An implementation of the SMB protocol in Erlang

Torbjörn Törnkvist
Nortel Networks
P.O. Box 6701
113 85 Stockholm, Sweden
tobbe@nortelnetworks.com

ABSTRACT

This paper describes the implementation of a subset of the SMB protocol in Erlang. We discuss the motivation for this work and its outcome, and compare the performance and memory consumption of our implementation with Samba.

Categories and Subject Descriptors

C.2.2 [**Computer Communications**]: Network Protocols; D.1 [**Software**]: Programming Techniques; D.1 [**Software**]: Programming Languages

1. INTRODUCTION

This work started as a hobby project. The author wanted to learn more about SMB (Server Message Block) [6], the underlying protocol of Microsoft's Windows Networking system. However, in his profession the author is a developer of Nortel's SSL-VPN product.[1] The SSL-VPN's control system is written in Erlang, and it uses the Open Source software package *Samba* [5, 1] [2] to access SMB file shares from Windows servers, and for some related tasks. Samba initially worked well for this task, but over time some drawbacks became a significant problem. Suddenly, the author's hobby project was able to solve the SMB problems in the SSL-VPN by replacing Samba.

1.1 Problems to be solved

One major problem was the size of the SSL-VPN software. Continually adding new features was increasing the overall size of the software and, despite attempts to trim the code down, we were reaching a limit. Removing Samba reduces the total code volume by 10%.

Another problem was the high memory consumption of a Samba process that we open via an Erlang port. One such process was opened for each Windows server to be accessed. The port mechanism was also a potential bottleneck since only one request-reply transaction could be served at a time.

The SSL-VPN product was also being enhanced with internationalization support. This would require us to modify the character handling mechanism in Samba.

There were many other smaller problems. We had already had to make small modifications to Samba to fit our needs, and these changes had to be reconciled when importing new Samba releases. We also had little control over the error messages that Samba produced.

2. SOME ERLANG MECHANISMS

We will assume that the reader has some knowledge of the Erlang programming language. Therefore, we will only highlight a few language mechanisms that are crucial for the rest of the discussion.

2.1 Port programs

Erlang [3] is not just a programming language. It also emulates its own little Operating System (for example: it has its own process scheduler and code loading mechanism.) When you start Erlang it runs as one process on the host operating system.[4] To be able to access external libraries and other kinds of external software we can make use of the *p*ort mechanism in Erlang.

Opening a port means causing the Erlang emulator to create a new OS process to run a given external program. A pipe is used for communication between Erlang and the external port program.

One good aspect of this mechanism is that any fatal error in the external software will not affect the running Erlang system. A drawback is that the context switch between the two processes and the communication over the pipe may be costly.

2.2 Linked-in drivers

An alternative to external port processes is to dynamically link object code directly into the Erlang emulator. These

[1] A remote access security solution that extends the reach of enterprise applications and file systems to mobile workers, telecommuters, partners, and customers.
[2] Samba (www.samba.org)

[3] i.e the implementation of Erlang
[4] In this article we will assume that Erlang is running on Linux

Erlang'04, September 22, 2004, Snowbird, Utah, USA.

"linked-in driver" modules are written in C according to a special Erlang driver interface.

With this approach we avoid the OS process context switches when communicating between Erlang and the driver code. The downside is that we can't recover from a fatal flaw in the driver code – it will crash the emulator itself.

2.3 The Erlang Bit Syntax

In Erlang, the *Binary* data type denotes an opaque chunk of bytes. Binaries are constructed with a special syntax. For example, a four-byte binary could be constructed as `<<1, 2, 1024:16>>`, where the first two bytes are given individually and the second two are specified as the big-endian encoding of the 16-bit integer 1024. The value 16 in this example is called a *size expression*. It gives the size of the element in *units* that depend on the element type. In this case the units are bits, which are the default for integer elements.

