Protocol programming in Erlang using binaries

Claes Wikström Tony Rogvall*
Computer Science Laboratory
Ericsson Telecom AB

29 Sep 1998

1 Introduction

This text is an introduction to the binary data object found in an experimental version of ERLANG. It is also a general purpose tutorial on protocol programming in ERLANG in general.

Binaries are basically a sequence of octets. The operations we can perform on binary objects as well as the internal representation of binaries are primarily intended to be used as a means for efficient as well as beautiful protocol implementations. However a wide variety of applications can take advantage of binaries. In particular all applications that perform extensive IO, such as disk IO, GUI systems, or networking TCP/IP IO.

2 Background

Implementing protocols correctly is hard. Implementing them efficiently is even harder. In particular if the protocol implementation is supposed to be a part of a greater whole, i.e. a very large system application, it is hard. The protocol must be implemented in a flexible manner which fits into the general model of IO of the larger application. If the protocol implementation is completely stand alone from the application which uses the protocol, buffer management suffers since then typically buffers must be copied from the protocol implementation to the application utilizing the protocol and vice versa. In the extreme, buffers are copied between protocol layers, which hurts execution speed bad.

We believe that the binary data object implementation we propose in this document is well suited for protocol implementations. In particular in combination with the other facilities previously found in the ERLANG programming

*email: {klacke,tony}@erix.ericsson.se

language we have a set of tools which makes it remarkably easy to implement protocols.

The different protocol implementations found in this document have been chosen for their reputation, not because any of them is considered to be particularly hard to implement. The advantage of this is that all readers are familiar with the protocols at hand.

2.1 History

The binary data object saw the light of day in the Erlang 3.3 release approximately 1992. The first application for binaries was to provide an efficient container for object code. Object code was read from a file and loaded directly into the system. This work was done in the code loading BIFs themselves and this was deemed inappropriate for a variety of reasons, mainly portability and flexibility. Thus the binary data object was introduced and the code loading BIFs were changed to take a binary object as input parameter instead of a filename.

Later on, the binary object was used to carry messages in distributed Erlang. Once the binaries were introduces into the system ERLANG users started to use binaries for all kinds of different purposes. In particular they have been used as IO buffers in various protocol implementations. The original binary object was fairly suitable for this but not perfect. We have now tried to remedy this situation by both providing a new internal implementation of binaries as well as providing means at the language level to efficiently build and match binaries.

3 Introduction

Binaries can be constructed and pulled a part by means of matching in a way similar to regular ERLANG lists. New binary objects are created as:

```
< Part1:Size1/Modifier1 , Part2:Size2/Modifier2 ..... >
```

The `Part` is an erlang value we want to use for the construction of the binary, the `Size` part is the number of bits (or sometimes bytes) we want to use from the `Part` value and finally the `Modifier` is a dash separated sequence of type modifiers.

This is possibly best illustrated with a simple example. For example the expression

```
<777:16/integer>.
```

creates a 16 bit (two bytes) binary object consisting of the two least significant bytes from the integer `777`. A wide variety of types, and modifiers are available.

Say we have some bit pattern bound to an integer value as:

```
X = 16#abcd,
```

Now we want to construct a four byte binary which has the equivalent contents as the list:

```
int32_to_list(X) ->
   [(X bsr 24) band 255, (X bsr 16) band 255,
    (X bsr  8) band 255, X band 255].
```

Why would we want to do such an operation in the first place ? Say we want to write an ERLANG term, represented as a binary object in the external term format to a file. Furthermore we want to prepend the size of the term in the file as in:

```
B = term_to_binary(SomeTerm),
write(FileDesc, [int32_to_list(size(B)) , B]).
```

I.e, we want to serialize an integer in order to use it in an output operation. This operation is directly supported in the language through the binary constructor syntax. The function `int32_to_list/1` returned a list, it is better to directly return a binary.

```
int32_to_bin(X) -> <X:32/unsigned-integer>.
```

.

Thus the previous sequence where we write a term to a file would become.

```
B = term_to_binary(SomeTerm),
write(FileDesc, <(size(B)):32/unsigned | B>).
```

Integers come in different varieties. Size, sign and endianism [1] may vary.

The size field in a binary expression indicates how many bits of the value we want to use. For example: `<255:4/integer, 255:4/integer>` evaluates to a one byte binary object holding the value `255`. We can try this out at the shell prompt:

```
1> <255:4/integer, 255:4/integer>.
<255>
```

In a manner similar to how we build binaries we can match a binary object when we need to pull a binary apart. We use a syntax which is equivalent to the construction syntax when we match binaries. So for example to pull apart some of the objects we constructed in the previous section we can write:

[1] The term "endian" can be traced to Jonathan Swift's novel "Gulliver's Travels". In one of Gulliver's adventures, he encounters an island whose inhabitants bitterly argue over the correct way to open soft-boiled eggs - the little end or the big end. Little endians and big endians are each convinced that their method is the only correct method for opening an egg.

```
X = <255:4/integer, 255:4/integer>,
<Hi:4/integer, Lo:4/integer> = X,
{Hi, Lo}.
```

The above code first constructs an 8-bit binary and then pulls it apart.

