6

The Session Service in Detail

> The best way to eliminate the problem is to remove Scopes completely.
>
> — John Terpstra, Samba Team, in a message to the Samba-Technical mailing list

This is the last big chunk of NBT. It is also the easiest, which should bring a great sigh of relief. We have already covered all of the background material we need to cover, so there is no need to waste any time with preliminaries. Let's dive right in...

6.1 Session Service Header

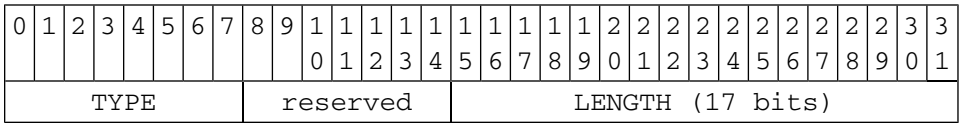
The Session Service header, as presented in RFC 1002, is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TYPE								FLAGS							
LENGTH															

The FLAGS field breaks down further into:

0	1	2	3	4	5	6	7
reserved							E

The reserved bits are always supposed to be zero, and the E bit is an additional high-order bit which is prepended to the LENGTH field. Another way to look at the layout is like this:



We will stick with the latter, simpler format and ignore the FLAGS field, which is never really used.

The LENGTH field contains the number of bytes of payload, and the TYPE field is used to distinguish between the six different Session Service message types, which are:

0x00 == Session Message
0x81 == Session Request
0x82 == Positive Session Response
0x83 == Negative Session Response
0x84 == Retarget Session Response
0x85 == Session Keepalive

Each of these message types is explained below.

6.2 Creating an NBT Session

The first step in setting up an NBT session is to discover the IP address of the remote node. The IP address is, of course, required in order to create the TCP session that will carry the NBT session. The NBT Name Service is generally used to find the remote host's IP address, though several implementations support kludges which bypass the Name Service. Once the TCP session is established (something we assume you know how to do) the NBT session is initiated using a SESSION REQUEST message, which looks like this:

```
SESSION REQUEST
  {
  HEADER
    {
    TYPE   = 0x81 (Session Request)
    LENGTH = 68   (See discussion below)
    }
  CALLED_NAME  = <Destination Level 2 Encoded NetBIOS name>
  CALLING_NAME = <Source Level 2 Encoded NetBIOS name>
  }
```

One oddity of the Session Service is that the Scope ID is dropped from the name fields in the `SESSION REQUEST` message. That results in a fixed length of 34 bytes per name. That's one byte for the leading label (always `0x20`), 32 bytes for the First Level Encoded NetBIOS name, and one more byte for the trailing label (always `0x00`). The payload of a `SESSION REQUEST` message is, therefore, fixed at 2 × 34 = 68 bytes.

Caveat Alert

The RFCs do not specify whether the Scope ID should or should not be included in the `CALLED` *or* `CALLING NAME`*. It would make sense to assume that the Scope ID should be included, since both the Name Service and Datagram Service require the Scope ID, but that's not how things actually work on the wire.*

As it is, the behavior of the Session Service is inconsistent with the rest of the NBT system. Fortunately, Scope is enforced by the Name Service, so it is not critical that it be enforced by the Session Service.

There are three possible replies to the `SESSION REQUEST` message:

0x82: POSITIVE SESSION RESPONSE

The remote node has accepted the session request, and the session is established. Kewl!

```
POSITIVE SESSION RESPONSE
  {
  HEADER
    {
    TYPE   = 0x82
    LENGTH = 0
    }
  }
```

0x83: NEGATIVE SESSION RESPONSE

Something went wrong, and the remote node has rejected the session request.

```
NEGATIVE SESSION RESPONSE
  {
  HEADER
    {
    TYPE   = 0x83
    LENGTH = 1
    }
  ERROR_CODE = <A Session Service Error Code>
  }
```

The one-byte ERROR_CODE field is supposed to indicate the cause of the trouble. Possible values are:

0x80: Not Listening On Called Name

The remote node has registered the CALLED NAME, but no application or service is listening for session connection requests on that name.

0x81: Not Listening For Calling Name

The remote node has registered the CALLED NAME and is listening for connections, but it doesn't want to talk to you. It is expecting a call from some other CALLING NAME.

There are some interesting implications to this. It means that a server could, potentially, be selective about which nodes may connect. On the other hand, it would be trivial to spoof the CALLING NAME.

0x82: Called Name Not Present

The remote node has not even registered the CALLED NAME. Better re-try your name resolution.

0x83: Insufficient Resources

The remote node is busy and cannot take your call at this time.

0x8F: Unspecified Error

Something is wrong on the far end, but we are not quite sure what the problem is.

It is annoying that the error code values overlap the Session Service message type values.

