SV_TYPE_ALL         == 0xFFFFFFFF
```

The first is used to query the browser for the list of all known workgroups. The second is used to query for a list of all known Providers in the specified (or default) workgroup.

Note that these are not the only allowed values. When we cover the reply message (next section) there will be a table of all known bit values. Queries for specific subsets of Providers can be generated using these bits.

Workgroup

In many cases, this will be an empty string (just a nul byte). An empty `Workgroup` field represents a request to list the Providers that are members of the browser's default workgroup. That means, of course, that the browser being queried *must have* a default workgroup.

This results in an interesting problem. Since the workgroup name is not always specified, a single system cannot (on a single IP address) be

the LMB for more than one workgroup. If a node were to become the LMB for multiple workgroups, then it would not know which set of servers to report in response to a `NetServerEnum2` query with an empty workgroup name.

...and that is "all you need to know" about the `NetServerEnum2` request message.

22.3.2 `NetServerEnum2` *Reply*

The response message is a bit more involved, so you may want to take notes. A packet capture, once again, is a highly recommended visual aide.

Starting at the top... The `TotalParamCount` field in the SMBtrans reply message will have a value of 8, indicating the size of the SMBtrans-level `Parameter[]` block. Those bytes fall out as follows:

```
struct
  {
  ushort Status;      /* Error Code        */
  ushort Convert;     /* See below         */
  ushort EntryCount;  /* Entries returned  */
  ushort AvailCount;  /* Entries available */
  }
```

`Status`

An error code. Available codes are listed in the Leach/Naik Browser draft.

`Convert`

More on this in a moment, when we get to the `Data[]` block.

`EntryCount`

The number of entries returned in the reply.

`AvailCount`

The number of available entries. This may be more than the number in `EntryCount`, in which case there are more entries than will fit in the data buffer length given in the request.

That's all there is to the `Parameter[]` block. It's nicely simple, but things get a little wilder as we move on. Do keep track of that `Convert` value...

The SMB-level Data[] block will start with a series of ServerInfo_1 structures, as described below:

