C open smpp-3.4 Library

This work is directed to C programmers with basic knowledge in SMPP protocol, at least in the specification and handling of sessions. Extending a little the development of the library, the scope of the same might involve a development methodology of protocols on TCP/IP.

Revision History

Revision 1.4 2012-01-02 Revised by: RT
Version 1.10, we add a new function unpack2 to resolve CMD_ID identification from buffer.
Revision 1.3 2006-10-28 Revised by: RT
Version 1.8, pending solved.
Revision 1.2 2006-06-28 Revised by: RT
Some corrections in english version.
Revision 1.1 2006-06-12 Revised by: Marcela Carbajo
Some corrections in english version.
Revision 1.0 2006-06-03 Revised by: Raúl Tremsal
Initial version.

Table of Contents

1. Introduction

The library main focus is to work in packaging and unpackaging of data structures. Independently that this implementation is about SMPP-3.4, the aim is to generate a simple way to implement any proprietary protocol on TCP.

1.1. The main objective

Any developer that attempts to enter in SMPP world, must go through Kannel software (http://www.kannel.org/). However, Kannel project is so big that it discourages anybody willing to develop a simple SMPP application.

So the library focus is to provide an implementation of SMPP-3.4 protocol but just for PDUs handling, taking as independent the managing of TCP connection and SMPP session. As we see, the library is directed to solve just the protocol issue, left to programmers the handling of the other levels in the communication

1.2. API definition

The API is defined in four functions:

- **Packed function:** In this function the first and last parameters are identifying the data structure that we want to pack. The first parameter indetifies the data structure through an integer and the last one refers to data object.

```
int smpp34_pack( uint32_t type,    /* in  */
                 uint8_t *ptrBuf,  /* out */
                 int      ptrSize, /* out */
                 int     *ptrLen,  /* out */
                 void    *tt       /* in  */  )
```

- **Unpacked function:** In this function a pointer to the buffer and its lenght is passed as parameters. The function returns a pointer to a data structure defined by a type field.

```
int smpp34_unpack( uint32_t type,   /* in  */
                   void    *tt,     /* out */
                   uint8_t *ptrBuf, /* in  */
                   int     ptrLen,  /* in  */ )
```

- **Structure dump function:** The utility of this function is to provide with a tool for to see the parameters and values of a PDU (data structure). The *in* parameters refers to data object and the *out* parameters is a string buffer where the new representation is done.

```
int smpp34_dumpPdu( uint32_t type,      /* in  */
                    uint8_t *dest,      /* out */
                    int      size_dest, /* in  */
                    void    *tt         /* in  */ )
```

- **Buffer dump function:** In this function, the *in* parameters refers to a buffer and the its lenght. This function prints in a external buffer a hexa representation of the binary source.

```
int smpp34_dumpBuf( uint8_t *dest,  /* out */
                    int      destL, /* in  */
                    uint8_t *src,   /* in  */
                    int      srcL   /* in  */ )
```

In addition to this function, the global variables that complete the API for developing are defined.

```
int smpp34_errno;
char smpp34_strerror[2048];
```