0x84: RETARGET SESSION RESPONSE

This Session Service message tells the calling node to try a different IP address and/or port number, something like a `Redirect` directive on a web page. When a client receives a `RETARGET SESSION RESPONSE` message in response to a `SESSION REQUEST`, it is supposed to close the existing TCP connection and open a new one using the IP address and port number provided.

```
RETARGET SESSION RESPONSE
  {
  HEADER
    {
    TYPE   = 0x84
    LENGTH = 6
    }
  RETARGET_IP_ADDRESS = <New IP address>
  PORT                = <New TCP port number>
  }
```

This feature opens up some interesting possibilities. Retargeting could be used for load balancing, fault tolerance, or to allow unprivileged users to run their own SMB servers on high-numbered ports.

Of course, client support for this feature is inconsistent. Based on some simple tests, it seems that Samba's `smbclient` handles retargeting just fine, as do Windows 95 and Windows 98. In contrast, Windows 2000 deals with the `RETARGET SESSION RESPONSE` as if it were an error message of some sort. W2K will retry the original IP address and port number, and then give up.

Listing 6.1: Session retargeting

```
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
```

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

void PrintL1Name( uchar *src, int max )
  /* ------------------------------------------------- **
   * Decode and pretty-print an L1-encoded NetBIOS name.
   * ------------------------------------------------- **
   */
  {
  int         suffix;
  static char namestr[16];

  suffix = L1_Decode( namestr, src, 1, max );
  Hex_Print( namestr, strlen( namestr ) );
  printf( "<%.2x>", suffix );
  }/* PrintL1Name */

int Get_SS_Length( uchar *hdr )
  /* ------------------------------------------------- **
   * Read the length field from an SMB Session Service
   * header.
   * ------------------------------------------------- **
   */
  {
  int tmp;

  tmp  = (hdr[1] & 1) << 16;
  tmp |= hdr[2] << 8;
  tmp |= hdr[3];
  return( tmp );
  } /* Get_SS_Length */

int OpenPort139( void )
  /* ------------------------------------------------- **
   * Open port 139 for listening.
   * Note: this requires root privilege, and Samba's
   *       SMBD daemon must not be running on its
   *       default port.
   * ------------------------------------------------- **
   */
  {
  int                result;
  int                sock;
  struct sockaddr_in sox;
```

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

  /* Bind the socket to any interface, port TCP/139.  */
  sox.sin_addr.s_addr = INADDR_ANY;
  sox.sin_family      = AF_INET;
  sox.sin_port        = htons( 139 );
  result = bind( sock,
                 (struct sockaddr *)&sox,
                 sizeof(struct sockaddr_in) );
  if( result < 0 )
    {
    printf( "Failed to bind() socket; %s.\n",
            strerror( errno ) );
    exit( EXIT_FAILURE );
    }

  /* Post the listen request. */
  result = listen( sock, 5 );
  if( result < 0 )
    {
    printf( "Failed to listen() on socket; %s.\n",
            strerror( errno ) );
    exit( EXIT_FAILURE );
    }

  /* Ready... */
  return( sock );
  } /* OpenPort139 */

void Listen( struct in_addr trg_addr, int trg_port )
  /* ------------------------------------------------- **
   * Accepts incoming connections, sends a retarget
   * message, and then disconnects.
   * ------------------------------------------------- **
   */
  {
  int                listen_sock;
  int                reply_sock;
  int                result;
  struct sockaddr_in remote_addr;
```

```
socklen_t          addr_len;
uchar              recvbufr[1536];
uchar              replymsg[10];

listen_sock = OpenPort139();

/* Fill in the redirect message. */
replymsg[0] = 0x84;   /* Retarget code. */
replymsg[1] = 0;
replymsg[2] = 0;
replymsg[3] = 6;      /* Remaining length. */
(void)memcpy( &(replymsg[4]), &trg_addr.s_addr, 4 );
trg_port = htons( trg_port );
(void)memcpy( &(replymsg[8]), &trg_port, 2 );

printf( "Waiting for connections...\n" );
for(;;)  /* Until killed. */
  {
  /* Wait for a connection. */
  addr_len = sizeof( struct sockaddr_in );
  reply_sock = accept( listen_sock,
                       (struct sockaddr *)&remote_addr,
                       &addr_len );

  /* If the accept() failed exit with an error message. */
  if( reply_sock < 0 )
    {
    printf( "Error accept()ing a connection: %s\n",
            strerror(errno) );
    exit( EXIT_FAILURE );
    }

  result = recv( reply_sock, recvbufr, 1536, 0 );
  if( result < 0 )
    {
    printf( "Error receiving packet: %s\n",
            strerror(errno) );
    }
  else
    {
    printf( "SESSION MESSAGE\n  {\n" );
    printf( "  TYPE   = 0x%.2x\n", recvbufr[0] );
    printf( "  LENGTH = %d\n", Get_SS_Length( recvbufr ) );
```

```
      if( 0x81 == recvbufr[0] )
        {
        int offset;

        printf( "  CALLED_NAME  = " );
        PrintL1Name( &recvbufr[4], result );
        offset = 5 + strlen( &(recvbufr[4]) );
        printf( "\n  CALLING_NAME = " );
        PrintL1Name( &recvbufr[offset], result );
        printf( "\n  }\nSending Retarget message.\n" );
        (void)send( reply_sock, (void *)replymsg, 10, 0 );
        }
      else
        printf( "  }\nPacket Dropped.\n" );
      }
    close( reply_sock );
    }
  } /* Listen */

int main( int argc, char *argv[] )
  /* ------------------------------------------------- **
   * Simple daemon that listens on port TCP/139 and
   * redirects incoming traffic to another IP and port.
   * ------------------------------------------------- **
   */
  {
  int            target_port;
  struct in_addr target_address;

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

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

  target_port = atoi( argv[2] );
```

```
if( 0 == target_port )
  {
  printf( "Invalid Port number.\n" );
  printf( "Usage:  %s <IP> <PORT>\n", argv[0] );
  exit( EXIT_FAILURE );
  }

Listen( target_address, target_port );
return( EXIT_SUCCESS );
} /* main */
```

