Uppsala University
Department of Information Technology
Project CS 2010

OpenMSC
Product Report

SMS Capable GSM Network

Muhammad Yousaf
Umesh Paudyal
Tobias Vehkajarvi
Wenbin Wu
Tejas Oza
Papanash Muthuswamy
Premkumar Janagarajan
Johan Drake
Wenfei Zhu

Contents

Abstract

This document describes the work by nine Computer Science students of Uppsala University taking the project DV course. The goal of our project is to implement an SMS capable Mobile Switching Centre MSC which is a part of the GSM network that follows 3GPP standards. The project was developed in the functional programming language erlang/OTP, using Scrum as a project methodology.

Glossary

API	Application Programming Interface
BSC	Base Station Controller
BSS	Base Station System
BSSAP	BSS Application Part
BSSMAP	BSS Management Application Part
BTS	Base Transceiver Station
CC	Connection Confirm
CM	Connection Management
CR	Connection Request
CREF	Connection Refuse
DA	Destination Address
DTAP	Direct Transfer Application Part
DT1	Data form 1
DT2	Data form 2
GSM	Global System for Mobile communication
HLR	Home Location Register
IMSI	International Mobile Subscriber Identity
IT	Inacivity Test/Timer
LM	Location Management
MAP	Mobile Application Part
MO	Mobile Originating
MM	Mobility Management
MS	Mobile Station
MSC	Mobile Switching Centre
MT	Mobile Terminating
OA	Originating Address
OTP	Open Telecom Platform
RLC	Release Complete
RLSD	Released
RR	Radio Resource
SCCP	Signalling Connection Control Part
SMS	Short message Service
SMSC	Short Message Service Centre
SS7	Signaling System 7
SIM	Subscriber Identity Module
TMSI	Temporary Mobile Subscriber Identity
UDT	Unitdata
VLR	Visitor Location Register
MSISDN	Mobile Subscriber Integrated Services Digital Network
MTP	Message Transport Part of SS7
PSTN	Public Switch Telephone Network

1 Introduction

The goal of our project is to implement an SMS capable Mobile Switching Centre (MSC) which is a part of the GSM network that follows 3GPP standards. Mobile phones connected to our GSM network and registered in HLR can send and receive SMS.

We would like to acknowledge Mobile Arts who provided us BTS, HLR, SMSC and the OpenBSC [10] which is an open source project of BSC.

The name of the project is OpenMSC since we are planning to make our project open source. Our software along with OpenBSC can provide a foundation for testing and research purposes in GSM network. To our knowledge, there is no other open implementation of Erlang based MSC (Mobile Switching Centre) and we are the first to implement an MSC that is capable of sending and receiving SMS.

2 Background

2.1 Erlang / OTP

Erlang[9] is a general-purpose concurrent programming language. It is a functional language, with strict evaluation, single assignment, and dynamic typing. OTP[9] abbreviates

- Open:- Refers to the openess towards other programming languages and protocols.
- Telecom:- Refers to the characteristic of telecom systems since its distributed, fault tolerant, concurrent and soft real time.
- Platform:- Refers to the role that OTP Plays as a middle ware in large scale development. Erlang/OTP includes Libraries, applications,tools and design principles.

2.2 GSM

GSM [16] (Global Systems for Mobile Communication) is a mobile telephony network based on the cellular concept. Users can place and receive calls without being fixed to a specific location or wired to a physical connection. The general architecture of the GSM network is shown below.

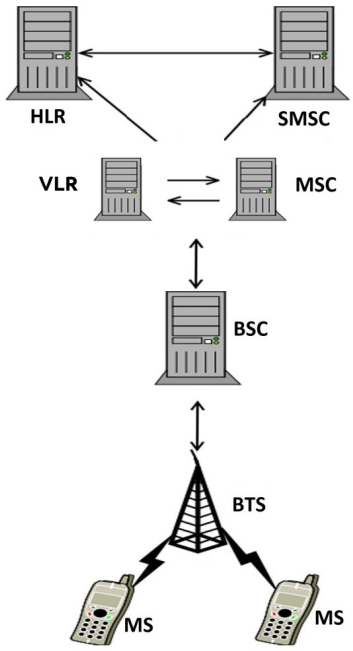

Fig 2.2 Basic GSM Architecture.

2.2.1 Mobile station (MS)

MS[16] is a GSM enabled handset OR mobile phone and most of the modern handsets are equipped with this feature. MS consist of components and software which are capable of communicating with the mobile network. These components are antenna, amplifier, receiver, transmitter and other hardware and software that are used to convert between Radio Frequency waves and audio signal. There are several different ways of identifying MS such as MSISDN[16], IMSI [16] and TMSI[16] for different purposes. MSISDN is the mobile number of user that uniquely identifies the subscriber and whenever making calls or sending SMS, MSISDN is used for routing. IMSI is also a unique number used to identify the SIM [16] card globally and generally, the size of IMSI is 64 bits. TMSI is a temporary identifier generated at random when the MS is connected to the network and is used for minimizing the risk of eaves-dropping over the air interface.

2.2.2 Base station subsystem (BSS)

The base station (BTS)[16] and base station controller (BSC)[16] is collectively known as the Base station subsystem. Each of these components are described below. Communication between MS and BTS is carried out over a interface known as the um interface and between BSC and BTS the Abis interface is used.

2.2.3 Base station (BTS):

It is a hardware device allowing a MS to connect over air interface and is controlled by a BSC so that it is basically a connector between MS and the network. General architecture of a BTS consists of tranceiver(TRX), power amplifier(PA), combiner, duplexer, antenna etc.

2.2.4 Base station controller(BSC):

Base station controller mainly deals with communication with Base station and mobile station controller. It is basically a software component to ensure proper communication between MSC and BTS. Generally a BSC can control hundreds of BTS at a time. BSC is very smart module in GSM where it handles the handover when the mobile phone moves from BTS to BTS and also allocates the radio channels. When mobile phone is turned on or comes into the respective network it sends its mobile measurements like mobile device is capable of GPRS functionality etc..to the network and BSC receives this information from mobile station and sends it to network.

2.2.5 Mobile switching centre (MSC)

MSC [16] is one of the GSM network components which routes the SMS, call and other services. So, basically the routing of SMS includes functionalities like ensuring the network connections, handling of location related stuffs and maintaining the packet flow according to GSM specifications like connections to VLR, HLR and SMSC.

2.2.6 Visitor location register (VLR)

VLR [16] is a database that contains temporary information about all visiting subscribers in the area which it serves. It keeps information about subscription level, supplementary services, current location of the visitor region. Its also keeps track of the current status of the MS like whether the mobile station is on or off. VLR contacts HLR to request data for the mobile phones attached in its location area. VLR is tightly connected to the MSC.

2.2.7 Home location register (HLR)

HLR [16] is a network subscribers database that contain all the information of the subscriber of the network for example there LAI,IMSI[18] and MSISDN so that whenever network requires any information about the subscribe its HLR responsibility to provide the necessary information.

2.2.8 Short Message Service Centre (SMSC)

SMSC[16] stores and forwards SMS messages. If the receiver is offline or unavailable due to any reason, then it stores the SMS and forwards as soon as it gets notified that the receiver available again. So its main purpose is to store the short message received from lower layers of the network and forward to the network when needed.

Some of the services that are necessary for SMS procedure are as follows:

2.3 Location Management

[7] This service is used between MSC and VLR to update location information in the network. It is initiated by an MS when changing the location area or at first registration. Whenever the user first turn on the mobile the location updating request is sent to the network. Also, When the user changes the location the location updating request is sent to the network by MS.

2.4 Identity Management

[7] This service is used by a VLR in order to get, via the MSC, the IMSI of subscriber. It happens when a subscriber has identified himself with a TMSI not allocated to any subscriber in the VLR. It also includes the TMSI reallocation procedure.

2.5 Subscriber Management

[7] This service is used by an HLR to update a VLR with certain subscriber data in the following events:

- the operator has changed the subscription of one or more supplementary services, basic services or data of a subscriber.
- the operator has applied, changed or removed Operator Determined Barring.[5]
- the subscriber has changed data concerning one or more supplementary services by using a subscriber procedure.

The HLR provides the VLR with subscriber parameters at location updating of a subscriber or at restoration. In this case, this service is used to indicate explicitly that a supplementary service is not provisioned, if the supplementary service specification requires it.

2.6 Paging and search

[7] This service is used between VLR and MSC to initiate paging of an MS for mobile terminated call set-up, mobile terminated short message or unstructured SS notification. This message basically tells the BSC to invoke search procedure for respective mobile phone in BTS. When the BTS finds the mobile phone it sends the page response to the network.

2.7 Access Management services

[7] This service is used between MSC and VLR to initiate processing of an MS access to the network, e.g. in case of mobile originated call set-up or after being paged by the network.

2.8 Short message service management services

This service is used between the gateway MSC and the SMSC to send and receive SMS. It also checks the status of Mobile Subscribers in the VLR to authorize sending and receiving. Moreover, this service also alerts HLR a mobile subscribers memory available situation.

2.9 Network protocols

Communication, MSC to HLR and MSC to SMSC is carried out by application layer protocol known as MAP.[7] MAP provides services like handling and routing calls and other services for short message services in the network. Communication between MSC and BSC uses SCCP. SCCP is responsible for establishing connection, providing flow control and error control services to the network.

2.9.1 Signaling System no.7 (SS7)

This standard is a global standard for signaling in telecommunication networks. The reason behind the development of SS7 [15] was to make the network more efficient and maximize the utilization of the resources. The development of SS7 lead to a considerably improved performance in the PSTN [15]. SS7 is a reliable packet-switched data network that has an important part in both wired and wireless networks.

2.9.2 Signalling Connection Control Part (SCCP)

SCCP [8] is used to transfer radio-related messages in mobile communication. This protocol resides in Layer 3 of OSI[16] model in Computer Networks and in GSM it is being widely used by Base Station Subsystem Mobile Application Part (BSSMAP)[2] and Direct Transfer Application Part (DTAP)[2] to transfer radio related messages. SCCP concurrently works with Transfer Capabilities Application Part (TCAP)[15] to handle this servies. SCCP has the capabilities of flow control and routing of messages during communication. When there is large message to fit in one Message Signal Unit (MSU)[2] SCCP enhances the transfer capabilities. SCCP supports connection oriented data transfer as well as connection-less data transfer and also it is responsible for management and addressing of subsystems.