Besides, all the data structure proprietary of th SMPP protocol are defined. There is a data structure for each PDU definition. For more information about SMPP please refer to SMPP protocol specification.

```
typedef struct tlv_t tlv_t;
typedef struct dad_t dad_t;              /* used in SUBMIT_MULTI PDU      */
typedef struct udad_t udad_t;            /* used in SUBMIT_MULTI_RESP PDU */
typedef struct bind_transmitter_t bind_transmitter_t;
typedef struct bind_transmitter_resp_t bind_transmitter_resp_t;
typedef struct bind_receiver_t bind_receiver_t;
typedef struct bind_receiver_resp_t bind_receiver_resp_t;
typedef struct bind_transceiver_t bind_transceiver_t;
typedef struct bind_transceiver_resp_t bind_transceiver_resp_t;
typedef struct outbind_t outbind_t;
typedef struct unbind_t unbind_t;
typedef struct unbind_resp_t unbind_resp_t;
typedef struct generic_nack_t generic_nack_t;
typedef struct submit_sm_t submit_sm_t;
typedef struct submit_sm_resp_t submit_sm_resp_t;
typedef struct submit_multi_t submit_multi_t;
typedef struct submit_multi_resp_t submit_multi_resp_t;
typedef struct deliver_sm_t deliver_sm_t;
typedef struct deliver_sm_resp_t deliver_sm_resp_t;
typedef struct data_sm_t data_sm_t;
typedef struct data_sm_resp_t data_sm_resp_t;
typedef struct query_sm_t query_sm_t;
typedef struct query_sm_resp_t query_sm_resp_t;
typedef struct cancel_sm_t cancel_sm_t;
typedef struct cancel_sm_resp_t cancel_sm_resp_t;
typedef struct replace_sm_t replace_sm_t;
typedef struct replace_sm_resp_t replace_sm_resp_t;
typedef struct enquire_link_t enquire_link_t;
typedef struct alert_notification_t alert_notification_t;
```

The description of each data structure is detailed in SMPP-3.4 document (Short Message Peer to Peer Protocol Specification v3.4). As another settled goal, the implementation of each structure doesn't differ at all from the protocol specification (with exception of the optional parameters, where a list of dynamic data are used).

So if you want to handle optional parameters in SMPP-3.4 protocol, you must use two functions:

```
int build_tlv( tlv_t **dest, tlv_t *source );
int destroy_tlv( tlv_t *sourceList );
```

If you want to handle destination address dinamic list in SUBMIT_MULTI and SUBMIT_MULTI_RESP PDUs, you must use two functions:

```
int build_dad( dad_t **dest, dad_t *source );
int destroy_dad( dad_t *sourceList );
int build_udad( udad_t **dest, udad_t *source );
int destroy_udad( udad_t *sourceList );
```

Please check the SMPP-3.4 specification and the examples added to handle this PDUs.

We'll see each function in detail. The library gives you an example of each PDU handled.

2. Packaging function of data structures

The function `smpp34_pack( ... )` takes five parameters and returns an integer value that describes the operation result. A value distinct of 0 (zero) is represetative of an error in the packaging attempt. Then there is a description in text mode in the global variable `smpp34_strerror`.

```
extern int smpp34_errno;
extern char smpp34_strerror[2048];

int smpp34_pack( uint32_t type,    /* in  */
                 uint8_t *ptrBuf,  /* out */
                 int      ptrSize, /* out */
                 int     *ptrLen,  /* out */
                 void    *tt       /* in  */  )
```

Where:

type: is the PDU command_id that we want to pack, the parameter value is related to a specific data structure.

ptrBuf: is a buffer reference, where we will store the PDU packaged. The memory must be reserved externally, it would be dynamic or static memory, but the function is not responsible for free any dynamic memory passed as a parameter.

ptrSize: This parameter is a integer that describes the buffer lenght where we will store de PDU packaged (the previus parameter).

ptrLen: In a success case, this variable keep the data lenght in the buffer. Obviously always *ptrLen* < *ptrSize*.

tt: It's a reference to any data structure listed in the introduction, and corresponds to the value of the first parameter.

2.1. An example

We'll see a small example to use this function, the creation of data object, the load of information in data structure and the pack in a buffer are detailed in this example.

Example 1. Pack and dumpBuff example.