```
struct
  {
  uchar  Name[16];      /* Provider name     */
  uchar  OSMajorVers;   /* Provider OS Rev   */
  uchar  OSMinorVers;   /* Provider OS Point */
  ulong  ServerType;    /* See below         */
  uchar *Comment;       /* Pointer           */
  } ServerInfo_1;
```

There will be <EntryCount> such structures packed neatly together. It is fairly easy to parse them out, because the Name field is a fixed-length, nul-padded string and the Comment field *really is* a pointer. The Leach/Naik Browser draft suggests that the Comment strings themselves *may* follow each ServerInfo_1 structure, but all examples seen on the wire show four bytes. Hang on to those four bytes... we'll explain in a moment.

Anywhich, the above structure has a fixed length — 26 bytes, to be precise. That makes it easy to parse ServerInfo_1 structures from the Data[] block.

The values in the ServerInfo_1 are the same ones announced by the Provider in its HostAnnouncement or DomainAnnouncement frames. They are stored in an internal database on the browser node. Some of these fields have been discussed before, but a detailed description of the ServerType field has been postponed at every opportunity. Similarly, the pointer value in the Comment field really needs some clarification.

Let's start with the Comment pointer...

The Comment pointer may just possibly be a relic of the long lost days of DOS. Those who know more about 16-bit DOS internals may judge. In any case, what you need to do is this:

- Read the Comment pointer from the ServerInfo_1 structure.
- Remove the two higher-order bytes: Comment & 0x0000FFFF.
- Subtract the value of the Convert field: offset = (Comment & 0x0000FFFF) - Convert.
- Use the resulting offset to find the actual Comment string. The offset is relative to the start of the SMBtrans Data[] block.

Well *that* was easy. This stuff is so lovable you just want to give it a hug, don't you?

Some further notes:

- The `Comment` strings are stored in the RAP-level heap.
- The `ServerInfo_1` blocks are considered RAP-level "data."
- Both of those are collected into the SMBtrans-level `Data[]` block.
- Just to make things simple, the RAP-level parameters are gathered into the SMBtrans `Parameter[]` block.

Right... Having tilted that windmill, let's take a look at the (more sensible, but also much more verbose) `ServerType` field. We have delayed describing this field for quite a while. Here, finally, it is... well, mostly. The list below is based on Samba sources. It is close to Ethereal's list, and less close to the list given in the Leach/Naik draft. Let the buyer beware.

Browser Provider type bits

Bit	Name / Bitmask	Description
31	`SV_TYPE_DOMAIN_ENUM` `0x80000000`	Enumerate Domains. This bit is used in the request to ask for a list of known workgroups instead of a list of Providers in a workgroup.
30	`SV_TYPE_LOCAL_LIST_ONLY` `0x40000000`	This bit identifies entries for which the browser is *authoritative*. That is, it is set if the Provider (or workgroup) entry was received via an announcement message, and clear if the entry is the result of a sync with the DMB.
29	`SV_TYPE_ALTERNATE_XPORT` `0x20000000`	No one seems to remember where this came from or what it means. Ethereal doesn't know about it.
28–24	`0x1F000000`	Unused.
23	`SV_TYPE_DFS_SERVER` `0x00800000`	The Provider offers DFS shares. Possibly a DFS root.
22	`SV_TYPE_WIN95_PLUS` `0x00400000`	Indicates a Provider that considers itself to be in the Windows 9x family.
21	`SV_TYPE_SERVER_VMS` `0x00200000`	Indicates a VMS (Pathworks) server.

Browser Provider type bits

Bit	Name / Bitmask	Description
20	SV_TYPE_SERVER_OSF 0x00100000	Indicates an OSF Unix server.
19	SV_TYPE_DOMAIN_MASTER 0x00080000	Indicates a Domain Master Browser (DMB).
18	SV_TYPE_MASTER_BROWSER 0x00040000	Indicates a Local Master Browser (LMB).
17	SV_TYPE_BACKUP_BROWSER 0x00020000	Indicates a Backup Browser...
16	SV_TYPE_POTENTIAL_BROWSER 0x00010000	...and, of course, a Potential Browser.
15	SV_TYPE_SERVER_NT 0x00008000	Indicates a Windows NT Server.
14	SV_TYPE_SERVER_MFPN 0x00004000	Unknown. Ethereal ignores this one, and it's not listed in the Leach/Naik Browser draft.
13	SV_TYPE_WFW 0x00002000	Windows for Workgroups.
12	SV_TYPE_NT 0x00001000	A Windows NT client.
11	SV_TYPE_SERVER_UNIX 0x00000800	An SMB server running Xenix or Unix. Samba will set this bit when announcing its services.
10	SV_TYPE_DIALIN_SERVER 0x00000400	The Provider offers dial-up services (e.g. NT RAS).
9	SV_TYPE_PRINTQ_SERVER 0x00000200	The Provider has printer services available.
8	SV_TYPE_DOMAIN_MEMBER 0x00000100	The Provider is a member of an NT Domain. That means that the Provider itself has authenticated to the NT Domain.
7	SV_TYPE_NOVELL 0x00000080	The Provider is a Novell server offering SMB services. This is probably used with SMB over IPX/SPX, but may be set by Novell's SMB implementation as well.
6	SV_TYPE_AFP 0x00000040	The Provider is an Apple system. Thursby's Dave product and Apple's SMB implementation may set this bit.

Browser Provider type bits

Bit	Name / Bitmask	Description
5	SV_TYPE_TIME_SOURCE 0x00000020	The Provider offers SMB time services. (Yes, there is an SMB-based time sync service.)
4	SV_TYPE_DOMAIN_BAKCTRL 0x00000010	The Provider is a Backup Domain Controller (BDC).
3	SV_TYPE_DOMAIN_CTRL 0x00000008	The Provider is a Domain Controller.
2	SV_TYPE_SQLSERVER 0x00000004	The Provider offers SQL services.
1	SV_TYPE_SERVER 0x00000002	The Provider offers SMB file services.
0	SV_TYPE_WORKSTATION 0x00000001	This bit indicates that the system is a workstation. (Just about everything sets this bit.)

Just to polish this subject off, here's a little code that can parse a `NetServerEnum2` response message and print the results:

Listing 22.3: Parsing `NetServerEnum2` Replies

```
#define NERR_Success 0

#define SV_TYPE_ALL               0xFFFFFFFF
#define SV_TYPE_UNKNOWN           0x1F000000