There are are two basic kinds of messages types; connectionless and connection oriented messages.

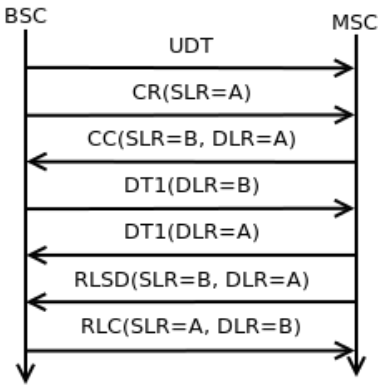

Fig 2.9.2 SCCP Message Flow

A connectionless message like Unit data (UDT)[3] does not use references. Only the BSC can request for a connection and that is achieved by sending a Connection Request (CR)[3] message containing its source local reference called A. If the connection request is accepted the MSC sends a Connection Confirm (CC)[3] message for the destination reference A and its own source local reference B, and at this point both parts knows each others references for the connection and the connection is established. Data form 1 (DT1)[3] is used for transferring data and contains the reference of the receiving part. When sending a release (RLSD)[3] message will close the connection and a release confirm (RLC)[3] should be sent back as acknowledgement.

3 Design and Implementation

This part explains the design of OpenMSC with sequence diagrams and an Erlang supervision tree. Also, it explains the communication of MSC with other nodes like HLR, SMSC and BSC.

3.1 Supervision Tree

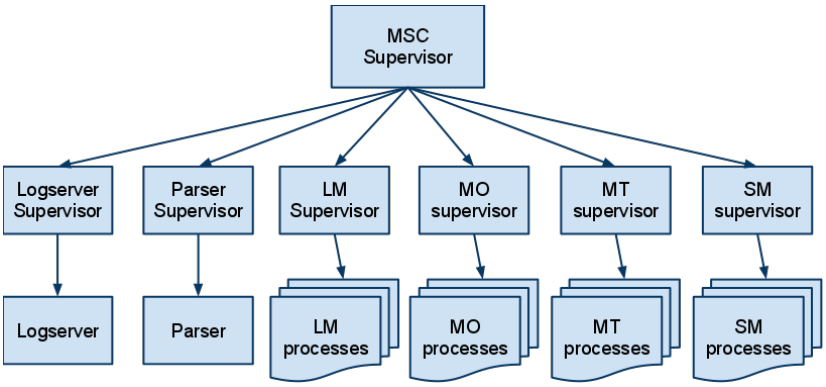

Fig 3.1.a Supervision tree of OpenMSC

The MSC applications supervision tree is four levels deep. On the first level is the top supervisor. This supervisor in turn controls one supervisor for each service provided by the application. Each service supervisor has its own identical supervision sub-tree with the following structure:

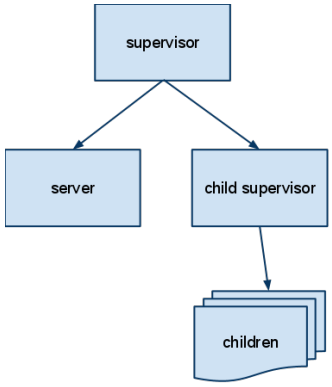

Fig 3.1.b Supervision tree in Erlang.

3.1.1 The server

Below the service supervisor there is a server and a child supervisor. The Server is the process that contains the services API. These functions are linked by the two interfaces towards the BSC and SS7. They are called whenever a message is received through either interface. When receiving a message it is the server's obligation to determine what the correct response is. In most cases the server will only spawn a child using the child supervisor to take care of the work. In some cases the server will instead send a message to an already existing child. The server might also contain some information about the children if needed.

3.1.2 The child

All the children are located below the corresponding child supervisor. The children (or workers) in each service is where most of the work is done. A child is usually spawned to perform a specific task, after which it dies. Most children are spawned when the MSC receives a message from outside. The child does some

work, then sends the message onwards in the GSM system and waits for a reply. If the a reply received or the child reaches its timeout, the child sends a response back to where the original message came from. The child then dies. There is no hard limit to the number of children that can be alive at any one point.

3.2 Communication

3.2.1 MSC communication with HLR and SMSC

SMSC and HLR is provided by MobileArts and we need the interface to communicate between them. As per the requirement we defined the interface and MobileArts implemented it. Detailed explanation is given below:

MAPP node: MSC communicates with HLR and SMSC through MAPP node. MAPP node has two interfaces called sp_map_api and msc_map_api which facilitates outward and inward communication for other nodes with the MSC respectively. API stands for application programming interface.

SP_MAP_API Interface: This interface facilitates the outward communication of MSC with HLR and SMSC. Actual communication between MSC and HLR, SMSC is done through TE SS7 stack which implements the SS7 protocols. The interface SP_MAP_API abstracts the communication between the MSC node and the SS7 stack and subsequently with HLR and SMSC. It provides API's through which modules in the MSC such as MO[7], MT[7], LM[7], SM[7] and VLR can communicate with other nodes. MO communicates with SMSC by calling mo_forward API when it wants to forward a message to the SMSC. MT communicates with SMSC by calling mo_forward_ack API when it wants to acknowledge SMSC about the delivery of the message to the MS. If MT fails to deliver the message to the MS, then it informs SMSC through the same API about the failure along with the reason which helps SMSC to retain that message in its SAF (Store and Forward) mechanism for later retries which is implementation specific. VLR communicates with HLR by calling update_location API when a MS wants to be connected to the network or when an already connected MS is switched off and turned on. MO and LM communicates with SMSC by calling ready_for_sms API when there is message waiting for MS in the SMSC because of previously failed delivery which may be because of memory unavailabilty in the MS or the MS has been switched off.

MSC_MAP_API Interface: This interface facilitates the inward communication of MSC with HLR and SMSC. It provides API's through which HLR and SMSC can communicate with the modules in the MSC node. SMSC communicates with MSC by calling mo_forward_ack API when it wants to send an acknowledgment of the message delivery or when it wants to report the failure to deliver the message along with the reason. SMSC communicates with MT by calling mt_forward API when it wants to send an MT short message. LM communicates with MSC by calling update_location_ack API when it wants to acknowledge the acceptance of MS in the network or when it wants to notify the rejection of MS along with the reason for reject.

3.2.2 MSC communication with BSC

As our BSC we choose to use OpenBSC which is a GSM network in a box software provided as an open source project written in C99 and contains minimal component needed for a complete GSM network. We used a version of OpenBSC which supports communication with BTS and BSC over abis interface and between BSC and a standalone MSC over A interface using SCCP protocol. While considering LM ,MO, MT ,SM, VLR as a whole MSC, the MBinterface is the medium to make MSC talk with OpenBSC. If we mention in detail there lists of modules from where the message of MSC transferred to the OpenBSC. Also we will talk about some protocols that must be followed to make successful communication with OpenBSC. All the modules implemented are listed and described as follows.

MBinterface: When MSC needs to communicate with BSC, it will call interfaces provided in MBinterface and MBinterface will make formal packets using Encoding module and forward to CNODE. After

receiving packets from MSC, CNODE will add BSSAP/DTAP and ip.access header to the packet and forward it to BSC. Interface is used to ensure easy communication between CNODE and each child processes in MSC.Interface is provided to communicate between CNODE and each child processes like MT child, MO child and LM child. So, when it is the case of Network to MS (i.e. the mobile terminated case), the respective child process calls the functions defined in interface to encode respective packets and forward it to BSC through CNODE. Similarly, if it's the case of MS to Network(i.e. the mobile originated case), the packets received from BSC is splitted and decoded into plain text values and forwarded through the interface to the destination child process.

SCCP Protocol: We have a CNODE that uses SCCP library written in C (libsccp.a) for wrapping and splitting header information of SCCP packets. This CNODE is a normal Erlang node that acts as a bridge between OpenBSC(Written in 'C') and OpenMSC(Written in Erlang). There are two processes created in CNODE by using fork(), one is for the communication sockets between CNODE and BSC, the other is for the communication between CNODE and MSC. Both the processes can make use of both sockets. CNODE splits incoming binary packets from OpenBSC into the header and payload. Then it sends the payload part with the header information to the MSC in Erlang binary format. Similarly, when the CNODE gets the message in binary form from OpenMSC it wraps with SCCP and IPA header with payload to a packet and sends it to the OpenBSC.

BSSAP Protocol: BSSAP stands for Base Station System Application Part which consists of Direct Application Part(DTAP)[2] and Base Station System Management Application Part (BSSMAP)[2]. DTAP messages mainly consists of CC[1] and MM[1] messages and used to transfer the messages content between MSC and MS. BSSMAP is responsible for administration and control of the radio resources of BSS. In our project we used some functions of BSSMAP which are required between BSS and MSC for RR[1] of the BSS.

If we go into technical detail we want a library that has functionality that can find the structure of a packet by specifying the BSSAP packet name identifier; for a given BSSAP packet is able to create a binary and from a binary parsing it as a BSSAP packet. The packets related to each protocol must follow 3GPP specifications. The Library API is defined as:

```
@spec find_packet(atom(), [argument()]) -> packet()
@spec create(packet()) -> binary()
@spec parse(binary()) -> packet()
```

The erlang definition of a BSSAP packet is as follows:

```
-record(packet, {identifier,     % Name of the packet.
                 byte,           % Byte representation given from 3GPP specifications.
                 format=[],      % In which order the layers and fields appear.
                 arguments=[]}). % The arguments for the variable fields.
```

Finding the format of the packet involves reading a lot of 3GPP specifications and the table 3.2.2 shows a example of how a packet is defined in the 3GPP specifications, it is the LU_REQ packet used for location management.

Table 9.2.17/3GPP TS 04.08: LOCATION UPDATING REQUEST message content

IEI	Information element	Type / Reference	Presence	Format	Length
	Mobility management protocol discriminator	Protocol discriminator 10.2	M	V	1/2
	Skip Indicator	Skip Indicator 10.3.1	M	V	1/2
	Location Updating Request message type	Message type 10.4	M	V	1
	Location updating type	Location updating type 10.5.3.5	M	V	1/2
	Ciphering key sequence number	Ciphering key sequence number 10.5.1.2	M	V	1/2
	Location area identification	Location area identification 10.5.1.3	M	V	5
	Mobile station classmark	Mobile station classmark 1 10.5.1.5	M	V	1
	Mobile identity	Mobile identity 10.5.1.4	M	LV	2-9

Table: 3.2.2

Short description on how to read the definition.