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <netinet/in.h>
#include "smpp34.h"
#include "smpp34_structs.h"
#include "smpp34_params.h"

char bufPDU[2048];
int  bufPDULen = 0;
char bPrint[2048];

int main( int argc, char *argv[] )
{

    int  ret = 0;
    bind_transmitter_t pdu;

    /* Init PDU */
    memset(&pdu, 0, sizeof(bind_transmitter_t));
    pdu.command_length   = 0;
    pdu.command_id       = BIND_TRANSMITTER; /* defined in smpp34.h */
    pdu.command_status   = ESME_ROK; /* defined in smpp34.h */
    pdu.sequence_number  = 1;
    snprintf(pdu.system_id, sizeof(pdu.system_id), "%s", "user");
    snprintf(pdu.password, sizeof(pdu.password), "%s", "pass");
    snprintf(pdu.system_type, sizeof(pdu.system_type), "%s", "type");
    pdu.interface_version = SMPP_VERSION;
    pdu.addr_ton          = 2;
    pdu.addr_npi          = 1;
    snprintf(pdu.address_range, sizeof(pdu.address_range), "%s", "123");


    /* Linealize PDU to buffer */
    memset(&bufPDU, 0, sizeof(bufPDU));
    ret = smpp34_pack( pdu.command_id,
            bufPDU, sizeof(bufPDU), &bufPDULen, (void*)&pdu);
    if( ret != 0 ){
        printf("Error in smpp34_pack():%d:\n%s\n",
                smpp34_errno, smpp34_strerror);
        return( -1 );
    };

    /* Print Buffer */
    memset(bPrint, 0, sizeof(bPrint));
    ret = smpp34_dumpBuf(bPrint, sizeof(bPrint), bufPDU, bufPDULen);
```

```
    if( ret != 0 ){
        printf("Error in smpp34_dumpBuf():%d:\n%s\n",
                smpp34_errno, smpp34_strerror );
        return( -1 );
    };

    printf("The PDU bind_transmitter is packet in\n%s", bPrint);
    return( 0 );
};
```

At the last, this same example is refered by `smpp34_dumpBuf( ... )`. This example is compiled by:

```
[rtremsal@localhost dist]$ gcc -o lo lo.c -I./include -static -L./lib -lsmpp34
[rtremsal@localhost dist]$ ./lo
The PDU bind_transmitter is packet in
   00 00 00 26 00 00 00 02   00 00 00 00 00 00 00 01   ...&.... ........
   75 73 65 72 00 70 61 73   73 00 74 79 70 65 00 34   user.pas s.type.4
   02 01 31 32 33 00                                   ..123.
```

3. Unpackaging function of data structures

The function `smpp34_unpack( ... )` takes four parameters and returns an integer value that describes the operation result. A value distinct of 0 (zero) is an internal error in the unpackaging attempt. Then there is a text description in the global variable `smpp34_strerror`.

```
extern int smpp34_errno;
extern char smpp34_strerror[2048];