It is also possible to mix the match on different types. We wrote a term to a file by means of the call

```
write(FileDesc, <(size(B)):32/unsigned | B>).
```

.

Similarly we can pull the same binary apart as

```
case Binary of
   <X:32/unsigned-integer|Btail> when size(Btail) == X ->
      ......
```

The above construction is particularly common where we have a `size` field which determines the size of a some later fields. When we match such values it is very convenient to be able to use a variable introduced in the actual match patter as a size indicator for objects in the same pattern. This is indeed possible and the above code could be written in a better way as:

```
case Binary of
   <X:32/unsigned-integer, BValue:X/binary | Tail>
      .....
```

Note how the `X` variable is introduced first in the pattern and then later on used in the same pattern.

4 Types, Compiler directives and default values

Many different erlang types can be used to match and construct binaries. In this section we list all erlang types that are supported for binary construction and matching as well how the different modifiers and size specifications apply to the different types.

4.1 Integers

4.1.1 Constructing

The size of a constructed integer is not constrained by anything but the size of the largest integer that is possible to represent.

The following list of type modifiers are applicable to integers.

- `integer` Indicates that we want to pack an integer.

- `little` The integer is packed in little endian byte order.

  Example: `<4042:16/unsigned-integer-big>` produces a 2 byte binary containing the integer `4042` represented in big endian byte order.

- `big` The integer is packed in big endian byte order. The `little` and `big` type modifiers are only applicable when the size indicator is a multiple of eight.

- `signed, unsigned` These type modifiers are allowed but don't mean anything when constructing a binary.

4.1.2 Matching

When we match integers from a binary we have the following valid type modifiers.

- `integer` Indicates that we want to match an integer.

- `little` The integer in the binary buffer is considered to be packed in little endian format.

- `big` The integer in the binary buffer is considered to be packed in big endian format.

- `signed` The integer in the binary buffer is considered to be represented with a 2-complement sign extension.

  Example:

  ```
  B = list_to_binary([255,255]),
  <X:16/signed-integer> = B,
  ```

  will bind the variable `X` to `-1`.

- `unsigned` The integer in the binary buffer is considered to an unsigned integer. Considering the same example as above with:

  ```
  B = list_to_binary([255,255]),
  <X:16/unsigned-integer> = B,
  ```

  will bind `X` to the value `65535`

4.2 Characters

The same modifiers that apply to integers apply to characters as well except that the modifier `integer` must be replaced with the modifier `char`.

4.3 Floating point numbers

Construction and matching of floating point numbers is supported. Floating point numbers are constructed and matched according to the IEEE 754 floating point standard. A float is considered to be packed in either 32 or 64 bits. The following type modifiers are applicable to floats.

- `float` The type is float.
- `big` The float is packed in big endian order.
- `small` The float is packed in small endian order.

For example the following code produces a 8-byte binary

```
F = 3.14,
Bin = <F:64/float-little>,
```

and if we want to match such a binary we can extract the packed float as in:

```
<F2:64/float-little> = Bin.
```

Note that the two floats `F1` and `F2` are not necessarily equal since the conversion process may change the floating point value with a small fraction[2].

4.4 Tuples

If a sequence of equally sized and typed values need to be processed we can either pack the values from or to a tuple.

For example if we have an 8-byte binary `Bin` and we want to pull the binary apart in four parts each part consisting of 2 bytes, where each sequence of two bytes is considered to be a 16 bit integer we can unpack `Bin` as:

```
<Tup:4/tuple-integer:16> = Bin,
```

Where `Tup` will be bound to an arity four tuple.

The size indicator `4` indicates how large we want the tuple to be. The new type modifier `integer:16` indicates how many bits shall be used for each element in the newly constructed tuple.

This syntax provides an efficient way of processing several values in one sweep. Whenever the `tuple` type modifier is present in the type modifier list, the remaining type modifiers apply to each individual element that is processed and packed in the tuple.

Similarly if we have a tuple of equally typed values and want to pack the sequence of elements in a binary we can to that with the `tuple` type modifier.

Example:

[2] It is generally considered to be poor programming practice to compare floating point numbers for exact equality

```
Tup = {1.23, 5.66, 9.00, 6.87654},
<Tup:4/tuple-float:64-little>
```

Will take the values in the tuple `Tup` from left to right and pack them in a binary object.

4.5 Binaries

It is possible to extract sub binaries from an input binary by explicitly providing a type modifier `binary` The size field in this case applies to number of bytes instead of bits as is the case with i.e integers. Example:

```
<_:10/binary, X:10/binary |_> = Bin,
```

will strip off the first 10 bytes from the binary `Bin` and bind the variable `X` to the next 10 bytes in the `Bin` buffer.

It is also possible to concatenate binaries with the same syntax. For example if we have a binary object `B` and want to build a new binary object `B2` with the size of `B` prepended we can evaluate:

```
Size = size(B),
<Size:32/unsigned-integer, B:Size/binary>.
```

or using a tail expression as described in the section about the binary tail:

```
Size = size(B),
<Size:32/unsigned-integer | B>
```

A function that concatenates a list of binaries can be written as:

```
lists:foldl(fun(B, Ack) ->
              <B/binary | Ack>
            end, <>, ListOfBinaries)
```

If the size field is ommited in an expression with the `binary` type modifier, the default value is the the size of the entire binary. Thus the expression `<B/binary?` is eqivalent with the more awkward `<B:(size(B))/binary>`.