- M standing for mandatory, O for optional.
- V for variable, L for length.
- Length is in number of bytes.

Note:- The length field represents the number measure in Octets i.e network representation of 1 Byte (8 bits) e.g 1/2 means half octet(4 bits).

After reading several of these definitions and understanding the protocol structure, you can construct a tree structure representing the BSSAP protocol.

A subset of the BSSAP protocol is shown in the figure 1 with real example packets used in the project.

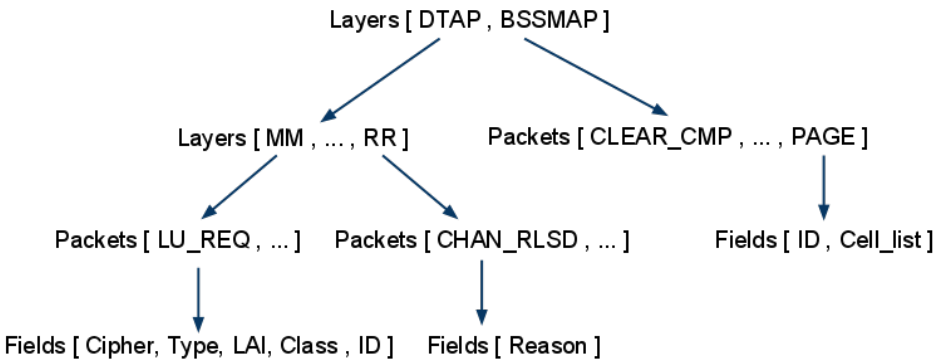

Fig 3.2.2 Tree structure representing a subset of the BSSAP protocol.

Layers consists of more underlying layers or packets and packets consists of variable fields. From the tree structure in figure 3.2.2 we can define an Erlang representation like this:

```
-define(BSSAP,
        [{layer, dtap, [{layer, mm, [{packet, lu_req, [{field, cipher}, {field, type},
                {layer, rr, [{packet, chan_rlsd, [{field, reason}]}]}]
          {layer, bssmap, [{packet, page, [{field, id}, {field, cell_list}]}
            {packet, clear_cmp}]}]).
```

Having the protocol defined in formal way gives us enough information about format and structure of the packets. For each field and layer identifier in the definition above shall have an encoding and corresponding parsing function defined so that they can be used for the more general functions involved in creation and parsing of the complete packet. Parsing will traverse the tree from the root and will try to find a matching branch according the defined parsing function for that particular leaf and adds its results to the erlang packet record and continues until there is nothing left of the binary. Encoding of a packet works in the opposite direction, it will recursively go through the format list and when needed it will pair a variable field with an argument and apply it to the respective encoding function for that element and append it to the resulting binary.

3.2.3 MSC Node

This part explains the design of OpenMSC with sequence diagrams and an Erlang supervision tree. Also, it explains the communication of MSC with other nodes like HLR, SMSC and BSC.

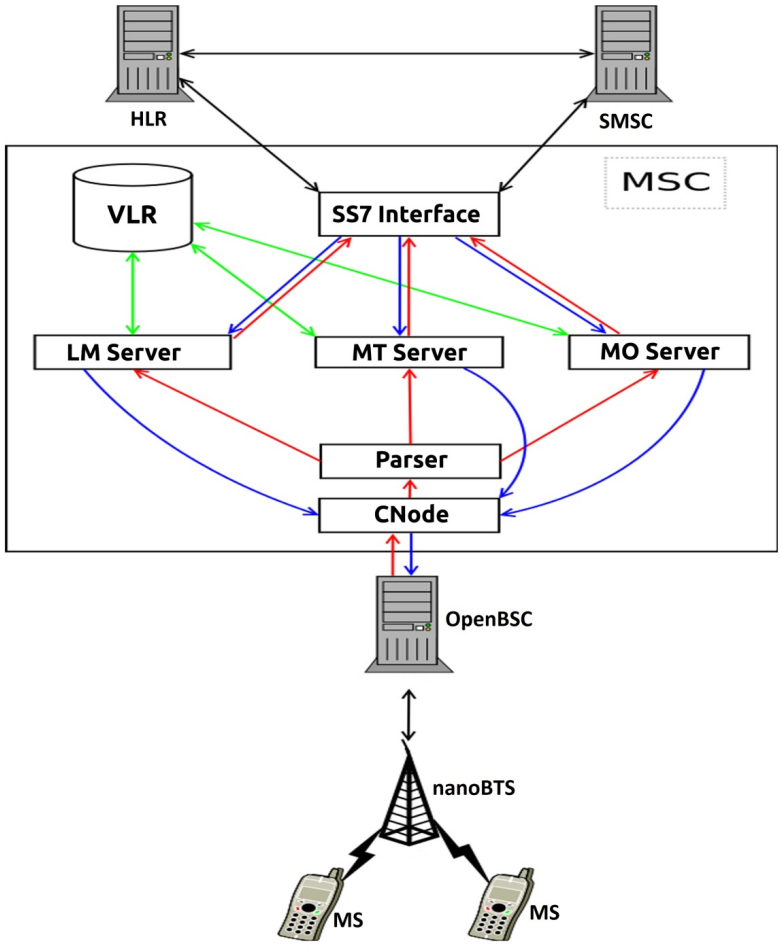

Fig 3.2.3 System Architecture of OpenMSC.

Log Server: The logserver is a server that registers traffic to and from the MSC. Whenever a message is sent or received over either the MB or MAP interface, an Erlang message is sent to the logserver process. When the logserver receives a message it writes some basic information about that message to a binary file. The information contains a time stamp, the type of message sent, the sender and receiver of the message and one field of information about the messages content.

3.2.4 Mobile Originating:

The mobile originated short message service procedure is responsible for delivering the short message to the SMSC. The procedure has been implemented using the most common OTP behavior, the gen_server.

A server process, named mo_server, implemented in the mo_server module is started once the MSC erlang application is started. This server process is a worker process of the mo_supervisor (section 3.1). This server is responsible for invoking the mo_child_sup (section 3.1), a supervisor process under mo_supervisor which is responsible for spawning a child process for each time a mobile tries to send a message or when the mobile station wants to alert the service center about its memory availability to accept more messages. Every time a mobile tries to send a short message or alert message it sends a cm_service_req to the mo_server through the Cnode which in turn tells mo_child_sup to spawn a separate process to handle that request.

A separate child process is spawned, each time when the mo_server forwards a cm_service_req to the mo_child_sup. The child process then handles the request by calling process_access_req in VLR interface (section 3.2.5), which in turn returns either ok or user_error. The child then sends a cm_service_acc or cm_service_rej respectively to the mobile station through the Cnode along with its process id. If it sends a cm_service_acc then the child process waits until the mobile sends a rp_mo_data (mobile originated forward short message, mo_forward) or rp_smma (memory availabilty, mem_available).

Once it receives the rp_mo_data the child process calls send_info_for_mo_sms in VLR to check whether the mobile originated short message is provisioned for that particular mobile phone, also it checks whether the mobile phone is barred. It also retrieves the MSISDN of the mobile station from the VLR database and returns either "ok" along with the MSISDN of the mobile station or user_error along with the reason. The child process then makes an erlang remote procedure call to along with its process id to the procedure named mo_forward in the interface sp_map_api which is in the MAPP node (section 3.2.1) to deliver the short message to the Service center and waits for the acknowledgment from the SMSC. Once it receives a reply from SMSC, it sends an acknowledgment to the mobile station through the Cnode informing either the delivery of the message or the failure of the message.

If the mo_child receives the rp_smma instead of rp_mo_data from the Cnode, then the child process calls makes a remote procedure call to the procedure named ready_for_sms in the interface sp_map_api in the MAPP node to alert the service center about the memory available condition in the mobile station to accept the messages which is stored in the store and forward of the SMSC. Then the child waits for an acknowledgment from SMSC, once it receives a reply, the child process sends an acknowledgment to the mobile station through the Cnode informing of either a delivery or failure.

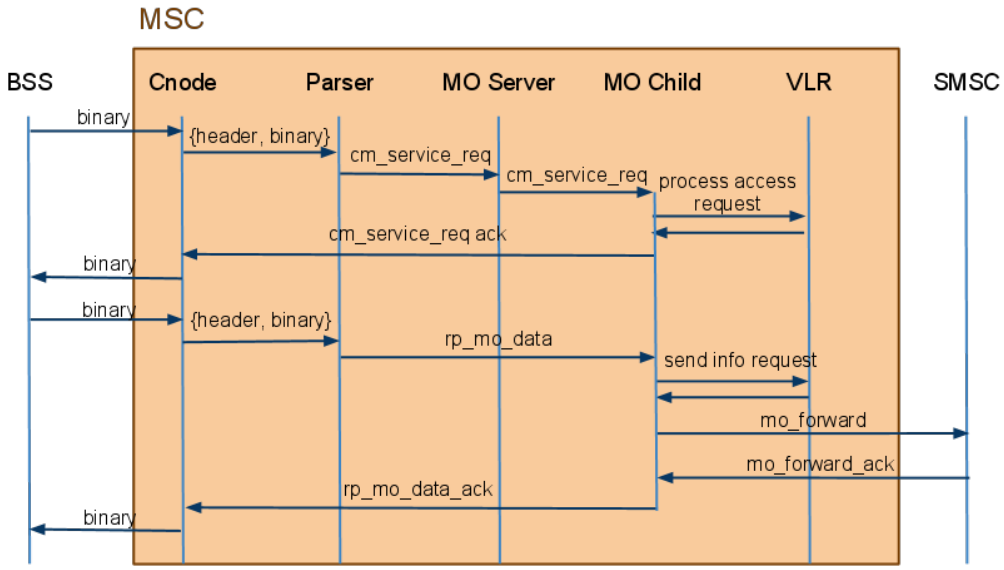

Fig 3.2.4 Sequence diagram for Mobile Originating

3.2.5 Mobile Terminating

A mobile terminating module consists of a mt supervisor, mt server, mt child supervisor and dynamically spawned mt children. The module is responsible for the mapp mt procedure described in GSM MAP Specification (downloaded from 3GPP TS 09.02). It is used to forward sms from a Service Center to a mobile station. Since the supervision tree is described in Chapter 4.1, below is the explanation of the design of mt server and mt child.