int smpp34_unpack( uint32_t type,   /* in  */
                   void    *tt,     /* out */
                   uint8_t *ptrBuf, /* in  */
                   int     ptrLen,  /* in  */ )
```

Where:

type: This parameter is the PDU command_id that we want to unpack, the value of this parameter is related to data structure described in the introduction.

tt: It's a pointer to one of the data structures listed in the introduction and it corresponds to the first parameter value. The pointer to that structure describes where the content of the buffer is to be stored.

ptrBuf: Is a buffer pointer, where the packaged PDU is stored and is ready to be processed.

ptrLen: This variable refers to the buffer lenght above described.

3.1. An example

Here you have a small example on how to use the function. The creation of a buffer with binary data, the unpack function call and the information load over a data structure is showed.

Example 2. Unpack and dumpPdu examples.

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <netinet/in.h>
#include "smpp34.h"
#include "smpp34_structs.h"
#include "smpp34_params.h"

char bufPDU[] = { 0x00, 0x00, 0x00, 0x36, 0x00, 0x00, 0x00, 0x01,
                  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
                  0x73, 0x79, 0x73, 0x74, 0x65, 0x6D, 0x5F, 0x69,
                  0x64, 0x00, 0x70, 0x61, 0x73, 0x73, 0x00, 0x73,
                  0x79, 0x73, 0x74, 0x65, 0x00, 0x00, 0x02, 0x01,
                  0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x5F,
                  0x72, 0x61, 0x6E, 0x67, 0x65, 0x00 };

int  bufPDULen = 0;
char bPrint[2048];

int main( int argc, char *argv[] )
{

    int  ret = 0;
    bind_receiver_t pdu;
    uint32_t tempo;
    uint32_t cmd_id;

    /* Init PDU ***********************************************************/
    memset(&pdu, 0, sizeof(bind_receiver_t));
    memset(&bPrint, 0, sizeof(bPrint));
    memcpy(&tempo, bufPDU+4, sizeof(uint32_t));
    cmd_id = ntohl( tempo );

    /* unpack PDU *********************************************************/
    ret = smpp34_unpack(cmd_id, (void*)&pdu, bufPDU, sizeof(bufPDU));
    if( ret != 0){
        printf( "Error in smpp34_unpack():%d:%s\n",
                smpp34_errno, smpp34_strerror);
        return( -1 );
    };

    /* Print PDU **********************************************************/
    memset(bPrint, 0, sizeof(bPrint));
    ret = smpp34_dumpPdu(cmd_id, bPrint, sizeof(bPrint), (void*)&pdu );
    if( ret != 0){
        printf( "Error in smpp34_dumpPdu():%d:%s\n",
```

```
                smpp34_errno, smpp34_strerror );
        return( -1 );
    };
    printf("The received PDU: \n%s\n", bPrint);

    return( 0 );
};
```

At the last, this example is refered from the description of `smpp34_dumpPdu( ... )`.

This example is compiled by:

```
[rtremsal@localhost dist]$ gcc -o ll ll.c -I ./include -static -L ./lib -lsmpp34
[rtremsal@localhost dist]$ ./ll
El PDU recibido es:
command_length                [00000036] - [54]
command_id                    [00000001] - [BIND_RECEIVER]
command_status                [00000000] - [ESME_ROK]
sequence_number               [00000001] - [1]
system_id                     [system_id]
password                      [pass]
system_type                   [syste]
interface_version             [00]       - [0]
addr_ton                      [02]       - [TON_National]
addr_npi                      [01]       - [NPI_ISDN_E163_E164]
address_range                 [address_range]
```

4. Other unpackaging function of data structures

The function `smpp34_unpack2( ... )` has the same efect that `smpp34_unpack( ... )`, but it doesn't need the command_id parameter. All functions need to identify the command_id, because buffer parsing depend about command_id. The function `smpp34_unpack2( ... )` returns an integer value that describes the operation result. A value distinct of 0 (zero) is an internal error in the unpackaging attempt. Then there is a text description in the global variable `smpp34_strerror`. The complete code is showed.

```
extern int smpp34_errno;
extern char smpp34_strerror[2048];

int smpp34_unpack2( void    *tt,     /* out */
                    uint8_t *ptrBuf, /* in  */
                    int     ptrLen,  /* in  */ )
{
    uint32_t cmdid;
```

```
    uint32_t tempo;
    memcpy(&tempo, ptrBuf + 4, sizeof(uint32_t)); /* get command_id PDU */
    cmdid = ntohl( tempo );
    return( smpp34_unpack(cmdid, tt, ptrBuf, ptrLen) );
};
```

Where:

tt: It's a pointer to one of the data structures listed in the introduction and it corresponds to the first parameter value. The pointer to that structure describes where the content of the buffer is to be stored.

ptrBuf: Is a buffer pointer, where the packaged PDU is stored and is ready to be processed.

ptrLen: This variable refers to the buffer lenght above described.