#define SV_TYPE_DOMAIN_ENUM       0x80000000
#define SV_TYPE_LOCAL_LIST_ONLY   0x40000000
#define SV_TYPE_ALTERNATE_XPORT   0x20000000
#define SV_TYPE_DFS_SERVER        0x00800000
#define SV_TYPE_WIN95_PLUS        0x00400000
#define SV_TYPE_SERVER_VMS        0x00200000
#define SV_TYPE_SERVER_OSF        0x00100000
#define SV_TYPE_DOMAIN_MASTER     0x00080000
#define SV_TYPE_MASTER_BROWSER    0x00040000
#define SV_TYPE_BACKUP_BROWSER    0x00020000
#define SV_TYPE_POTENTIAL_BROWSER 0x00010000
#define SV_TYPE_SERVER_NT         0x00008000
#define SV_TYPE_SERVER_MFPN       0x00004000
#define SV_TYPE_WFW               0x00002000
#define SV_TYPE_NT                0x00001000
#define SV_TYPE_SERVER_UNIX       0x00000800
```

```
#define SV_TYPE_DIALIN_SERVER     0x00000400
#define SV_TYPE_PRINTQ_SERVER     0x00000200
#define SV_TYPE_DOMAIN_MEMBER     0x00000100
#define SV_TYPE_NOVELL            0x00000080
#define SV_TYPE_AFP               0x00000040
#define SV_TYPE_TIME_SOURCE       0x00000020
#define SV_TYPE_DOMAIN_BAKCTRL    0x00000010
#define SV_TYPE_DOMAIN_CTRL       0x00000008
#define SV_TYPE_SQLSERVER         0x00000004
#define SV_TYPE_SERVER            0x00000002
#define SV_TYPE_WORKSTATION       0x00000001

typedef struct
  {
  ushort Status;
  ushort Convert;
  ushort EntryCount;
  ushort AvailCount;
  } NSE2_ReplyParams;

void PrintBrowserBits( ulong ServerType )
  /* ------------------------------------------------- **
   * Itemize Browse Service Provider Type Bits.
   * This is boring, and could probably be done better
   * using an array and a for() loop.
   * ------------------------------------------------- **
   */
  {
  if( SV_TYPE_ALL == ServerType )
    {
    printf( "  All/Any Server types.\n" );
    return;
    }

  if( SV_TYPE_UNKNOWN & ServerType )
    printf( "  Warning: Undefined bits set.\n" );

  if( SV_TYPE_DOMAIN_ENUM & ServerType )
    printf( "  Enumerate Domains\n" );
  if( SV_TYPE_LOCAL_LIST_ONLY & ServerType )
    printf( "  Local List Only\n" );
  if( SV_TYPE_ALTERNATE_XPORT & ServerType )
    printf( "  Alternate Export (Unknown type)\n" );
  if( SV_TYPE_DFS_SERVER & ServerType )
    printf( "  DFS Support\n" );
  if( SV_TYPE_WIN95_PLUS & ServerType )
    printf( "  Windows 95+\n" );
```

```
if( SV_TYPE_SERVER_VMS & ServerType )
  printf( "  VMS (Pathworks) Server\n" );
if( SV_TYPE_SERVER_OSF & ServerType )
  printf( "  OSF Unix Server\n" );
if( SV_TYPE_DOMAIN_MASTER & ServerType )
  printf( "  Domain Master Browser\n" );
if( SV_TYPE_MASTER_BROWSER & ServerType )
  printf( "  Local Master Browser\n" );
if( SV_TYPE_BACKUP_BROWSER & ServerType )
  printf( "  Backup Browser\n" );
if( SV_TYPE_POTENTIAL_BROWSER & ServerType )
  printf( "  Potential Browser\n" );
if( SV_TYPE_SERVER_NT & ServerType )
  printf( "  Windows NT (or compatible) Server\n" );
if( SV_TYPE_SERVER_MFPN & ServerType )
  printf( "  MFPN (Unkown type)\n" );
if( SV_TYPE_WFW & ServerType )
  printf( "  Windows for Workgroups\n" );
if( SV_TYPE_NT & ServerType )
  printf( "  Windows NT Workstation\n" );
if( SV_TYPE_SERVER_UNIX & ServerType )
  printf( "  Unix/Xenix/Samba Server\n" );
if( SV_TYPE_DIALIN_SERVER & ServerType )
  printf( "  Dialin Server\n" );
if( SV_TYPE_PRINTQ_SERVER & ServerType )
  printf( "  Print Server\n" );
if( SV_TYPE_DOMAIN_MEMBER & ServerType )
  printf( "  NT Domain Member Server\n" );
if( SV_TYPE_NOVELL & ServerType )
  printf( "  Novell Server\n" );
if( SV_TYPE_AFP & ServerType )
  printf( "  Apple Server\n" );
if( SV_TYPE_TIME_SOURCE & ServerType )
  printf( "  Time Source\n" );
if( SV_TYPE_DOMAIN_BAKCTRL & ServerType )
  printf( "  Backup Domain Controller\n" );
if( SV_TYPE_DOMAIN_CTRL & ServerType )
  printf( "  Domain Controller\n" );
if( SV_TYPE_SQLSERVER & ServerType )
  printf( "  SQL Server\n" );
if( SV_TYPE_SERVER & ServerType )
  printf( "  SMB Server\n" );
if( SV_TYPE_WORKSTATION & ServerType )
  printf( "  Workstation\n" );
} /* PrintBrowserBits */
```