MT SERVER: Mt server follows the Erlang OTP server behaviour. It receives messages mt_forward and page_rsp from ss7 stack and bssmap respectively. It also maintains an ets table which contains each mt childs pid, destination IMSI of the MT sms and whether SMSC will send more messages to the destination MS. (example picture of an ets table)

When the server receives a mt_forward(Dest_addr, Sc_addr, Smrpui, More_msg) message from the SMSC, it first checks the ETS table which has the same destination address and with more_msg flag set to true. If there is such an mt child, the server will send the mt_forward message to that mt child. Otherwise, the server will spawn a new mt child to deal with this mt_forward message and add the new mt childs pid, Dest_addr and More_msg to the ets table.

MT CHILD: Mt child is an Erlang OTP finite state machine. It takes care of the mobile terminated short message transfer procedure in MSC. It has 8 states: idle, wf_page, wf_page_rsp, wf_search_rsp, wf_answer_from_vlr, wf_answer from_hlr, wf_answers_from_hlr, wf_sms_cnf and wf_more_msg. The following figure shows the finite state machine of MT Child.

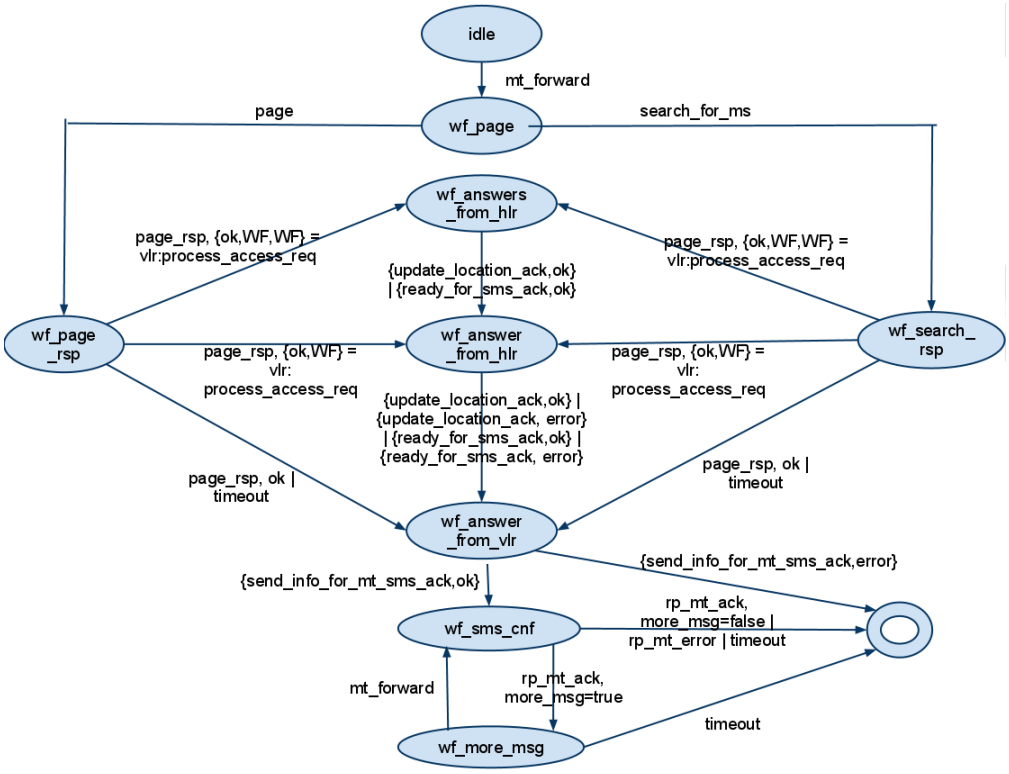

Fig 3.2.5.a State diagram for Mobile Terminating

When a child is spawned, it starts in the idle state, and will receive mt_forward message from the mt server. After it receives mt_fwd, it adds its pid to the mt server's ets table and invoke send_info_for_mt_sms in vlr, then it goes to wf_page state. Wf_page receives either page or search_for_ms from vlr, and goes to wf_page_rsp or wf_search_rsp respectively. Wf_page_rsp and wf_search_rsp are very similar, they both go to wf_answer_from_vlr, wf_answer_from_hlr or wf_answers_from_hlr. The diffecence is that in wf_search_rsp,

after it receives page_rsp, it invokes search_for_ms_ack method in VLR to update a subscriber location, while in wf_page_rsp, it does not. In both states, the child invokes process_access_req in vlr to access a mobile subscriber into the network. The return value of process_access_req will show whether the child should wait for any response from HLR, such as update_location_ack and ready_for_sms_ack. And depends on how many messages the child is waiting for, it goes to wf_answer_from_vlr, wf_answer_from_hlr or wf_answers_from_hlr. If it is not waiting for any messages from the HLR, it goes to wf_answer_from_vlr state; if it is waiting for one message from HLR, it goes to wf_answer_from_hlr state; if it is waiting for two messages from hlr, it goes to wf_answers_from_hlr state.

In wf_page_rsp and wf_search_rsp, if there is a timeout, it invoke search_for_ms_ack in VLR indicating that there is an error because of "no page response", it goes to wf_answer_from_vlr state and terminate itself. The timeout value in wf_search_rsp, wf_page_rsp is 30 seconds, and is 1 minute in wf_sms_cnf. Because when we test our system in real case, we find that sometimes, after a mobile receives a message, it will reply a confirmation after 30 seconds. In that case, if the timeout value in wf_sms_cnf is still 30 seconds, it will terminate before it sends mt_forward_ack to the SMSC, and SMSC will send the same message to the cellphone again.

The sequence diagram for message termination message is as follows.

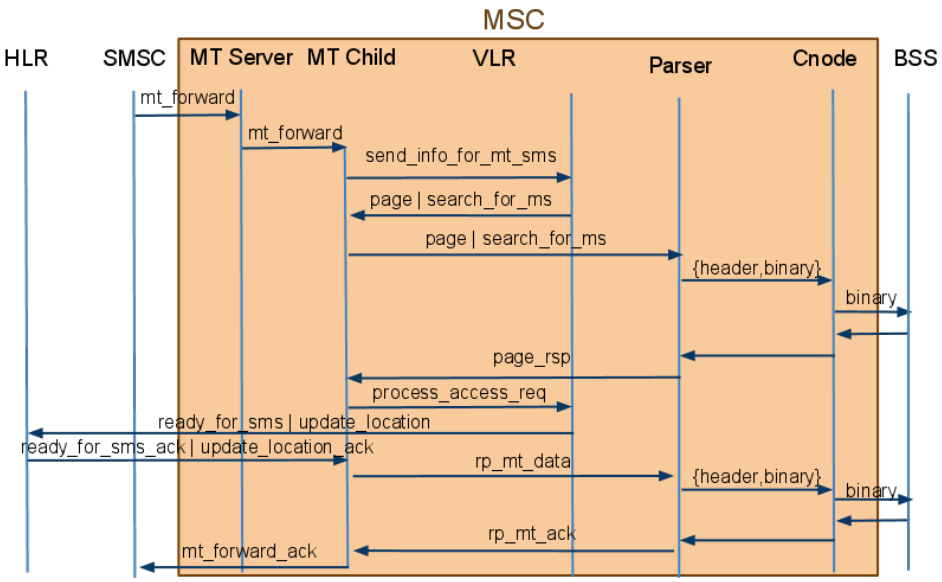

Fig 3.2.5.b Sequence diagram for Mobile Terminating

From the sequence diagram we can see that it is the mt_child and vlr which execute the procedure of mobile terminating. When VLR deals with send_info_for_mt_sms, it will send page or search_for_ms if the the subscriber can receive a message, otherwise, it will send send_info_for_mt_sms_ack with an error message. If the subscriber has an LAI[7] in VLR, the MSC will only page a specific LAI; if not, the MSC will search for the mobile subscribe in every LAI until it finds the MS. In the process_access_req procedure, VLR sends ready_for_sms, update_location to HLR with the process pid as reference, and HLR will send back to the pid, in this case is the mt_child since module VLR is just an interface between MSC and VLR database. After mt_child receives ready_for_sms_ack or update_location_ack, if everything is fine, it sends itself send_info_for_mt_sms_ack indicating the process with VLR is over and sends sms to the MS. After it receives rp_mt_ack indicating the MS has received the sms, it sends mt_forward_ack to the SMSC so that the sms can be deleted from it.

3.2.6 Location Management:

This module contains the LM supervision tree that includes a supervisor, a server and a set of dynamic children. It is responsible for some of the location management and identity management services. It implements the following procedures.

Location updating: This updates the location of the mobile station in VLR and HLR.

Ready for sms: It informs the HLR that a mobile station is reachable and also resets the "mobile not reachable" flag in the VLR.

Provide imsi: It requests for the IMSI to the mobile station when the mobile sends TMSI that is not known in the VLR during location updating procedure.

Forward new tmsi: It forwards the new tmsi to the mobile station when tmsi is reallocated in the VLR.

The procedure is explained below with a sequence diagram

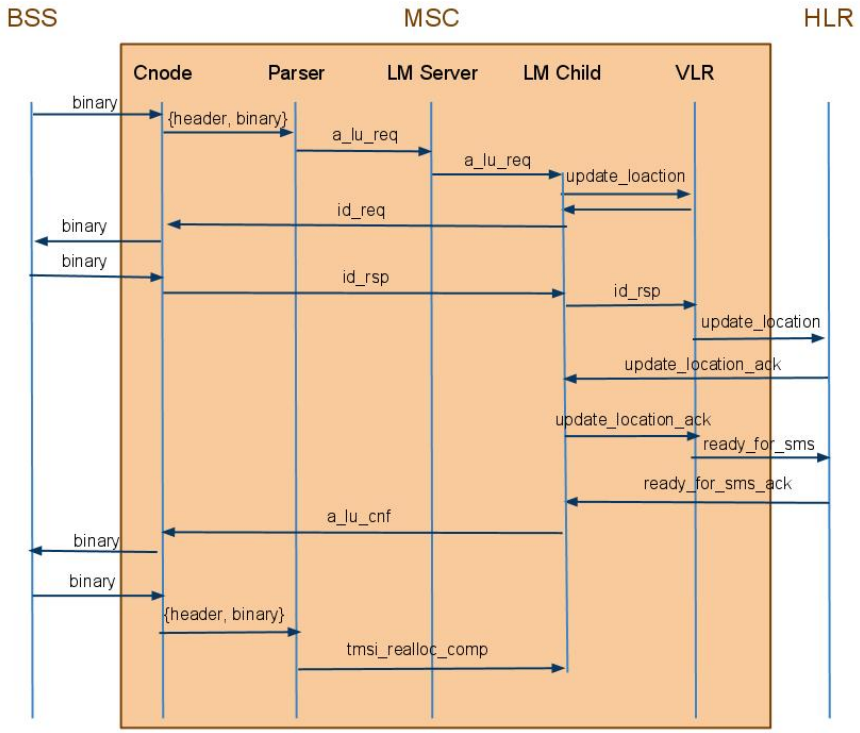

Fig 3.2.6 Sequence diagram for Location Management

When a mobile station gets active(switched on and it comes into a new VLR area) it sends an a_lu_req(location update) to the LM module. The LM child is created as explained in the supervision tree procedure. This child is gen_server process that takes care of all the procedures listed under the LM module.