5. Data structures dump function

The function `smpp34_dumpPdu( ... )` takes four parameters and returns an integer value that describes the operation result. A value distinct of 0 (zero) means an error in the operation. There is a description in text mode of the error in the global variable `smpp34_strerror`.

```
extern int smpp34_errno;
extern char smpp34_strerror[2048];

int smpp34_dumpPdu( uint32_t type,      /* in  */
                    uint8_t *dest,      /* out */
                    int      size_dest, /* in  */
                    void    *tt         /* in  */ )
```

Where:

type: Is the PDU command_id that identifies the data structure. This value is directally related with a specific data structure.

dest: Is a buffer reference, where we will store the PDU dump. The memory must be reserved externally, it would be dynamic or static memory, but the function is not responsible of freeing it.

size_dest: This integer describes the lenght of the destination buffer (the previous parameter).

tt: This reference is a pointer to data structures listed in the introduction. This structure is identified by the first parameter.

5.1. An example

The Example 2 describes the use of this function.

6. Buffer dump function

The function `smpp34_dumpBuff( ... )` takes four parameters and returns an integer value that describes the operation result. A value distinct of 0 (zero) describes an operation error and a description in text mode is stored in `smpp34_strerror`.

```
extern int smpp34_errno;
extern char smpp34_strerror[2048];

int smpp34_dumpBuf( uint8_t *dest,  /* out */
                    int      destL, /* in  */
                    uint8_t *src,   /* in  */
                    int      srcL   /* in  */ )
```

Where:

dest: This parameter is a buffer reference where the dumped buffer will be stored. The memory must be reserved externally, it would be dynamic or static memory, but the function is not responsable for freeing it.

destL: This integer is the lenght of the buffer (the previus buffer).

src: This pointer is a reference to the source buffer (binary buffer).

srcL: This integer is the lenght of the buffer (the previus buffer).

6.1. An example

The Example 1 describes the use of this function.

7. Optional parameters

The functions `build_tlv( ... )` and `destroy_tlv( ... )` allow to create and destroy elements linked in a list of variables called optional parameters. This parameters are in some data structures in the SMPP-3.4 protocol, and they are called optional parameters.

```
int build_tlv( tlv_t **dest,
               tlv_t  *source );

int destroy_tlv( tlv_t *sourceList );
```

Where:

dest: This pointer is a reference to pointer of data type tlv_t. In an example, we'll see how use this function.

source: This pointer is a reference to tlv_t.

sourceList: This pointer is a reference to tlv_t. In `destroy_tlv( ... )` the parameter is a list linked reference.

7.1. An example

The use of optional parameters is bounded to data type tlv_t that is not present in all data structures of SMPP-3.4 protocol We'll see an example of this function.

```
#define TEXTO "mensaje de texto numero 01"
:
{
    submit_sm_t pdu;
    tlv_t tlv;

    memset(&tlv, 0, sizeof(tlv_t));
    tlv.tag = TLVID_user_message_reference;   /* tag present in submit_sm */
    tlv.length = sizeof(uint16_t);
    tlv.value.val16 = 0x0024;                                    /* valor */
    build_tlv( &(pdu.tlv), &tlv );    /* value attached to main structure */

    memset(&tlv, 0, sizeof(tlv_t));
    tlv.tag = TLVID_more_messages_to_send;    /* tag present in submit_sm */
    tlv.length = sizeof(uint8_t);
    tlv.value.val8 = 0x24;                                       /* valor */
    build_tlv( &(pdu.tlv), &tlv );    /* value attached to main structure */

    memset(&tlv, 0, sizeof(tlv_t));
    tlv.tag = TLVID_message_payload;          /* tag present in submit_sm */
    tlv.length = strlen(TEXTO);
    memcpy(tlv.value.octet, TEXTO, tlv.length);                  /* valor */
    build_tlv( &(pdu.tlv), &tlv );    /* value attached to main structure */

    /* Pack and send data in pdu */

    destroy_tlv( pdu.tlv ); /* Free pdu list */
}
```