A Binary can be deconstructed (decoded) into segments by means of pattern matching. A segment is a set of contiguous bits of the binary (not necessarily on a byte boundary). The example below shows how we can define a function `bitrev` to reverse the bit order of a binary consisting of only one byte:

```
bitrev(<<A:1,B:1,C:1,D:1,E:1,F:1,G:1,H:1>>) ->
  <<H:1,G:1,F:1,E:1,D:1,C:1,B:1,A:1>>.
```

We can also include a *type specifier* for an element to say how it is coded with respect to *signedness* and *endianness*. Since the SMB protocol represents integer values as little-endian we can encode/decode these values very elegantly using the *little* type specifier. For example, the expression `<<H:16/little, T/binary>> = Bin` decodes a 16 bit integer in little-endian format and binds the result to the variable `H`. The rest of the Binary is then bound to the variable `T`.

3. USING THE SMB PROTOCOL

SMB is a client-server protocol that allows clients to request access to resources on a server, and is the underlying protocol of the Microsoft Windows networking services. We won't delve into the history of the SMB protocol here, the interested reader can find historical information in [5]. One historical implication is that SMB runs over a protocol called NetBIOS, which in its turn can run over TCP as defined in [3, 4].

There are a number of SMB implementations. One of the most well-known (non-Microsoft) is the Open Source implementation named Samba. Samba makes it possible to integrate other operating systems into a Microsoft Windows network environment. For example, Samba makes Windows file shares available to machines running Linux, and vice-versa.

The original Erlang control system of the Nortel SSL-VPN product interfaced with Samba via an Erlang port program. This port mechanism was used by the SSL-VPN's Web portal to allow users to manipulate file shares using a Web browser.

The set of operations required by the SSL-VPN Web portal are: list/create/remove directories and download/upload/delete files. These operations require only about a dozen different types of SMB protocol messages.

In addition, the SSL-VPN is able to list the available shares on a Windows machine. This is using an extension of the SMB protocol called the *Lan Manager Remote Administration Protocol, (RAP)*, an RPC-like protocol that runs on top of SMB.

4. THE IMPLEMENTATION

4.1 Setup of a SMB session

To perform a file sharing operation on an SMB server we first need to setup an SMB session. This begins with making a TCP-connect to the server on the standard port 139, followed by these four steps:

1. *NetBIOS session setup* - After the TCP connection is made a NetBIOS session is established between the client and the server. Each SMB PDU[5] begins with a four-byte NetBIOS header, as this snippet demonstrates: `Session-Service` header:

   ```
   -define(SESSION_SERVICE, 16#00).

   nbss_session_service_pdu(SMB_pdu) ->
     Length = size(SMB_pdu),
     <<?SESSION_SERVICE, 0, Length:16,
       SMB_pdu/binary>>.
   ```

2. *Protocol negotiation* - The SMB protocol allows the client and server to negotiate for a mutually-compatible protocol version.[6] The result of this negotiation determines such things as how authentication will be performed and whether to use *Unicode*.[7]

3. *Authentication* - The client sends a `Session-Setup` SMB requests containing a *Username* and *Password*. The password may be encrypted in one of two ways or sent in cleartext, depending on the result of the earlier protocol negotiation.

4. *Resource request* - The client sends a `Tree-Connect` SMB request containing the name of the disk share he wants to access. In response the server will send a tree identifier (*TID*) that the client must provide in all future protocol operations.

We are now ready to start issuing SMB protocol messages representing the file system operations we want to perform.