```
void PrintNetServerEnum2Reply( uchar *ParamBlock,
                               int    ParamLen,
                               uchar *DataBlock,
                               int    DataLen )
  /* ------------------------------------------------------ **
   * Parse a NetServerEnum2 Reply and print the contents.
   * ------------------------------------------------------ **
   */
  {
  NSE2_ReplyParams Rep;
  int              i;
  int              offset;
  uchar           *pos;

  /* Check for an obvious error.
   */
  if( ParamLen != 8 )
    Fail( "Error parsing NetServerEnum2 reply.\n" );

  /* Grab all of the parameter words.
   */
  Rep.Status     = smb_GetShort( ParamBlock, 0 );
  Rep.Convert    = smb_GetShort( ParamBlock, 2 );
  Rep.EntryCount = smb_GetShort( ParamBlock, 4 );
  Rep.AvailCount = smb_GetShort( ParamBlock, 6 );

  /* Check for problems (errors and warnings).
   */
  if( Rep.Status != NERR_Success )
    Fail( "NetServerEnum2 Error: %d.\n", Rep.Status );
  if( Rep.EntryCount < Rep.AvailCount )
    printf( "Warning: The list is incomplete.\n" );

  /* Dump the ServerInfo_1 records. */
  pos = DataBlock;
  for( i = 0; i < Rep.EntryCount; i++ )
    {
    printf( "%-15s V%d.%d\n", pos, pos[16], pos[17] );
    PrintBrowserBits( smb_GetLong( pos, 18 ) );
    offset  = 0x0000FFFF & smb_GetLong( pos, 22 );
    offset -= Rep.Convert;
    if( offset >= DataLen )
      Fail( "Packet offset error.\n" );
    printf( "  Comment: %s\n", (DataBlock + offset) );
    pos += 26;
    }
  } /* PrintNetServerEnum2Reply */
```

22.3.3 *On the Outskirts of Town*

There is another RAP call that you need to know about. It comes in handy at times. It doesn't really belong to the Browse Service, but you may have heard its name mentioned in that context. It lives on the edge, somewhere between browsing and filesharing, and it goes by the name `NetShareEnum`.

The `NetShareEnum` RAP call does the job of listing the shares offered by a server. The shares, as you already know, are the virtual roots of the directory trees made available via SMB.

The wire format of the request is as follows:

```
struct
  {
  ushort RAPCode;
  uchar *ParamDesc;
  uchar *DataDesc;
  struct
    {
    ushort InfoLevel;
    ushort BufrSize;
    } Params;
  } NetShareEnumReq;
```

and it is filled in like so:

```
NetShareEnumReq
  {
  RAPCode   = 0 (NetShareEnum)
  ParamDesc = "WrLeh"
  DataDesc  = "B13BWz"
  Params
    {
    InfoLevel = 1 (No other values defined)
    BufrSize  = <Same as smb_Transaction_Request.MaxDataCount>
    }
  }
```

Yes, the RAP code for `NetShareEnum` is zero (`0`).

There's not much to that call, particularly once you've gotten the `NetServerEnum2` figured out. The response also contains some familiar concepts. In fact, the `Parameter[]` section is exactly the same.

The RAP-level data section is supposed to contain an array of `ShareInfo_1` structures, which look like this:

```
struct
  {
  uchar  ShareName[13];
  uchar  pad;
  ushort ShareType;
  uchar *Comment;
  } ShareInfo_1;
```

Again, there are many similarities to what we have seen before. In this case, though, the `ShareType` field has a smaller set of possible values than the comparable `ServerType` field.

Share type values

Name	Value	Description
`STYPE_DISKTREE`	`0`	A disk share (root of a directory tree).
`STYPE_PRINTQ`	`1`	A print queue.
`STYPE_DEVICE`	`2`	A communications device (e.g. a modem).
`STYPE_STYPE`	`3`	An Inter-Process Communication (IPC) share.

...and that is "all you need to know" about the `NetShareEnum` call. Oh, wait... There is one more thing...

Can't Get It Out Of My Head Alert

There is one great big warning regarding the `NetShareEnum` *response. Some Windows systems have been seen returning parameter blocks that are very large (e.g. 1024 bytes). The first eight bytes contain the correct values. The rest appear to be left-over cruft from the buffer on the server side. The server is returning the buffer size (and the whole buffer) rather than the* `Parameter[]` *block size.*

Other transactions may exhibit similar behavior.

22.3.4 *Transaction Fragmentation*

A promise is a promise, and we did promise to cover fragmented transactions.[2]

The idea is fairly simple. If you have twenty-five sacks of grain to bring to town, and a wagon that can hold only twelve sacks, then you will need to

2. Maybe we could cover them with leaves and fallen branches and just let them compost themselves quietly in an out-of-the-way place or something.

make a few trips. Likewise with transactions. Due to the limits of the negotiated buffer size, a transaction may attempt to transfer more data than can be carried in a single SMB. The solution is to split up the data and send it using multiple SMB messages.

The mechanism used is the same for SMBtrans, Trans2, and NTtrans. There are slight differences between the transaction request and transaction response, though, so pay attention.

Sending a fragmented transaction request works like this:

1. Fill in the transaction SMB, packing as many `Parameter[]` and `Data[]` bytes into the transaction as possible. `Parameter[]` bytes have precedence over `Data[]` bytes.
2. Send the initial message and wait for a reply (known as the "Interim Server Response"). This step is a shortcut. It gives the server a chance to reject the transaction before it has been completely transferred. Only the response header has any meaning. If there is no error, the transaction may proceed.
3. Send as many secondary transaction messages as necessary to transfer the remaining `Parameter[]` and `Data[]` bytes. Note that the SMB command and structure of secondary transactions is not the same as those of the initial message.
4. Wait for the server to execute the transaction and return the results.

Now go back and take a look at Listing 22.1. Note that the `smb_Transaction_Request` structure keeps track of the number of `Parameter[]` and `Data[]` bytes already packed for shipping. That makes it easy to build the secondary messages should they be needed.

Fragmenting a transaction response is a simpler process. The response SMBs all have the same structure (no special secondary messages) and they all have an SMB header, which may contain an error code if necessary. So, the server can just send as many transaction response SMBs as needed to transfer all of the results. That's it.

22.3.5 *RAP Annoyances*

RAP can be quite annoying — that's just its nature. There are two particular annoyances of which you should be aware:

Authentication

It is common for a server to deny a `NetShareEnum` request on an anonymous SMB connection. A valid username/password pair may be required. Some servers also require non-anonymous authentication for the `NetServerEnum2` request, though this is less common.

Limitations and Permutations

Grab a capture of a `NetShareEnum` request and take a look at the data descriptor string for the returned data (which should be "B13BWz", as described above). The number 13 in the string indicates the maximum length of the share names to be returned, and it includes the terminating nul byte.

"B13BWz" means that the `NetShareEnum` function will not return share names with a string length greater than 12 characters each. Of course, there are many systems that can offer shares with names longer than 12 characters. Thus, we have a problem.

One way to solve the problem would be to change the field size in the descriptor string to, for example, something like "B256BWz". That trick isn't likely to work against all servers, however, because the descriptor strings are constant enough that some implementations probably ignore them. Another option is to use short share names, but that only works if you have control over all of the SMB servers in the network.

The prescribed solution is to use a different function, called `NetrShareEnum`. Note that there's an extra letter 'r' hidden in there. Also note that the `NetrShareEnum` function is an MS-RPC call, not a RAP call, and thus is beyond the scope of this book. You can watch for it in packet captures, however, and possibly consider it as a starting point should you decide to explore the world of MS-RPC.

So, now you know.