8. Handling of destination address lists in SUBMIT_MULTI and SUBMIT_MULTI_RESP PDUs

Like the previus section, the problem of handle dinamic list, now in SUBMIT_MULTI y SUBMIT_MULTI_RESP PDU, is resolved with the following functions: `build_dad( ... )`, `build_udad(`

`... )`, `destroy_dad( ... )` and `destroy_udad( ... )`. This functions allow create and destroy dinamic lists of parameters, now of destination address.

```
int build_dad( dad_t **dest,
               dad_t  *source );
int destroy_dad( dad_t *sourceList );
int build_udad( udad_t **dest,
               udad_t  *source );
int destroy_udad( udad_t *sourceList );
```

Please check the following applications code for understand how to handle this PDUs: `submit_multi_test` and `submit_multi_resp_test`. If you already handle optional parameters, you don't be problems with this functions.

9. Examples included in library release.

9.1. PDU samples.

The library has an example for each PDU on SMPP-3.4 protocol. The applications are executed without parameters and the idea is to test the API functions through a serie of steps.

1. To declare two variables of the same type and to assign values to one of them.
2. Apply a PDU dump to the first data structure. It will show all data components of de object data.
3. Pack the structure into a buffer and dump it.
4. Unpack this buffer over the second variable defined in the first point.
5. Do a dump of the second structure.

9.2. An ESME basic.

We have added a little ESME that submits a message. The sintax to run the ESME is.

```
[rtremsal@localhost bin]$ ./esme
Error in parameters
usage: ./esme  -c file.xml [-h]
    -c /path/to/file.xml: config file path.
    -h : Help, show this message.

[rtremsal@localhost bin]$ more esme.xml
<?xml version="1.0"?>
<config>
    <conn_tcp host="127.0.0.1" port="9090"/>
    <conn_smpp system_id="sytem" password="asdfg34" system_type="type01"/>
    <smpp_msg src="5565" dst="0911110000" msg="Este es un ejemplo 01"/>
```

```
</config>
```

The config file has parameters to made the connection tcp, the connection smpp and the parameters to send a message. This application is ejecuted by:

```
[rtremsal@localhost bin]$ ./esme -c esme.xml
Error in connect(127.0.0.1:9090)
Error in tcp connect.

[rtremsal@localhost bin]$ ./esme -c esme.xml
------------------------------------------------------------
SENDING PDU
command_length                [00000029] - [41]
command_id                    [00000002] - [BIND_TRANSMITTER]
command_status                [00000000] - [ESME_ROK]
:
:
:
------------------------------------------------------------
RECEIVE BUFFER
   00 00 00 10 80 00 00 06   00 00 00 00 00 00 00 03   ........ ........

RECEIVE PDU
command_length                [00000010] - [16]
command_id                    [80000006] - [UNBIND_RESP]
command_status                [00000000] - [ESME_ROK]
sequence_number               [00000003] - [3]
```

In the first case, a smpp server is not present, then there was an error in the tcp level. In the second case, helped by a test smpp client program `sctt` - (SMPP Client Test Tool). available in SMS Forum (http://www.smsforum.com).

In the second case, the internal operations that allow sending a message are:

- Connect the ESME in TCP level.
- Connect the ESME in SMPP level through PDU BIND with validation parameters (We connect in TRANSMITTER mode), wait for a confirmation through PDU BIND_TRANSMITER_RESP.
- Send the message (SUBMIT_SM PDU) and receive the confirmation in SUBMIT_SM_RESP PDU.
- Disconnect the ESME in SMPP level sending UNBIND PDU, we wait for a confirmation UNBIND_RESP PDU.
- Disconnect the ESME in TCP level closing the socket.

Although this example is very basic it allow us to see in detail the function of the library inside an ESME.

10. Conclusions and future works.

The library main focus is based in the management of data structures. Although this example is about SMPP-3.4 implementation, we think that the same principle would be applied to any TCP protocol.

The next goal is the implementation of version 5.0 of SMPP protocol.