4.2 Encoding and Decoding SMB PDUs

SMB messages are encoded and decoded using the Erlang bit syntax. Each SMB message consists of a 32-byte SMB header, possibly followed by some associated data. In the example below, which shows the encoding of an SMB message, we use an Erlang record to hold the SMB header information. The actual data has already been encoded into Binaries:

[5]Protocol Data Unit, a single protocol message
[6]other names are: protocol variant or protocol dialect
[7]http://www.unicode.org/

```
enc_smb(Pdu) ->
  <<16#FF, $S, $M, $B,           % SMB header
   Pdu#smbpdu.cmd,               % SMB protocol OP
   Pdu#smbpdu.eclass,            % Error class
   0,                            % zero (not used)
   (Pdu#smbpdu.ecode):16/little, % Error code
   Pdu#smbpdu.flags,             % Flags
   (Pdu#smbpdu.flags2):16/little,% More Flags
   0:12/unit:8,                  % Pad (12 bytes)
   (Pdu#smbpdu.tid):16/little,   % Tree ID
   (Pdu#smbpdu.pid):16/little,   % Process ID
   (Pdu#smbpdu.uid):16/little,   % User ID
   (Pdu#smbpdu.mid):16/little,   % Multiplex ID
   Pdu#smbpdu.wc,                % # of param.words
   (Pdu#smbpdu.wp)/binary,       % param.words
   (Pdu#smbpdu.bc):16/little,    % # of bytes
   (Pdu#smbpdu.bf)/binary>>.     % The bytes
```

As can be seen, the bit syntax makes the encoding trivial. Note in particular how we creates short integers in little endian format by using the `little` type specifier. This kind of construct is used for both encoding and decoding integers with a unit size larger than one byte.

The following example shows how an `SMB-Close` PDU is created:

```
smb_close_file_pdu(InReq) ->
  {Wc,Wp} = wp_close_file(InReq#smbpdu.fid),
  Rec = #smbpdu{cmd = ?SMB_CLOSE,
                pid = InReq#smbpdu.pid,
                uid = InReq#smbpdu.uid,
                tid = InReq#smbpdu.tid,
                flags2 = InReq#smbpdu.flags2,
                wc = Wc,
                wp = Wp},
  {Rec, enc_smb(Rec)}.

wp_close_file(Fid) ->
  {3,                % # of param.words
   <<Fid:16/little,  % File handle
     0:32/little>>}. % Time of last write
                     % (0 == set by server)
```

The resulting PDU contain the file handle (the `fid` of `InReq`) to be closed. The input parameter `InReq` holds information that we have gathered in previous SMB requests. Note the use of the previously defined `enc_smb` function.

4.3 Unicode

An SMB session may negotiate to use Unicode. In this case strings are encoded in UTF-16 format, which is an encoding scheme defined by the Unicode standard [2]. The Unicode characters used by the Windows server are serialized into this format using little-endian byte order.

The Nortel SSL-VPN product supports internationalization of the Web portal presented to the users so that different character sets (languages) can be used for different users. This requires that we convert strings received in SMB's UTF-16LE format into the appropriate character set for the user.

Fortunately, there is a program on Linux called *iconv* that does this conversion. Because calling iconv is a frequent operation an Erlang linked-in driver was developed. The example below shows code to convert from UTF16-LE into another character set:

```
ucs2_to_charset(Str, Cset) ->
  case iconv:open(Cset, ?CSET_UTF16LE) of
    {ok, Cd} ->
      case iconv:conv(Cd, Str) of
        {ok, Res}       -> iconv:close(Cd), Res;
        {error, Reason} -> mk_unconv_str(Str, Cset)
      end;
    {error, Reason} ->
      mk_unconv_str(Str, Cset)
  end.

mk_unconv_str(Str, Cset) ->
  Qstr = string:copies("?", length(Str) div 2),
  ascii_to_charset(Qstr, Cset).
```

Note that in this example if the conversion fails we return a string containing question marks.

4.4 Authentication

While developing the Erlang implementation of SMB it was convenient to begin by negotiating the most basic SMB dialect: *"PC NETWORK PROGRAM 1.0"*. This way the development of password-encryption could be deferred until after most of SMB was working and then developed and tested separately.

Either of two encryption methods can be used by SMB depending on which dialect is negotiated [6]. In both methods the server sends a *challenge* and the client computes its *response* by encrypting the challenge with a *session key* derived from the user's password. Encryption is performed with the DES block-mode encryption function. The difference between the two methods lies in the way the session key is computed. In the first method the *LM Session Key* is computed using the DES function. In the other the *NT session key* is computed by taking the MD4 hash of the password's UTF-16LE representation. To cater for the second method, an Erlang linked-in driver was developed for MD4 hashing.