One more note regarding the Retarget message: there are NetBIOS name issues to consider. The `CALLED NAME` must be in the name table of the node that sends the `RETARGET SESSION RESPONSE` message, but it must also be accepted by the node to which the session is retargeted. That may take some juggling to get right.

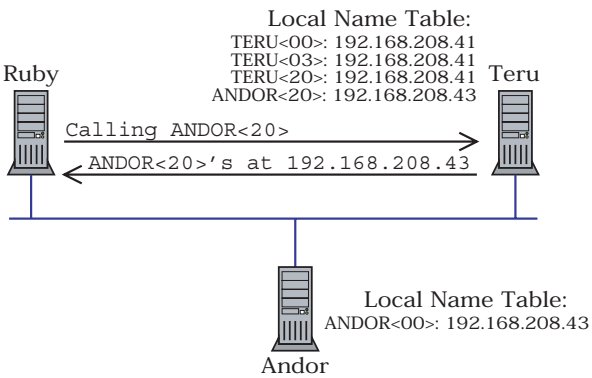


Figure 6.1: *Naming and session retargeting*

Node `Ruby` is trying to open a connection to a service named `ANDOR<20>`. Node `Teru` has the name `ANDOR<20>` in its local name table, so `Ruby` tries to connect to node `Teru`. `Teru` retargets `Ruby` to IP address 192.168.208.43 which (we hope) will accept the connection from `Ruby`.

The `RETARGET SESSION RESPONSE` message does not work well with normal NetBIOS name management.

For those interested in playing with retargeting, it is fairly easily done. Samba's `smbd` daemon can be told to listed on a non-standard port and, as a bonus, it ignores the `CALLED NAME` in the session request. You can run the retarget daemon listed above in combination with the Samba

nmbd Name Service daemon, and retarget connections to smbd running on a high port on the same machine, or running on a remote machine.

6.3 Maintaining an NBT Session

There are two more Session Service message types to cover:

0x00: SESSION MESSAGE

Once you have established a session (by sending a SESSION REQUEST and receiving a POSITIVE SESSION RESPONSE) you are ready to send messages. Each message is prefixed with a SESSION MESSAGE header, which looks like this:

```
HEADER
  {
  TYPE   = 0x00
  LENGTH = <Length of data to follow>
  }
```

Since the TYPE byte has a value of 0x00, and the next seven bits are always supposed to be zero as well, the Session Message header may be viewed simply as a long integer length value.

```
length  = ntohl( *(ulong *)packet );
```

It might be wise to mask out the unused FLAGS bits, just in case.

0x85: SESSION KEEPALIVE

The Keepalive is used to detect a lost connection. Basically, if one node hasn't sent anything to the other node for a while (typically five to six minutes), it will send a SESSION KEEPALIVE, just to make sure the remote end is still listening. The receiver simply discards the message.

```
HEADER
  {
  TYPE   = 0x85
  LENGTH = 0
  }
```

TCP is a connection-oriented protocol, so the Keepalive should generate an ACKnowledgement, or possibly a series of retries if the TCP

ACK doesn't show up right away. The Keepalive message forces TCP to verify that the connection is still working, and to report back if there is a problem. If a problem *is* detected, the client or server can gracefully shut down its end of the connection.

RFC 1001 makes it clear that sending the NBT Session Service Keepalive message is optional. TCP itself also has a keepalive mechanism, which should be used instead, if possible.

6.4 Closing an NBT Session

Nothing to it. Once all activity across the session has stopped, simply shut down the TCP connection. At the NBT level, there are no special messages to send when closing the session.