4.6 Lists

Exactly the same technique and syntax we used in the previous section with tuples can be used with lists. For example to take the first 100 characters of a binary we can call:

```
<Str:100/list-char:8> = Bin,
```

The resulting variable `Str` will be bound to a list of 100 characters.

The ability to combine binaries and char-list is often useful.

4.7 The Tail

Whenever a tail is provided in a binary pattern, either in a matching left hand expression or in a right hand construction expression, the tail value is considered to be a binary object. For example in the match expression:

```
<X:8/char, Z:32/integer | Tail> = Bin,
```

The `Tail` value will be the remainder of the input binary `Bin` when first 5 bytes have been consumed by the `X` and `Z` variables. Thus the above expression is equivalent to

```
Sz = size(Bin),
<X:8/char, Z:32/integer, B2:Sz/binary> = Bin,
```

The same principle applies when we construct binaries, if a tail is given in a righthandside construct expression, it is considered to be a binary tail.

```
<X:8/char, Y:8/char | Tail>
```

Will prepend the two characters `X` and `Y` on the head of `Tail`.

4.8 System endianism

Sometimes it can be convenient to be able to produce an an array of integers formated in the endianism of the underlying machine. We have the following bit types predefined:

- `sys_int` The endianism and size of a system `int`
- `sys_short`
- `sys_long`
- `sys_char`
- `sys_float`
- `sys_souble`

If we use these types to produce abinary object as in:

```
B = <77/sys_int, 88/sys_int>,
```

and then send the resulting binary to an Erlang port as in:

```
Port ! {self(), {command, B}},
```

The c program at the other end of the port can read the data as:

```
int a[2];
read(fd, a, sizeof(a));
```

The above code work on both little endian and big endian machines. Of cource the c program can also use this feature to directly read binary data into a structure, thus :

```
B = <"abcd", X/sys_int, F/sys_float>,
Port ! {self(), {command, B}},
```

can be read from the port program as:

```
struct foo {
        char c[4];
        int x;
        float f;
};

struct foo f;
read(fd, &f, sizeof( struct foo));
```

4.9 Bit defaults

Two different compiler directives are applicable when we write code that manage binaries.

The compiler directive

```
-bitdefault(integer, unsigned-little)
```

will add the three modifiers `unsigned`, `integer` and `little` to each list of type modifiers in the remainder of the source code file where adding these type modifiers is applicable.

This can be useful in a number of different settings. Say that we are writing a module that only deals with unsigned 32 bit integers, in that case it can be awkward to specify the full modifier list throughout the entire file. Thus, the lazy programmer can save keystrokes by specifying

```
-bitdefault(integer, size:32-unsigned-little)
```

Here we added a size modifier saying that the default integer uses 32 bits.

Similarly we can indicate the default type modifiers for floats as in:

```
-bitdefault(float, size:32-big).
```

where we say that the default floating point value is a 32 bit float in big endian byte order.

Another situation is when we want to write code that is independent on endianism of the integers. In that case we can have a single line at the top of the file that specifies endianism.

```
-bitdefault(little).
```

If no bit default is given as a compiler directive and no type modifier is specified in the code, a bit pattern still has a meaning since almost everything has a default value. The same rules apply regardless of whether the pattern appear in a match or as a constructor. We have the following default values:

- `<V>` is equivalent with `<V:8/char>`
- `<V/integer> is equivalent with \verb`¡V:32/integer-signed-big¿+
- `<V/float> is equivalent with \verb`¡V:32/float-big¿+
- `<V/binary> is equivalent with \verb`Sz = size(V), ¡V:Sz/binary¿+
- `<V:Size>` is equivalent with `<V:Size/signed-integer-big>`

4.10 Bit typedefs

We can specify a name for a particular sequence of type modifiers as in:

```
-bittype(uint, integer-unsigned).
```

This allows us to use the name `uint` as type modifier as in

```
<X:32/uint>.
```

or if we want to indicate the size we define:

```
-bittype(uint, integer:32-unsigned).
```

4.11 Alignment

A sequence of match or construct values must always add up to a multiple of 8. The alignment requirements for binaries are very low and basically the only requirement is that the resulting binary in a build expression is byte aligned.

It is for example possible to produce a binary with a floating point inserted at non byte aligned positions in the output binary as in:

```
<X:1/uint, Pi:32/float, Y:7/uint>.
```

However

```
<X:9/uint>.
```

Will produce a runtime error.

4.12 The empty binary

The empty binary which has size 0 is specified as <>.

4.13 Bit Groups

......