The DES encryption is done by encrypting the challenge, which is always 8 bytes long, by the session key. The session key is always 21 bytes long but needs to be chopped into blocks of 7 bytes for encryption. The example below shows how we use the Erlang crypto library to implement the encryption function[8]:

```
ex(<<K0:7/binary,K1:7/binary,K2:7/binary>>,Data) ->
```

[8]The function s2k/1 converts a 7 character string (7 bytes, 8 bits per byte, total 56 bits) to a DES key (8 bytes, 7 bits per byte, total 56 bits).

```
  concat_binary([e(K0,Data),e(K1,Data),e(K2,Data)]);

e(K,D) ->
  crypto:des_cbc_encrypt(s2k(K), null_vector(), D).

null_vector() -> <<0,0,0,0,0,0,0,0>>.
```

It is worth noting that in the CIFS-1.0 Technical Reference [6] a bit-swap operation named *swab* is included in the definition of how the *NT Session Key* is computed. After some trial and error we have concluded that no such operation is used in reality.

5. EVALUATION

To evaluate the Erlang SMB implementation compared with Samba we are interested in three things: performance (speed), memory consumption, and the size of release packages (disk space consumption).

As explained earlier, the purpose of this work was not to increase speed. We did however want to ensure that performance didn't decrease with the new implementation. The lab equipment, representing the SSL-VPN hardware, contained one 500 MHz Pentium-III processor with 500 MB of memory. This particular machine did not have an SSL-acceleration card, so SSL processing was turned off. In all test cases we used 4 physical machines simulating a variable number of users. Each simulated user logs into our Web portal and then downloads one or more files from an SMB backend server through the portal.

We will call the old implementation that is using Samba for *OLD*, and the new Erlang implementation for *NEW*.

5.1 Test case 1

Each simulated user logs in once and then downloads the same 20 MB file 10 times. One backend machine running Samba on FreeBSD was used as the SMB server. Each line shows the number of users (*Users*), amount of used memory in kB (*Memory*), size in MB of the Erlang beam process (*Beam*), the CPU load in percent of user/system/idle load (*U/S/I*), and finally the throughput per user. All values are averages taken from several runs of the test case.

Users	Memory	Beam	U/S/I	Mbit/s
4	105	11	26/68/6	7.0
8	120	14	24/74/2	3.5
16	160	20	22/77/1	1.8
32	250	35	20/80/0	0.9
64	-	-	-	-

Table 1: Test case 1: OLD

We can see from Table 1 that the memory usage grows until the system starts to misbehave. At 64 users the SSL-VPN box ran out of memory and the clients couldn't connect to it anymore. The heavy memory use can be explained by the fact that each user accessing an SMB file server will start a port program to interface Samba. This will cause one Linux process to be created for each user, consuming about 1.4 MB of memory per process. Still, for 64 users, this didn't seem to sum up correctly. A closer look revealed that the test program failed to logout the users from the SSL-VPN portal. As a result, port programs from earlier test runs were being kept until the users idle timeout triggered. We choose to keep the first measurments since the scenario of users not doing a logout probably is very close to the reality. Also, later tests showed that the throughput figures were not affected by this.

Three Linux processes are involved in delivering the data to a client: the Samba port program, the Erlang beam process, and the HTTP proxy process. These processes communicate over Unix pipes, which may explain the increasing system load figures.

The total throughput of the SSL-VPN box seem to be pretty constant, around 28 Mbit/s.

Users	Memory	Beam	U/S/I	Mbit/s
4	104	11	35/60/5	7.2
8	112	14	34/65/1	3.8
16	123	21	30/70/0	2.1
32	150	39	30/70/0	1.0
64	212	101	28/72/0	0.5
128	240	117	28/72/0	-

Table 2: Test case 1: NEW

The *NEW* implementation behaved much better. Table 2 shows that we now could serve 64 users without any problem. At 128 users however, the SMB backend server failed, so no throughput figure could therefore be obtained.