The a_lu_req contains the location information(LAI) and the identity of the mobile station which can be either IMSI or TMSI. This identity is checked against the VLR by the LM child. Based on this, there are 4 possible cases.

- When the IMSI is known in the VLR, the LM child does not send the id_req request to the mobile station nor does it send the update_location to the HLR. It stores the LAI in the VLR and responds to the request with a_lu_cnf message with the newly allocated TMSI. TMSI is allocated in the VLR whenever it receives a new IMSI or a new TMSI.
- When the IMSI is not known in the VLR, it is inserted in the VLR along with the LAI and update_location message is sent to the HLR that replies with an update_location_ack after communicating with subscriber management module. The TMSI is reallocated in the VLR and an a_lu_cnf message is sent to the mobile station.

- When the TMSI is known is the VLR, it is similar to case 1. It does not require id_req, update_loaction and tmsi allocation. It sends the a_lu_cnf to the mobile station after storing the LAI and TMSI in the VLR.
- When the TMSI is not known, the LM child sends an id_req to the mobile station that responds with id_rsp. This contains the IMSI of the mobile and is inserted into the VLR along with the LAI. The update_location is sent to the HLR and the HLR sends the update_location_ack as a reply. Finally the LM child sends the a_lu_cnf with a newly allocated TMSI to the mobile station.

  In all of the cases, the child process dies after closing the connection when it receives a tmsi_realloc_comp from the mobile station. During this location updating procedure, irrespective of the case the VLR sends a ready_for_sms message to the HLR when the "mobile not reachable" flag of the mobile is set to notify that the mobile is reachable now.

3.2.7 Subscriber management

Subscriber management handles the messages "insert subscriber data" and "delete subscriber data" that the MSC receives from the HLR. Currently the only function that insert subscriber data provides is to insert the msisdn sent into the VLR. The message "delete subscriber data" can be handled by the MSC but it currently only sends an ack back to the HLR and performs no actions in the VLR.

3.2.8 VLR

Our VLR module is composed of a database to store visiting subscribers' information such as IMSI, MSISDN and etc., a database interface to manipulate database, and an interface between the database interface and MSC module.

We uses CouchDB[12] as the database , erlang_couchdb as the database interface. CouchDB is an open source document oriented database. It is written completely in Erlang, so it is easily accessible from erlang applications. Erlang_couchdb is also an open source erlang library. It is actively developed and tries to do as little as possible. It can be downloaded from https://github.com/ngerakines/erlang_couchdb/

By using erlang_couchdb to manipulate the database, we develop the interface between the VLR database and MSC ourselves. The VLR interface provides subscriber management, location management services, network access services, and short messages services. In the include file, it contains default db host, db name, db view name, db fields and so on. Following is a list of database fields.

```
-define(C_BY_HLR,confirmed_by_hlr).
-define(C_BY_VLR,confirmed_by_vlr).
-define(C_BY_R,confirmed_by_radio).
-define(L_INFO_C_HLR,loc_info_confirmed_in_hlr).
-define(IMSI,imsi).
-define(MNRF,mnrf).
-define(TMSI,tmsi).
-define(LAI,lai).
-define(IMSI_DTCH,imsi_detached).
-define(LA_N_ALW,la_not_allowed).
-define(MWF,msg_waitig_flag).
-define(MO_SMS_P,mo_sms_provision).
-define(MO_SMS_B,mo_sms_barred).
-define(OPRT_B,operator_barring).
-define(MSISDN,msisdn).
-define(A_TMSI,allocate_tmsi).
```

Fig 3.2.8 List of database fields in VLR

In the database, we use imsi as the key. But sometimes, the mobile sends its tmsi as its identity for security reason. So we create a view which consists only tmsi and imsi, and set tmsi as a key, so that we can also find subscribers' information by its tmsi.

In VLR interface, we export functions: create_database, id_rsp, update_location_area, send_info_for_mo_sms, send_info_for_mt_sms, insert_subscriber_data, process_access_req, page_ack, search_for_ms_ack, update_location_area_ack and ready_for_sms_ack.

When we start the application, we first create a VLR database by invoking create_database method. In this method, you can also give a specific host address, otherwise, it will be the default host address defined in the include file.

VLR also contains internal functions such as is_sub_known to check whether a imsi exists in the database, vlr_update_location to update the subscriber's location in the HLR and so on. These functions shall be kept private to prevent exposing the data.

3.2.9 Parser

The purpose of this server is to receive a binary string and possible sccp header containing information such as references and protocol class from the CNODE.

One special procedure that has to be done is to keep CNODE updated with the current parser servers pid and that is done by simply sending the pid when starting or restarting .

When receiving a binary string the server will spawn a child process in order to parse it with our implemented BSSAP parser library and if successful the child process makes a corresponding function call in the mbinterface and terminate.

4 Evaluation and Testing

OpenMSC consists of components like Mobilearts HLR,Mobilearts SMSC,Mobile originating server, Mobile termination server, location update server of MSC and BSS. We have tested its functionality and come to the conclusion that all components are functioning properly and all the requirements in requirement specification are met.

We have started our project with connecting BTS to only one system in our network and we tested and analyzed the performance of the system by allowing only one MS to connect to our network and send SMS to itself only. By doing this we are able to concentrate in only few network packets and be able to get rid of few extra messages that we are sending to the network that results in increasing the performance of the overall system.

Finally we tested our system by connecting MS of all nine group members of our network and we can successfully send and receive SMS from one MS to other without effecting the performance of the system but we haven't carried out the extreme stress/load testing as our project goal is rather a proof of concept than having a good performance.

So, having a functional "SMS capable GSM Network" as per project goal statement, has been achieved. Having said that, we still have improved our system by tweaking a little bit on architectural design in the communication part between BSC and MSC without having CNODE since there is always a place for improvement in softwares.

iphone 4,Nokia e51, Nokia 5200, Sony Ericsson K310i, K500i, K700i, W810i, K800i and N86 were the phones used during the testing.

5 Known Issues

- PID as a Reference for Connection Oriented SCCP messages.
- The child process that initially takes care of a incoming CR will send its Pid as a destination reference in connection oriented CC messages, it may cause problem if the child process happened to die or get restarted in any case. In such cases the message may not reach the proper destination as the Pid gets changed if the process got restarted or won't find the process if it died. Using some other unique number instead of process PID is better or keeping track of restarted child processes.
- Parser Restarting:

  When we start our application we need to send the Pid of the Parser server to the CNODE so that the CNODE knows where to send its replies. But whenever the parser restarts, CNODE needs to get updated again but this issue could not be resolved due to pointer error in CNODE.
- Timeouts: Timeouts are not implemented correctly in the MO and LM services. They are child processes that will be waiting for response infinitely. In MT, we find that sometimes, the mobile confirms a received message after 30 seconds. So we set timeout of state wf_sms_cnf as 1 minute, while others as 30 seconds.
- Error messages to BSC: When errors occur during the communication between the MSC and the SMSC or the HLR , the error value are returned in form of an atom by the HLR and SMSC to the MSC. They are translated into byte values when they are forwarded to BSC according to the 3GPP specifications. But, there could be some errors(that we are familar with) which might translate to default byte values.
- Minimal implementation of BSSAP: Does not support optional fields for creation and parsing of packets in the BSSAP protocol.
- TMSI : Creation of TMSI in the location update procedure is based on the IMSI and is not randomly generated. It it unique as long as the IMSI is unique.
- No slots in OpenBSC : Sometimes when paging, during MT the OpenBSC will not respond with an CR message due to "No available slots" error.

6 Future Work

- Calls: Since we have completed the SMS capable GSM Network following the 3gpp documentation, there is room to extend for the calling feature as well. We have implemented the features using abstraction so while adding functionality like calling will be fun and quite smooth to implement.
- Roaming: From this project we have learned the location management but we have not implemented the roaming part as it requires the inter MSC and Intra MSC Handover and also need to change the VLR functionality.
- 3G: OpenBSC has support for 3G/GSM, we can extend our MSC to support 3G also.
- GPRS: OpenBSC is capable of supporting GPRS connection as well but the OpenBSC implementation of GPRS(SGSN+GGSN inside OpenBSC) is not fully tested according to OpenBSC wiki. So, even if we think of implementing GPRS functionality we should be prepared to make changes in the OpenBSC code as well which will require detail study of Open BSC and that can be very time consuming.
- Supporting several BSC's: As of now openMSC only support one BSC and in real world application there can be several hundreds under one MSC.
- Skipping SCCP function in CNODE(Future): CNODE is using the sccp library which is written in C and it is possible that we can create the sccp header in erlang and handle the communication with OpenBSC. So, we can skip the CNODE and have these features in erlang itself.

7 Installation and Instruction

7.1 BTS

A base transceiver station (BTS) or cell site is a piece of equipment that facilitates wireless communication between user equipment (UE) and a network. UEs are devices like mobile phones (handsets), WLL phones, computers with wireless internet connectivity, WiFi and WiMAX gadgets etc.

The ip.access nanoBTS is small BTS with an A-bis over IP interface. It's about the size of a WiFi access point.The nanoBTS picocells are complete GSM basestations that use the standard Um interface to the handset and an Abis interface carried over IP for the backhaul. Pop the basestation in. Connect to the BSC. You're up and running.