4.14 Variable field size

A very common situation in many protocols is to let the Protocol Data Unit (PDU) contain a size specifier which is indicating the size of some later field in the PDU. As a very simple example the Erlang Port mechanism typically prepend a two byte length indicator on each message it sends to an external process that is connected to the port. When we receive such a message we can both have the cake and eat it since we are able to first bind the initial two bytes to an integer and then use that integer in the same match expression. So to receive such a message we would write code such as:

```
receive
   {data, <Size:16/uint, Data:Sz/binary>} ->
       handle_data(Data),
       .....
```

For a more complex but truly useful example, assume that we have a port and receive data from the port where each packet from the port is prepended with a two byte header indicating the size of the entire packet, then the following code receives messages from the Port and sends complete packets to the user of the port.

```
-bitdefault(unsigned-integer).

port_loop(Port, User, Ack) ->
    case Ack of
        <H:16/unsigned, Pdu:H/binary | Tail>
            User ! {pdu, Pdu},
            port_loop(Port, User, Tail);
        Other ->
            receive
                {Port, {data, Data}} ->
                    port_loop(Port, User, <Ack/binary | Data>)
            end
    end.
```

This code solves a fairly complicated problem. Assume we have a linked in driver which reads data from some input channel, in a UNIX environment, this would typically be a file descriptor. If the above code sits at the Erlang end of the port, the driver can freely read as much as it can from the file descriptor and whatever the driver reads, it simply passes on to the port. The alternative to this is to either let the driver perform two read operations, one where the first two bytes, being the length indicator are read, and then yet another read operation where the actual contents are read, or alternatively, the driver can blindly try to read as much data as it can. In this case, the driver must be prepared for all possible outcomes of the read operation. It can receive a half header, a header and a half message, a header a message and yet another half header and so on. If the PDU structure is more complicated than the one above, the c-code to perform all these operation correctly can be fairly complicated.

5 Are they strings ?

Yes, binaries can be used as builtin character strings.

Syntactic sugaring makes the expression:

```
<"funky">
```

equivalent to the more awkward expression:

```
<"funky":5/list-char:8>
```

When we use binaries for strings it is very convenient to be able to search a binary for a specific substring and split it into two different parts with the first part being the part upto the search string and the remainder the rest.

A binary match pattern may contain unbound variables with the size field being unbound too. This feature can be used to search a binary for a specific string. For example the match pattern

```
<X:Sz/binary, "cat" | Tail> = Bin,
```

Will search the binary `Bin` for the first occurence of the string `"cat"` and divide `Bin` in two parts, the part leading upto the string `"cat"`, and the part following `"cat"`. If no `"cat"` is found in `Bin`, the pattern does not match.

This feature in combination with the rest of the binary system can be used to implement efficent string manipulation functions.

Many of the string manipulation functions that have complexity $O(n)$ when strings are implemented as lists of cons cells have complexity $O(1)$ when strings are implemented as binary objects in Erlang. For example the length of a string:

```
len(S) -> size(S).
```

As well as concatenation:

```
concat(S1, S2) ->
    <S1/binary | S2>.
```

have complexity $O(1)$, whereas the `chr/2` function that returns the position of the first occurence of a char in a string:

```
chr(S, C) ->
    case S of
        <_:Size/binary, C:8/char |_> ->
            Size + 1;
        _ ->
            0
    end.
```

of cource have complexity $O(1)$.

The module `bstring` implements a number of string utility functions that work on binary strings. The module has exactly the same interface functions as the original `string` module which operates on lists of integers/characters.

Since binary strings have completely different internal representation than list based strings, they behave differently concerning memory consumption and speed. Many programs do behave much better with binary strings than list based strings. In particular programs that manipulate large amounts of data. For example a function that opens a file, partition the file into lines and returns a list of `{LineNumber, Line}` tuples:

```
file_lines(File) ->
    case file:open(File, [read,raw,binary]) of
        {ok, Fd} ->
            Res = file_lines(Fd, <>, 1, []),
            file:close(Fd),
            Res;
        Err ->
            Err
    end.

file_lines(Fd, Old, Lno, Ack) ->
    case file:read(Fd, 2000) of
        {ok, B} ->
            {A2, T2, L2} = split_chunk(<Old/binary | B>, Ack, Lno),
            file_lines(Fd, T2, L2, A2);
        eof ->
            Ack;
        Err ->
            Err
    end.
```

```
split_chunk(<Line/binary, $\n/char | Tail>, Ack, Lno) ->
    split_chunk(Tail, [{Lno, Line} |Ack], Lno+1);
split_chunk(Other, Ack, Lno) ->
    {Ack, Other, Lno}.
```

This code is very efficient and works well even on very large input sets, unfortunately there are a number of different operation commonly performed on strings that are considerably less efficient. For example reversing a binary string.

```
breverse(S) ->
    breverse(S, <>).
breverse(<H/char|Tail>, Ack) ->
    breverse(Tail, <H/Char|Ack>);
breverse(<>, Ack) ->
    Ack.
```

The cons cell is a superior data structure in this case.

6 Classical internet style text based protocols

Many typical internet protocols are text and newline based. That means that the participating computers send readable newline terminated text strings to each other. Examples of such protocols are SMTP for electronic mail, NNTP for news postings and HTTP for the world wide web.

The binary strings of ERLANG are well suited for the implementation of such protocols. In particular the ability to search for substrings using the construct with a variable length binary followed by a bound expression, for example:

```
<B:Sz/binary, $:/char |_> = InputString
```

extracts the intial chars of the `InputString` upto and excluding the first occurrence of a `:` character.