The throughput is almost identical between the two system, with a slight edge for the *NEW* implementation..

We also tried to native compile the Erlang SMB module, but that did not affect any of the measurements.

5.2 Test case 2

This test case compares the raw performance between the *smbclient* program and the corresponding Erlang program. In this test we authenticate ourselves to a Windows 2000 server and download a 20 MB file to our local disk. The smbclient command was measured like this:

```
time smbclient <servicename> <passwd> -U <user> \
               -c 'get 20M.file'
```

The corresponding Erlang program was measured with the *timer:tc/3* command. The startup time of the Erlang emulator is not included in the measurment.

*	Mbyte/sec
Erlang/SMB	3.3
smbclient	4.8

Table 3: Test case 2

In this case the smbclient command is some 30% faster. The performance figures are not very fast. We will not discuss the possible reasons for this here. However, note that for all

measurements we have used default installations of Windows and FreeBSD/Samba servers, without doing any attempt of tuning them. In general though, we have seen much better performance figures from combinations of FreeBSD/Linux and Samba, than from our Windows machines.

5.3 Size

The size of our original system had exceeded the available disk space on the smallest of our supported target hardware platforms. The new solution decreases the size with 3.4 MB, which gives us roughly 3 MB for future expansion. [9]

6. CONCLUSIONS

The throughput figures for the SSL-VPN target machine show a somewhat surprising similarity between the two implementations. The 30% better throughput of Samba, as shown in Table 3, disappears when measuring the performance of the whole system. Instead we see a better overall behaviour from the Erlang implementation when the system is under heavy load.

Thus, we have shown that with a suitable system architecture using Erlang can have a positive effect on overall system performance.

During the work of integrating the Erlang-SMB implementation into the SSL-VPN software it became clear that one major benefit is in character-set handling. The new solution makes it easy to convert e.g filenames in a file listing into the character set that the client Web browser expects.

The Erlang-SMB implementation will be introduced in the next major release of Nortel's SSL-VPN products.

7. ACKNOWLEDGMENTS

I want to thank those that helped out answering question, commenting the work or wrote software I made use of. Especially, I would like to thank Luke Gorrie who helped me review the paper and correct my writing. I also got valuable comments from Per Hedeland and Johan Bevemyr.

A big thanks must also go to the people behind Samba and Ethereal. Two fine pieces of software.

Magnus Johanson wrote the test tool used for the measurements in Table 1 and 2. Peter Högfeldt helped me by explaining how the Erlang crypto library should be called for this particular kind of DES encryption. Sebastian Strollo showed me the *bitrev* example.

The esmb code can be found in the Jungerl collection at Sourceforge [10].

8. REFERENCES

[1] R. Eckstein, D. Collier-Brown, and P. Kelley. *Using Samba*. O'Reilly Associates Inc, Sebastpol, CA, 2000.

[2] R. Gilliam. *Unicode Demystified*. Addison Wesley, Boston, MA, 2002.

[3] NetBIOS-Working-Group. Protocol standard for a netbios service on a tcp/udp transport: Concepts and methods. Technical Report RFC1001, IETF, Mar. 1987.

[4] NetBIOS-Working-Group. Protocol standard for a netbios service on a tcp/udp transport: Detailed specifications. Technical Report RFC1002, IETF, Mar. 1987.

[5] R. Sharpe, T. Potter, and J. Morris. *Using Samba*. QUE Publishing Company, Indianapolis, Indiana, 2000.

[6] SNIA. Common internet file system (cifs), technical reference. Technical Report 1.0, Storage Networking Industry Association (SNIA), Jan. 2002.

[9] NB: This size restriction is irrelevant for our more recent hardware platforms

[10] http://sourceforge.net/projects/jungerl/