Fig 7.1 nanoBTS used during project.

The basestations offer:

- Available in 850, 900, 1800 or 1900MHz bands
- Indoor coverage up to 125,000m2
- Low-cost IP backhaul
- Simple deployment – using a single Ethernet connection for power, traffic and signaling.
- Network Listen^{TM} – supplements RF planning, allowing the planners to see into the difficult indoor environment to optimize
- Coverage and Avoid interference issues.

7.2 Configuration steps for connecting nanoBTS with your Computer

Before you connect the nano BTS to your machine you need to Install DHCP server in your system.

- sudo apt-get install dhcp3-server
- Configuring DHCP server
- If you have two network cards in your ubuntu server you need to select which interface you want to use for DHCP server listening.By default it listens to eth0. You can change this by editing /etc/default/dhcp3-server file.

- sudo vim /etc/default/dhcp3-server
- Find this line *INTERFACES="eth0"*
- Replace with the following line
  - INTERFACES="eth1"
- Save and exit.
- sudo /etc/init.d/dhcp3-server restart

The unconfigured ip.access nanoBTS needs to be configured as follows

- The BTS is configued automatically to obtain an IP address via DHCP
- The BTS is listening on UDP port 3006 for broadcast packets (e.g. should be found by ipaccess-find)
- typical response by ipaccess-find will be

  Trying to find ip.access BTS by broadcast UDP... MAC Address='00:02:95:00:16:48' IP Address='192.168.1.85' Unit ID='1801/0/0' Location 1=" Location 2='BTS_NBT131G' Equipment Version='139_029_31' Software Version='120a002_v209b24d0' Unit Name='nbts-00-02-95-00-16-48' Serial Number='00065049'
- The BTS is listening on TCP port 3006 for incoming Abis-over-IP connections. This is called Secondary OML Link
- The BTS has an unconfigured Unit ID (65535/0/0) and will refuse to work until a Unit ID has been set
- You can set the Unit ID and Primarly OML IP using ipaccess-config as follows:
- $ ./ipaccess-config -u 1801/0/0 -o 192.168.1.1 -r 192.168.1.85

The config file for nanoBTS needs to be configured as follows

- cd openbsc/openbsc/src/ open file *openbsc.cfg.nanobts* and change the value of
  - mobile network code(MNC) *28*
  - short name *OpenMSC*
  - long name *OpenMSC*
  - auth policy *accept-all*
  - auth policy *closed* represents: Don't allow anyone to network who is not marked as authorized=1 in the HLR database
- Run ./bsc_msc_ip -c openbsc.cfg.nanobts

NOTE:- At Initial stage the nanoBTS will blink with red light and after successful connection you will see the green light.

7.3 Erlang Installation

- # install libraries and tools
- sudo apt-get install fop freeglut3-dev libwxgtk2.8-dev g++ libncurses5-dev m4
- # download source code
- wget http://erlang.org/download/otp_src_R14B.tar.gz
- tar zxvf otp_src_R14B.tar.gz
- cd otp_src_R14B/
- # compile and install
- ./configure
- make
- sudo make install
- # clean
- cd ..
- rm -r otp_src_R14B
- rm otp_src_R14B.tar.gz

7.4 OpenBSC

Get libosmocore from OpenBSC github and install the libosmocore as mentioned below

- sudo apt-get install libdbi0 libdbi0-dev libdbd-sqlite3 autoconf git-core libtool pkg-config
- git clone git://git.osmocom.org/libosmocore.git
- cd libosmocore
- autoreconf -i
- ./configure
- make
- make install
- ldconfig

Get the openBSC source from github and install as mentioned below

- git clone git://bs11-abis.gnumonks.org/openbsc.git
- cd openbsc/openbsc
- autoreconf -i
- export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
- ./configure
- make
- git checkout -b remotes/origin/on-waves/bsc-master

Get libosmo sccp from OpenBSC github and follow the below steps to install it

- git://git.osmocom.org/libosmo-sccp.git
- cd libosmo-sccp
- autoreconf -i
- export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
- ./configure
- make

7.5 Wireshark

- Get the source for Wireshark
- Patch the source file of Wireshark with patches files from wireshark folder in OpenBSC
- sudo aptitude install autoconf libgtk2.0-dev libglib2.0-dev libgeoip-dev libpcre3-dev libpcap0.8-dev libtool byacc flex subversion
- cd wireshark
- ./autogen.sh
- ./configure
- make
- sudo make install

7.6 Install CouchDB Installation

Installing the couchdb is as simple as running the below command
sudo apt-get install couchdb
Then you can get it running in http://localhost:5984/_utils
To get access to CouchDB, .htaccess is needed in apache websites folder.
The following is a sample:

```
    RewriteEngine on
RewriteRule ([^]*)_design([^]*)_view([^]*) http://localhost:5984/$1/_design/$2/_view/$3 [p]
RewriteRule ([^]*)_view([^]*)(.*) http://localhost:5984/$1/_view/$2/$3 [p]
RewriteRule ([^]*)([^]*)(.*) http://localhost:5984/$1/$2\%2F$3 [p]
RewriteRule (.*) http://localhost:5984/$1 [p]
```

7.7 Making CNODE

- Go to CNODE folder
- Change the location of openBSC specified in Makefile as per your convenience
- run make

7.8 Making MSC

- Go to MSC directory
- Modify node name and cookie mentioned in parser.erl
- make

7.9 Run the system

- Run BSC
  - Goto *openbsc/openbsc/src* and run ./bsc_msc_ip
- Run CNODE
  - Goto CNODE directory and run ./cnode
- Run MSC
  - Goto MSC/ebin, run erlang node with cookie same as used in CNODE and using short name

  In erlang terminal :

  Start MSC application as ***application:start(msc)***
  no more pinging, it is automated

7.10 Using the logserver

Compile the logserver and the klib then use the following functions:
logserver:start(Filename) - To start the logserver.

- **Filename** is a string that will be the name of the file where the log will be saved. **logserver:log(From,To,Messge,Con**
  - To log a message in the server. Timestamps are added automatically by the server when the message is saved.
- **From** should be the name of the node sending the message.
- **To** should be the name of the node receiving the message.
- **Message** should be the name of the message sent between the nodes.
- **Content** can be any erlang term, perferably a string or int that represents the content of the message.

  logserver:upread() - To print the logfile that is currently used by the logserver to the terminal. **logserver:stop()** - To kill the process. If you want to print an old logfile use start(Filename) where Filename is an already existing file that you want to print. Then call logserver:upread().

8 Tools

8.1 Wireshark

Wireshark is an opensource protocol analyzer tool which is helpful for developing communication protocol and analysing, troubleshooting.

Wireshark is very similar to tcpdump, but has a graphical front-end, and many more information sorting and filtering options (although similar sorting and filtering can be achieved on the command line by combining tcpdump with grep, sort, etc.). Wireshark allows the user to see all traffic being passed over the network (usually an Ethernet network but support is being added for others) by putting the network interface into promiscuous mode.

We used the wireshark to analyse the packet flow between BSC and MSC, to debug and verify that the packets were correctly constructed. It is quite useful tool in understanding the packet flows in network and to get the idea how the packets are constructed. We used some patches from OpenBSC to enable the wireshark to track the packets related to IPA, SCCP, BSSMAP. We used the Wireshark v1.4.3 which is a freely available stable version of wireshark.

8.2 Git

Git is a distributed revision control system. It is quite useful in tracking the source versions. We used git to manage our project source and to reflect the changes made by people in the team. Git was pretty simple and fast to handle and very effective tool in managing things in a group work. We used the Git v1.7.4.1 which is a freely available stable version of Git.

8.3 CouchDB

CouchDB is a document based distributed database. It is a non relational nosql database with high performance and scalability. We used couchdb for subscriber management in VLR. The couchdb version we used was apache-couchdb 1.0.2.

8.4 Emacs

Open source text editor that offers good support for Erlang by having alot of features such as code skeletons and text high-lighting. We used it to write code. For our project we used emacs21 (21.4a-3ubuntu2.2).

8.5 Vim

Open source text editor that works fine in linux based operating systems and some of us used it to write code. For our project we used Vim version 7.3 which is latest stable version.

8.6 Eclipse

Eclipse is a software development environment which supports multiple programming languages like Ada, Java, Erlang, C etc. It comprised of integrated development environment (IDE). For our project we used Helios Version released on 23 june 2010 of Eclipse IDE for C/C++ Developers. We used this tool to utilize the OpenBSC code written in 'C'. It helped us for navigation call back functions and analyzing protocol data fields.

8.7 Ubuntu

Ubuntu is a debian GNU/Linux distribution based operating system. We used it as a platform for the development of our application. The Ubuntu version we used was 10.04.

9 Conclusion

In general, we have been able to implement an SMS capable Mobile Switching Centre (MSC) and Visitor Location Register (VLR) by following 3GPP standards. We were able to integrate our implementation with the hardware nanoBTS provided by MobileArts, open source implementation of BSC and the modules HLR, SMSC also provided by Mobile Arts to complete a GSM network. We tested our product by sending several messages at a time and checked if messages and delivery reports are delivered. We also tested with the mobile terminal switched off, in which case the mobile receives the message once it is turned on. The project was fully developed using the functional programming language erlang/OTP.

10 References

1. 3GPP GSM 04.08, Mobile radio interface layer 3 specification, 1998
2. 3GPP GSM 08.08, MSC - BSS interface layer 3 specification, 1999
3. 3GPP GSM 08.06, Signalling transport mechanism specification for BSS - MSC interface, 1999
4. 3GPP GSM 04.11, Point-to-Point Short message service support on mobile radio interface, 1996
5. 3GPP GSM 22.041, Operator Determined Barring, 2009
6. 3GPP GSM 03.03, Numbering, addressing and identification, 1998
7. 3GPP GSM 09.02, Mobile application part specification, 1998
8. ITU-T Q.712, SCCP Simply SS7, 2001-2002,SS8 Networks
9. Erlang Programming, A concurrent approach to software development, Francesco Cesarini and Simon Thompson, OReilly, 2009
10. http://openbsc.osmocom.org/trac/
11. Orebaugh, Angela; Ramirez, Gilbert; Beale, Jay (February 14, 2007). Wireshark and Ethereal Network Protocol Analyzer Toolkit. Syngress. ISBN 1597490733
12. CouchDB: The Definitive Guide by J. Chris Anderson, Jan Lehnardt, Noah SlaterPublisher:O'Reilly Media, Released: January 2010
13. A. Olsson, M. Narup, C. Helgeson, T. Eriksson, L. Lundberg, A. Lindberg, J. Carlbom, U. Vallenor, L. Bergquist, S. Johansson. Att forsta telekommunikation. Lund: Studentlitteratur, 1996.
14. TietoEnator Internal SS7 course material.
15. L. Dryburgh and J. Hewett. Signaling System No. 7 (SS7C7): Protocol, Architecture, and Services. Indianapolis USA: Cisco Press, 2005.
16. Mobile Communications (2nd Edition) (9780321123817), Jochen Schiller, Addison Wesley; 2 edition (September 21, 2003)

Overview

Module bssap

Library for encoding and decoding BSSAP packets, (not completed).