For example a typical http request to a http server looks like

```
GET /index.html HTTP/1.0
User-Agent: Mozilla/4.04 [en] (X11; I; SunOS 5.5.1 sun4u)
Pragma: no-cache
Host: gin.du.etx.ericsson.se:5999
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Via: 1.0 proxy.du.etx.ericsson.se:82 (Squid/1.1.20)
```

```
X-Forwarded-For: unknown
Cache-control: Max-age=2592000
```

Each line is terminated with a pair of carriage return / newline except the last line which has two such pairs. An efficent parser to unpack such MIME like headers may look like:

```
parse(<"GET ", What/binary, "\r\n" | Tail>) ->
    {Fields, Contents}  = parse_tail(Tail, empty, []).
    {What, Fields, Contents}.

parse_tail(<B/binary, "\r\n\r\n" | Cont>, A) ->
    {Cont, Ack};
parse_tail(<B/binary, "\r\n" | Tail>, A) ->
     parse_tail(Tail, [B|A]).
```

This code uses substrings extensively. Http is more complicated than this with `POST` requests etc, but the above code can be augmented in a straightforward way to handle the entire http protocol.

The above code can form the heart of a microscopic http server.

As another example we could have the code to extract the "From:" field from SMTP emails. This could be done as:

```
from_header(<_/binary, "\nFrom:", Sender:Sz/binary, "\r\n" |_>) ->
    Sender.
```

6.1 Case insensitivity

Many of the text based internet protocols are case insensitive. To examplify, the SMTP mail protocol always start a session with a line of text:

```
HELO <domain> CRNL
```

where the client tells the SMTP server which his fully qualified host name is. The initial "HELO" string is case insensitive, so not only the string "HELO" is accepted, but also the string "HeLo". This can be addressed with the binary type modifier `icase`. So to match the HELO mesage in an SMTP server we would write:

```
case Message of
   <"HELO"/-icase, Domain/binary, "\r\n" |_>
      handle_domain(Domain);
```

6.2 An SMTP client

In this section we shall present a small SMTP client that constructs an email, talks the SNMP protocol with an SMTP server and sends away the email. This is not a complete implementation of SMTP, albeit a fully working implementation: We start off with some definitions and support functions:

```
-define(READY,   "220").
-define(CLOSE,   "221").
-define(OKAY,    "250").
-define(START,   "354").
-define(NOAVAIL, "550").
-define(CRNL,    "\r\n").

-record(rec, {from,     %% The from address, who am i
              hname,    %% our hostname
              smtp}).   %% name of the smtp server
```

And a function to initialize the record:

```
getrec() ->
    #rec{from = "<" ++ user() ++ "@" ++ hname() ++ ">",
         hname = hname(),
         smtp = smtp()}.

user() ->
    case os:getenv("USER") of
        false ->  "luzer";
        User -> User
    end.

hname() ->
    case inet:gethostname()  of
        {ok, H} ->
            case inet:gethostbyname(H) ofy
                {ok, HE} ->
                    HE#hostent.h_name;
                {error, _} ->
                    "localhost"
            end;
        {error, _} ->
            "localhost"
```

```
    end.

smtp() ->
    case os:getenv("SMTP") of
        false -> "localhost";
        H -> H
    end.
```

Then we have the two top level functions that are the API:

```
send(M, To) ->
    send(M, To, "No subject").
send(M, To, Subject) when binary(M), list(To), list(Subject) ->
    R = getrec(),
    To2 = map(fun([$<|T]) -> [$<|T];
                 (X) -> "<" ++  X ++ ">"
              end, To),
    M2 = fix_mess(M, To2, R, Subject),
    case gen_tcp:connect(R#rec.smtp, 25, [binary, {packet, 0}]) of
        {ok, S} ->
            init(S, M2, To2, R);
        Other ->
            Other
    end;
send(_,_,_) ->
    {error, "Bad input type(s)"}.
```

The message is a binary object and it is massaged by the `fix_mess/4` function:

```
%% We need to add the headers to the message body
fix_mess(M, To, R, Subj) ->
    S = <"Subject: ", Subj/char-list, ?CRNL>,
    F = <"From: ", (R#rec.from)/char-list, ?CRNL>,
    To2 = to(To, <>),
    T = <"To: ", To2/binary, ?CRNL>,
    <F/binary, T/binary, S/binary,  ?CRNL, M/binary, ?CRNL, ".", ?CRNL>.


to([Last], Ack) ->
```

```
    <Ack/binary , Last/char-list>;
to([T|More], Ack) ->
    to(More, <T/char-list , "," | Ack>).
```

If the message is correctly massaged (according to the rules in SMTP, we try to connect to the SMTP server and enter the `init` state.

```
init(S, M, To, R) ->
    receive
        {tcp, S, <?READY | _>} ->
            ready(S, M, To, R)
    after 5000 ->
            {error, timeout}
    end.
```

If we get a `READY` message from the SMTP server we enter the `ready` state and wait for an `OKAY`.

```
ready(S, M, To, R) ->
    gen_tcp:send(S, <"HELO ", (R#rec.hname)/char-list, " \r\n">),
    wait_okay(S, M, To, R).

wait_okay(S, M, To, R) ->
    receive
        {tcp, S, <?OKAY |_>} ->
            gen_tcp:send(S, <"MAIL FROM: ", (R#rec.from)/char-list, ?CRNL>),
            receive
                {tcp, S, <?OKAY |_>} ->
                    send_messages(S, M, To, R, [], []);
                {tcp, S, <Err:3/char-list |_>} ->
                    err(S, Err)
            after 5000 ->
                    err(S, timeout)
            end;
        {tcp, S, Other} ->
            <Err:3/char-list | _> = Other,
            err(S, Err)
    after 5000 ->
            {error, timeout}
    end.
```

If all is well so far, we start to transmit message to the list of recepients:

```
send_messages(S, M, [], R, Ok, Nok) ->
    gen_tcp:send(S, <"QUIT \r\n">),
    receive
        {tcp, S, <?CLOSE |_>} ->
            gen_tcp:close(S),
            {ok, Ok, Nok};
        {tcp, S, <Err:3/char-list |_>} ->
            err(S, Err)
    after 5000 ->
            err(S, timeout)
    end;

send_messages(S, M, [To|More], R, Ok, Nok) ->
    gen_tcp:send(S, <"RCPT TO:",  To/char-list, ?CRNL>),
    receive
        {tcp, S, <?NOAVAIL |_>} ->
            send_messages(S, M, More, R, Ok, [To | Nok]);
        {tcp, S, <?OKAY|_>} ->
            gen_tcp:send(S, <"DATA ", ?CRNL>),
            receive
                {tcp, S, <?START | _>} ->
                    gen_tcp:send(S, M),
                    receive
                        {tcp, S, <?OKAY|_>} ->
                            send_messages(S, M, More, R, [To|Ok], Nok);
                        {tcp, S, <Err:3/char-list|_>} ->
                            send_messages(S, M, More, R, Ok, [To|Nok])
                    after 10000 ->
                            err(S, timeout)
                    end;
                {tcp, S, <Err:3/char-list|_>} ->
                    send_messages(S, M, More, R, [To|Ok], Nok)
            after 10000 ->
                    err(S, timeout)
            end;
        {tcp, S, <Err:3/char-list|_>} ->
            send_messages(S, M, More, R, Ok, [To | Nok])
    after 10000 ->
            err(S, timeout)
    end.

err(S, Reason) ->
    gen_tcp:close(S),
```

```
    {error, Reason}.
```

And that is all there is. In order to mail a mail, we merely call

```
smail:send(Mess, ["klacke@erix.ericsson.se",
                  "tony@erix.ericsson.se"],
           "Is this cool or what "),
```

7 UDP/IP

In this section we provide a microscopic implementation of the protocol suite UDP/IP. This is used to exemplify a number of useful techiques. The reader who is not familiar with the internals of the IP protocol suite is well advised to read an introductury text on the topic.

The only magic we assume here is the ability receive ethernet frames into the ERLANG application. This can be achieved in a number of ways, however they differ wildly on different operating systems. No name resolution is performed at all, all addresses are supposeed to be proper IP addresses. This is of cource a ridiculous application since there already exists a large number of highly optimized implementations of the TCP/IP protocols. It is however interesting from a educational point of view since it shows how easily protocols can be developed in ERLANG.

We have a receive loop at the bottom most layer that receives Ethernet frames, decodes them and dispatches them to either of the ARP [?] or IP [?] layers. This receive loop is aware of the local ethernet address by means of the previously mentioned magic.

8 IDLs

Many telecommunications protocols make use of ASN.1[?] to define the datatypes that are used in a protocol. Different encodings duch as BER[?] and PER[?] are typically used. In this section we show how what code an ASN.1 to ERLANGcompiler should generate utilizing the bit syntax. In particular we show how a binary string produced by the application, inserted into an ASN.1 data structure can find its way all the way down to the IO system without ever being copied.

Lets assume we have an ASN.1 datatype `Struct`:

```
Struct ::= SEQUENCE {
    b BOOLEAN,
    s OCTET STRING }
```

The equivalent ERLANGrecord would of cource be:

```
-record('Struct', {b, s}).
```

In this example we shall assume that the Basic Encoding Rules, BER is used. For a breief introduction to ASN.1 and BER [?] can be consulted.

BER is an encoding scheme which uses a Tag/Length/Value encoding. This means that every value is encoded as three parts. The tag, identifies which type is encoded, the length is the length of the encoded value, and the value is of cource the coded value itself. This is often refered to TLV encodings.

In this case we have three ASN.1 components to encode, a SEQUENCE, a BOOLEAN and an OCTET STRING.