Version: '1.0'.

Authors: Tobias Vehkajarvi.

Description

Library for encoding and decoding BSSAP packets, (not completed).
Usage example:
> P = bssap:packet(id_req, [imsi]), % will create a packet record.
> B = bssap:create(P), % encodes packet into a binary.
> P = bssap:parse(B), % creates the same packet again.

Data Types

packet()

```
packet() = #packet{identifier = atom(), byte = integer(), format = [{atom(),
atom()}], arguments = [term()]}
```

Function Index

create/1	Creates a binary for a given packet.
find_packet/1	Find a packet with the name in the protocol description.
give_packets/0	Gives the list of all defined packets.
packet/2	Given a identifier and arguments it will try to create a packet.
parse/1	Parse a binary by traversing our defined protocol structure into a packet.

Function Details

create/1

```
create(Packet::packet()) -> binary()
```

Creates a binary for a given packet.

find_packet/1

```
find_packet(N::atom()) -> packet()
```

Find a packet with the name in the protocol description.

give_packets/0

```
give_packets() -> [packet()]
```

Gives the list of all defined packets.

packet/2

```
packet(N::atom(), Args::[any()]) -> binary()
```

Given a identifier and arguments it will try to create a packet.

parse/1

```
parse(Bin) -> any()
```

Parse a binary by traversing our defined protocol structure into a packet.

Overview

Generated by EDoc, Jan 14 2011, 11:01:57.

Module child_sup

Generalized child supervisor used for the different services.

Version: '1.0'.

Behaviours: `supervisor`.

Authors: openMSC.

Description

Generalized child supervisor used for the different services.

Function Index

init/1	Init function for a transient worker in a simple_one_for_one supervisor.
spawn_child/2	Spawn a child with given argument under the give supervisor name.
spawn_lm/1	Spawn a child with given argument under lm child supervisor.
spawn_mo/1	Spawn a child with given argument under mo child supervisor.
spawn_mt/1	Spawn a child with given argument under mt child supervisor.
spawn_sm/1	Spawn a child with given argument under sm child supervisor.
start_link/2	Start a new supervisor and register under the given name and this will spawn children from the given module.

Function Details

init/1

```
init(X1::[atom()]) -> {ok, any()}
```

Init function for a transient worker in a simple_one_for_one supervisor.

spawn_child/2

```
spawn_child(Sup::atom(), Args::[any()]) -> ok
```

Spawn a child with given argument under the give supervisor name.

spawn_lm/1

```
spawn_lm(Args::[any()]) -> ok
```

Spawn a child with given argument under lm child supervisor.

spawn_mo/1

```
spawn_mo(Args::[any()]) -> ok
```

Spawn a child with given argument under mo child supervisor.

spawn_mt/1

```
spawn_mt(Args::[any()]) -> ok
```

Spawn a child with given argument under mt child supervisor.

spawn_sm/1

```
spawn_sm(Args::[any()]) -> ok
```

Spawn a child with given argument under sm child supervisor.

start_link/2

```
start_link(Sup::atom(), Child::atom()) -> ok
```

Start a new supervisor and register under the given name and this will spawn children from the given module.

Overview

Module gsm0408

Mobile radio interface layer 3 specification.

Version: '1.0'.

Authors: openMSC.

Description

Mobile radio interface layer 3 specification. Parsing and creating DTAP packets.

Data Types

id_type()

```
id_type() = imsi | tmsi
```

lu_rej_cause()

```
lu_rej_cause() = integer()
```

lu_type()

```
lu_type() = {update_type, FOR::boolean(), lu_normal | lu_periodic | lu_attach}
```

rp_err_cause()

```
rp_err_cause() = integer()
```

service_type()

```
service_type() = {service_type, Type}

    Type = mocall | emergency | sms | supplementary | groupcall | broadcastcall |
    location
```

Function Index

channel_release/0	Channel release with normal cause.

cm_serv_acc/0	CM service accept.
cm_serv_rej/1	CM service reject.
encode/1	Encode the identification type.
id_req/1	Identification request.
lu_cnf/2	Location update confirm.
lu_rej/1	Location update reject.
parse/1	Parse a binary according as gsm0806:dtap and returns the packets name and parsed parameters if successful otherwise returns error with reason and the rest of the binary that didn't get parsed.
read_lai/1	Parse LAI and return the possible leftover.
tmsi_realloc_cmd/2	TMSI reallocation command.

Function Details

channel_release/0

```
channel_release() -> packet()
```

Channel release with normal cause.
Documentation: GSM 04.08 § 9.1.7

cm_serv_acc/0

```
cm_serv_acc() -> any()
```

CM service accept.
Documentation: GSM 04.08 § 9.2.5

cm_serv_rej/1

```
cm_serv_rej(Cause) -> any()
```

CM service reject.
Documentation: GSM 04.08 § 9.2.5

encode/1

```
encode(X1) -> any()
```

Encode the identification type.
Documentation: GSM 04.08 § 10.5.3.4

id_req/1

```
id_req(Id) -> any()
```

Identification request.
Documentation: GSM 04.08 § 9.2.10

lu_cnf/2

```
lu_cnf(LAI, Id) -> any()
```

Location update confirm.
Documentation: GSM 04.08 § 9.2.13

lu_rej/1

```
lu_rej(Reason) -> any()
```

Location update reject.
Documentation: GSM 04.08 § 9.2.14

parse/1

```
parse(Bin::binary()) -> {ok, atom(), [any()]} | {error, string(), binary()}
```

Parse a binary according as gsm0806:dtap and returns the packets name and parsed parameters if successful otherwise returns error with reason and the rest of the binary that didn't get parsed.

read_lai/1

```
read_lai(Bin::binary()) -> {lai(), binary()}
```

Parse LAI and return the possible leftover.
Documentation: GSM 04.08 § 10.5.1.3

tmsi_realloc_cmd/2

```
tmsi_realloc_cmd(LAI, Id) -> any()
```

TMSI reallocation command.
Documentation: GSM 04.08 § 9.2.18

Overview

Generated by EDoc, Jan 14 2011, 11:01:57.

Module gsm0411

Point-to-Point Short message service support on mobile radio interface.

Version: '1.0'.

Authors: openMSC.

Description

Point-to-Point Short message service support on mobile radio interface. Parsing and creating CP-Layer packets for SMS.

Data Types

id_type()

```
id_type() = imsi | tmsi
```

lu_rej_cause()

```
lu_rej_cause() = integer()
```

lu_type()

```
lu_type() = {update_type, FOR::boolean(), lu_normal | lu_periodic | lu_attach}
```

rp_err_cause()

```
rp_err_cause() = integer()
```

service_type()

```
service_type() = {service_type, Type}

    Type = mocall | emergency | sms | supplementary | groupcall | broadcastcall |
    location
```

Function Index

cp_ack_resp/1	CP-ACK for CP-DATA recieved from MS for MO.
cp_ack_send/1	CP-ACK for CP-DATA sent to MS for MT.
cp_error_resp/2	CP-ERROR for CP-DATA recieved from MS for MO.
cp_error_send/2	CP-ERROR for CP-DATA sent to MS for MT.
parse/1	Parse a binary according as sms and returns the packets name and parsed parameters if successful otherwise returns error with reason and the rest of the binary that did not get parsed.
rp_ack/2	RP-ACK.
rp_error/3	RP-Error that is contained within the CP-DATA.
rp_mt_data/5	RP-DATA that is contained within the CP-DATA.

Function Details

cp_ack_resp/1

```
cp_ack_resp(TIO) -> any()
```

CP-ACK for CP-DATA recieved from MS for MO.
Documentation: GSM 04.11 § 7.2.2

cp_ack_send/1

```
cp_ack_send(TIO) -> any()
```

CP-ACK for CP-DATA sent to MS for MT.
Documentation: GSM 04.11 § 7.2.2

cp_error_resp/2

```
cp_error_resp(TIO, Cause) -> any()
```

CP-ERROR for CP-DATA recieved from MS for MO.
Documentation: GSM 04.11 § 7.2.3

cp_error_send/2

```
cp_error_send(TIO, Cause) -> any()
```

CP-ERROR for CP-DATA sent to MS for MT.
Documentation: GSM 04.11 § 7.2.3

parse/1

```
parse(Bin::binary()) -> {ok, atom(), [any()]} | {error, string(), binary()}
```

Parse a binary according as sms and returns the packets name and parsed parameters if successful otherwise returns error with reason and the rest of the binary that did not get parsed.

rp_ack/2

```
rp_ack(TIO, Ref) -> any()
```

RP-ACK. that is contained within the CP-DATA.

Documentation: GSM 04.11 § 7.3.3

rp_error/3

```
rp_error(TIO, Ref, Reason) -> any()
```

RP-Error that is contained within the CP-DATA.
Documentation: GSM 04.11 § 7.3.4

rp_mt_data/5

```
rp_mt_data(TIO, MsgRef, RPOA, RPDA, RPUD) -> any()
```

RP-DATA that is contained within the CP-DATA.
Documentation: GSM 04.11 § 7.3.1.1

Overview

Module gsm0806

Signalling transport mechanism specification for BSS - MSC interface.

Version: '1.0'.

Authors: openMSC.

Description

Signalling transport mechanism specification for BSS - MSC interface. Parsing and creating BSSAP packets.