First we need a function to encode a boolean:

```
ebool(true)  ->
    <?BOOL/8, 1:8, 1:8>;
ebool(false) ->
    <?BOOL/8, 1:8, 0:8>.
```

Here we made use a constant `BOOL` which is the tag that defines a boolean in BER code:

```
-define(BOOL,      2).
-define(OSTRING,  16).
-define(SEQUENCE, 48).
```

Furthermore we need to have a library function to encode octet strings. We have:

```
eostring(Bstr) ->
   Sz = size(Bstr),
   Bits = lensize(Sz))
   <?OSTRING:8/integer, Sz:(Bits*8)/integer-little, Bstr/binary>.
```

We need a library function `lensize/1` to calculate the number of bytes necessary to hold the BER length value. We have:

```
lensize(S) ->
   lensize(S bsr 7, 1).
lensize(0, I) -> I;
lensize(X, I) ->
   lensize(X bsr 8, I+1).
```

The function `lensize/1` is defined according to the rules for BER length encodings.

Now that we have the primitive library functions that we need we can look at the code that is generated by a proper ASN.1 to Erlang compiler to encode our `Struct` structure:

```
enc_Struct(S) ->
   B1 = ebool(S#'Struct'.b),
   B2 = eostring(S#'Struct'.s),
   Sz = size(B1) + size(B2),
   Bits = lensize(Sz) * 8,
   <?SEQUENCE:8/integer, Sz:Bits/integer,
    B1/binary, B2/binary>.
```

This code is particular interesting since it shows how a string produced at the application layer, can find its way through a BER encoder without ever being copied. The result of the `enc_Struct/1` operation may in its turn be manipulated by lower layers in the protocol stack, still without ever copying the initial string provided by the user. This can be achieved since the memory management system for the user is the same as the memory management system for the protocol layers. So here we see how the binary object implementation acts as a general purpose buffer manger for protocol implementations.

This section made a point using ASN.1 as IDL, however exactly the same reasoning applies to other IDLs, such as Corba and XDR, albeit in the case of Corba, more complicated.

9 Internals

In this chapter we describe how binaries are represented internally in the runtime system. We believe that this information is important since one of the main objectives with this experimental syntax is to achieve Erlang programs that are not only more beautiful but also faster and less memory consuming. Thus it is important for protocol implementors to be at least vaguely familiar with the internals of binaries, and thus with the characteristics of binaries.

A binary object can be represented in four different ways inside the runtime system depending on how it was created. All binaries consist of a tagged pointer to a structure on the Erlang process heap. These structure, called thing structures come in four different varieties. The user of binaries, i.e. the Erlang programmer doesn't see anything about what sort of binary a specific object is represente as internally.

At the language level all binaries are seen as consecutive series of octets.

9.1 Heap binaries

First we have the simplest variant which can be described by the c-code struct:

```
typedef struct heap_binary {
    uint32 thing_word;   /* tagged thing, with subtag and tari */
    int size;            /* number of bytes */
    uint32 data[1];      /* The data */
```

```
} HeapBin;
```

Where the `uint32` type is an unsigned 32 bit value. The `thing_word` field is used to identify the type of the structure, the `size` field is the number of bytes that this binary contains, and finally the `data` field is used to locate the beginning of the actual data.

This type of binary object is used for small binaries, for example the result of `<X:16>`. will be a tagged binary pointer pointing to a `HeapBin` structure on the regular process heap. These type of objects are garbage collected similar to tuples and all other regular ERLANG objects. Furthermore they are copied in message passing as well as when they are inserted into ets-tables.

9.2 Reference counted binaries

Secondly we represent large binary objects as a pointer to a structure:

```
typedef struct refc_bin {
    uint32 thing_word;      /* tagged thing, with subtag and tari */
    int size;               /* nuber of bytes in binary */
    RefcBin *next;          /* heap pointer to next RefcBin */
    Binary *val;            /* Pointer to refc'ed object */
    byte *bytes;            /* actual byte* pointer */
} RefcBin;
```

The principal goal of this data structure is to be able to send large binaries to other ERLANG processes without copying the actual data. Thus the `Binary*` field points to a structure:

```
typedef struct binary {
    int orig_size;         /* total length of binary */
    int refc;              /* number of references to this binary */
    char orig_bytes[1];    /* the data (char instead of byte!) */
} Binary;
```

The `Binary` structure contains the actual data and it is allocated off heap, typically by means of `malloc()`. It contains a reference counter, so whenever a `RefcBinary` object is sent in a message to another process the following things occur. First the reference counter is incremented, then a `RefcBin` structure is created on the receiving process heap. This structure is linked into a list chained list of `RefcBin` objects on that heap. This last step is important. All `RefcBins` created on a process heap are chained in a list visible to the process. When the process heap is garbage collected, the chain is traversed and any object in the chain which has not been moved by the garbage collector is first unlinked from the chain, and then followed in order to decrement the reference counter. When the reference counter is zero, the entire `Binary` structure can be released, typically by means of a call to `free()`.

The ability to send large binary objects without copying them is especially important in implementations of protocols that carry data as opposed to pure signalling protocols.

9.3 Segmented binaries

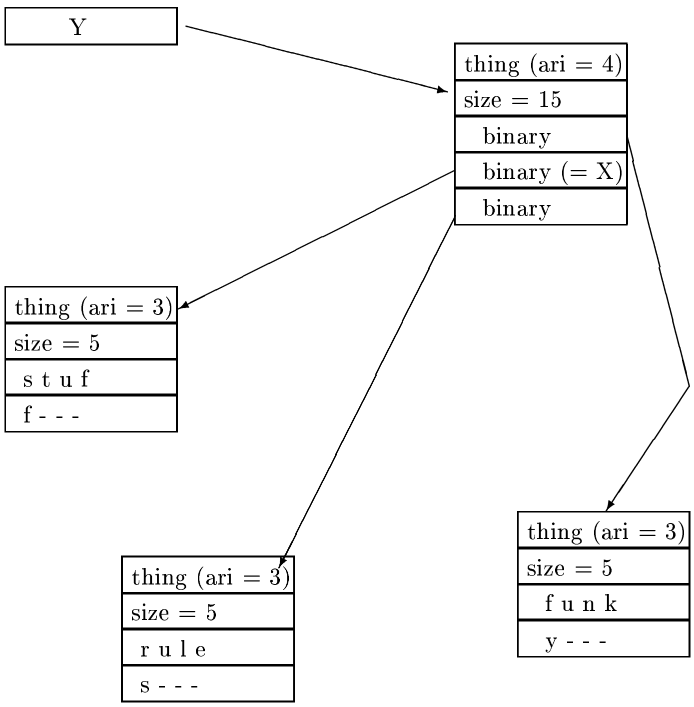
When constructing binaries by adding components as head or a trailer onto a PDU it important to be able to do that in constant time. For example:

```
X = <" stuff ">,
Y = <"funky ", X/binary, " rules">.
```

Therefore a binary object can consist of an array of other binary objects, we call this a segmented binary. A segmented binary is represented on the ERLANG heap as as structure:

```
typedef struct segm_binary {
    uint32 thing_word;   /* tagged thing, with subtag and tari */
    int size;            /* number of bytes in the total binary */
    uint32 data[1];
} SegmBin;
```

Both the `HeapBin` strucure and the `SegmBin` structure are variable sized. The number of words that any of these structures consist of can be calculated by extracting the 16 least significant bits of the `thing_word`. In the case above we would have the following pointers and data structures:


Y
thing (ari = 4)
size = 15
binary
binary (= X)
binary
thing (ari = 3)
size = 5
s t u f
f - - -
thing (ari = 3)
size = 5
r u l e
s - - -
thing (ari = 3)
size = 5
f u n k
y - - -

In the case with the "funky stuff rules" string, all data is stored on the regular ERLANG heap and thus subject to the standard copying garbage collection. When a structure like this is copied, the structure is compactified and the result will be that the varibale `Y` from above is bound to a single `HeapBinary` instead of a segmented binary.

9.4 Sub Binaries

Code that strips off a head from a binary and performs a calculatation on the head, and then basen on the outcome of that calculatation continues to process the Sub is typical for man applications. The `file_lines/1` from a previous section is an example of this. Therefore the runtime system represents such a binary in an optimized way. If we have:

```
<X:16 | Tail> = Bin.
```

the tail will be represented as a pointer to a `TailBin` structure on the heap:

```
typedef struct Sub_binary {
    uint32 thing_word;
    int size;
    int offs;
    uint32 orig;  /*original binary object to use for offset */
} SubBin;
```

This means that the tail is only represented as an offset into the original object. This is used for another interesting optimization in the execution machinery. When a match operation is initiated on a binary object, a `match_buffer` is initialized in the runtime system. If the match succeeds, a word in the match buffer is set to the value of the initial input to the match machinery. On the next call to the match machinery, this word is checked to see if it is the same word we set the last time we performed a match, if this is the case no initialization of the match buffer is neccessary at all. For example:

```
funky() ->
   X = <"abcd">,
   count_16(X).

count_16(<X:16|Tail>) ->
   1 + count_16(Tail);
count_16(<>) ->
   0.
```

In the above code, the match buffer will only be initialized once. This is on the first call to `count_16/1`. On the second call to `count_16/1` the match buffer will be both initialized as well as initialized to the correct value with regards to bit offset into the original binary object `X`.

The same data structure can also be used when a substring is extracted from an initial binary. For example:

```
sub(Str, From, To) ->
    Sz = To - From,
    case Str of
       <_:From/binary, Result:Sz/binary |_>
           Result
    end.
```

The result value `Result` will be prepresented as a `SubBinary` to the original input binary `Str`.

9.5 Binaries and IO

Most modern IO devices today support a mode of IO called gather/scatter IO. The UNIX system call `writev()` takes an output buffer which consists of an array of `iovec` structures. An `iovec` structure contains the following members:

```
caddr_t   iov_base;
int       iov_len;
```

This maps pefectly with a segmented binary. The net result of this is that the Erlang programmer can construct an output buffer as a number of small operations each operation either adds some data to the head or the tail of the buffer. Once the buffer is ready for output, the runtime system constructs an array of `iovec` structures by setting pointers, i.e. without copying any data. This array is then passed directly to the `writev()` routine.

All IO in Erlang is performed through a driver. In order for the `writev()` scenario to take place, the driver must export a `writev()` routine. How this is done is described in an appendix to this document.

Appendix

Scatter IO in a linked in driver

The scatter IO interface in linked drivers in the Erlang system ought to be described in some proper OTP documentation. Since it is not and the scatter IO mechanism is of paramount importance for the implementation of efficent protocol drivers in Erlang the mechanism is described here. ......