Data Types

id_type()

```
id_type() = imsi | tmsi
```

lu_rej_cause()

```
lu_rej_cause() = integer()
```

lu_type()

```
lu_type() = {update_type, FOR::boolean(), lu_normal | lu_periodic | lu_attach}
```

rp_err_cause()

```
rp_err_cause() = integer()
```

service_type()

```
service_type() = {service_type, Type}

    Type = mocall | emergency | sms | supplementary | groupcall | broadcastcall |
    location
```

Function Index

bssmap/1	Protocol header for BSSMAP.
dtap/2	Protocol header for DTAP.
parse/1	Parse a binary according as bssap and returns the packets name and parsed parameters if successful otherwise returns error with reason and the rest of the binary that didn't get parsed.

Function Details

bssmap/1

```
bssmap(Msg::[integer()]) -> packet()
```

Protocol header for BSSMAP.
Documentation: GSM 08.06 § 6.3.3

dtap/2

```
dtap(DCLI::integer(), Msg::[integer()]) -> packet()
```

Protocol header for DTAP.
Documentation: GSM 08.06 § 6.3.2

parse/1

```
parse(Bin::binary()) -> {ok, atom(), [any()]} | {error, string(), binary()}
```

Parse a binary according as bssap and returns the packets name and parsed parameters if successful otherwise returns error with reason and the rest of the binary that didn't get parsed.

Generated by EDoc, Jan 14 2011, 11:01:57.

Module gsm0808

MSC - BSS interface layer 3 specification.

Version: '1.0'.

Authors: openMSC.

Description

MSC - BSS interface layer 3 specification. Parsing and creating BSSMAP packets.

Data Types

id_type()

```
id_type() = imsi | tmsi
```

lu_rej_cause()

```
lu_rej_cause() = integer()
```

lu_type()

```
lu_type() = {update_type, FOR::boolean(), lu_normal | lu_periodic | lu_attach}
```

rp_err_cause()

```
rp_err_cause() = integer()
```

service_type()

```
service_type() = {service_type, Type}

    Type = mocall | emergency | sms | supplementary | groupcall | broadcastcall |
    location
```

Function Index

clear_cmd/0	Clear channel command.

page/1	Paging with only IMSI.
page/2	Paging with IMSI and LAI.
parse/1	Parse a binary according as bssmap and returns the packets name and parsed parameters if successful otherwise returns error with reason and the rest of the binary that didn't get parsed.
reset_ack/0	Reset acknowledgement.

Function Details

clear_cmd/0

```
clear_cmd() -> any()
```

Clear channel command.
Documentation: GSM 08.08 § 3.2.1.21

page/1

```
page(Id::imsi()) -> packet()
```

Paging with only IMSI.
Documentation: GSM 08.08 § 3.2.1.19

page/2

```
page(Id::imsi(), Area) -> packet()

     Area = lac() | lai()
```

Paging with IMSI and LAI.
Documentation: GSM 08.08 § 3.2.1.19

parse/1

```
parse(Bin::binary()) -> {ok, atom(), [any()]} | {error, string(), binary()}
```

Parse a binary according as bssmap and returns the packets name and parsed parameters if successful otherwise returns error with reason and the rest of the binary that didn't get parsed.

reset_ack/0

```
reset_ack() -> any()
```

Reset acknowledgement.
Documentation: GSM 08.08 § 3.2.1.24

Overview

Generated by EDoc, Jan 14 2011, 11:01:57.

Module lists_misc

A bunch of helper function that is used all over the project.

Version: '1.0'.

Authors: openMSC.

Description

A bunch of helper function that is used all over the project.

Function Index

binary_to_integer/1	Converts a whole binary into a big integer.
drop_last/1	Drop the last element of the list.
evenOdds/2	Splits a list into => {[1,_,3,_,5,_,7], [_,2,_,4,_,6,_]}.
int_to_binary3/1	Create a three octet binary of the integer.
int_to_intlist/1	Convert a integer to a list of each digit in the number.
int_to_octets/2	Create N number of octets of a integer.
int_to_pid/1	Convert a integer to a pid so we can call it again.
intlist_to_int/1	Convert a list of digits to a integer.
pid_to_int/1	Convert a pid to a integer that can be used as a reference in SCCP.
to_octet/2	Constructs one octet with MSB and LSB.
with_length/1	Prepend a list with the length of the list.

Function Details

binary_to_integer/1

```
binary_to_integer(Bin::binary()) -> integer()
```

Converts a whole binary into a big integer.

drop_last/1

```
drop_last(XS::list()) -> list()
```

Drop the last element of the list.

evenOdds/2

```
evenOdds(XS::list(), OddCase::any()) -> {list(), list()}
```

Splits a list into => {[1,_,3,_,5,_,7], [_,2,_,4,_,6,_]}

int_to_binary3/1

```
int_to_binary3(I::integer()) -> binary()
```

Create a three octet binary of the integer.

int_to_intlist/1

```
int_to_intlist(I::integer()) -> [integer()]
```

Convert a integer to a list of each digit in the number.

int_to_octets/2

```
int_to_octets(I::integer(), N::integer()) -> [integer()]
```

Create N number of octets of a integer.

int_to_pid/1

```
int_to_pid(I::integer()) -> pid()
```

Convert a integer to a pid so we can call it again.

intlist_to_int/1

```
intlist_to_int(XS::[integer()]) -> integer()
```

Convert a list of digits to a integer.

pid_to_int/1

```
pid_to_int(Pid::pid()) -> integer()
```

Convert a pid to a integer that can be used as a reference in SCCP.

to_octet/2

```
to_octet(MSB::integer(), LSB::integer()) -> integer()
```

Constructs one octet with MSB and LSB.

with_length/1

```
with_length(Msg::list()) -> list()
```

Prepend a list with the length of the list.

Generated by EDoc, Jan 14 2011, 11:01:57.

Module lm_child

Gen server child that does location management services.

Version: '1.0'.

Behaviours: gen_server.

Authors: openMSC.

Description

Gen server child that does location management services.

Data Types

id_type()

```
id_type() = imsi | tmsi
```

lu_rej_cause()

```
lu_rej_cause() = integer()
```

lu_type()

```
lu_type() = {update_type, FOR::boolean(), lu_normal | lu_periodic | lu_attach}
```

rp_err_cause()

```
rp_err_cause() = integer()
```

service_type()

```
service_type() = {service_type, Type}

    Type = mocall | emergency | sms | supplementary | groupcall | broadcastcall |
    location
```

Function Index

code_change/3	Convert process state when code is changed.
handle_call/3	handles all call messages.
handle_cast/2	handles a_lu_req request from the lm_server.
handle_info/2	Handling all non call/cast messages.
init/1	initates the gen server child.
start_link/5	starts a new gen server child.
terminate/2	called when the server is about to terminate.

Function Details

code_change/3

```
code_change(OldVsn::any(), State::any(), Extra::any()) -> {ok, any()}
```

Convert process state when code is changed

handle_call/3

```
handle_call(Data::any(), From::pid(), State::any()) -> {noreply, any()}
```

handles all call messages

handle_cast/2

```
handle_cast(X1::tuple(), X2::atom() | tuple()) -> Result

returns: Result = {noreply,tuple()} | {stop,normal,any()}
```

handles a_lu_req request from the lm_server

handle_info/2

```
handle_info(Info::any(), State::any()) -> {noreply, any()}
```

Handling all non call/cast messages

init/1

```
init(X1::List) -> {ok, null}

returns: List = [id(),lai(),integer(),integer(),pid()]
```

initates the gen server child

start_link/5

```
start_link(ID::tuple(), LAI::lai(), Update_type::integer(), BSCRef::integer(),
From::pid()) -> {ok, pid()}
```

starts a new gen server child

terminate/2

```
terminate(Reason::any(), State::any()) -> ok
```

called when the server is about to terminate

Generated by EDoc, Jan 14 2011, 11:01:57.

Module lm_server

Gen server that recieve location update request from bsc and spawn a gen server child process for every such request.

Version: '1.0'.

Behaviours: gen_server.

Authors: openMSC.

Description

Gen server that recieve location update request from bsc and spawn a gen server child process for every such request

Data Types

id_type()

```
id_type() = imsi | tmsi
```

lu_rej_cause()

```
lu_rej_cause() = integer()
```

lu_type()

```
lu_type() = {update_type, FOR::boolean(), lu_normal | lu_periodic | lu_attach}
```

rp_err_cause()

```
rp_err_cause() = integer()
```

service_type()

```
service_type() = {service_type, Type}

    Type = mocall | emergency | sms | supplementary | groupcall | broadcastcall |
    location
```

Function Index

a_lu_req/4	recieve location update request from bsc.
code_change/3	Convert process state when code is changed.
handle_call/3	handles call mesaages from bsc.
handle_cast/2	handles cast messages.
handle_info/2	Handling all non call/cast messages.
init/1	initates the gen server child.
start_link/0	starts a new gen server child.
terminate/2	called when the server is about to terminate.

Function Details

a_lu_req/4

```
a_lu_req(Update_type::integer(), LAI::lai(), ID::id(), BSCRef::integer()) ->
{ok, pid()}
```

recieve location update request from bsc

code_change/3

```
code_change(OldVsn::any(), State::any(), Extra::any()) -> {ok, any()}
```

Convert process state when code is changed

handle_call/3

```
handle_call(X1::Tuple, From::pid(), X3::null) -> {noreply, null}

returns: {Tuple = a_lu_req, id(), lai(),integer(),integer()}
```

handles call mesaages from bsc

handle_cast/2

```
handle_cast(Data::any(), State::any()) -> {noreply, any()}
```

handles cast messages

handle_info/2

```
handle_info(Info::any(), State::any()) -> {noreply, any()}
```

Handling all non call/cast messages

init/1

```
init(X1::[]) -> {ok, null}
```

initates the gen server child

start_link/0

```
start_link() -> {ok, pid()}
```

starts a new gen server child

terminate/2

```
terminate(Reason::any(), Loop_data::any()) -> ok
```

called when the server is about to terminate

Generated by EDoc, Jan 14 2011, 11:01:57.

Module mbinterface

The interface used between MSC and BSC.

Version: '1.0'.

Authors: Projectgroup number 55..

Description

The interface used between MSC and BSC. Created : Yes.

Data Types

bscref()

```
bscref() = integer()
```

cell_info()

```
cell_info() = {lai(), ci()} | {{lac, integer()}, ci()} | {undefined, ci()}
```

ci()

```
ci() = {ci, integer()}
```

id()

```
id() = tmsi() | imsi()
```

id_type()

```
id_type() = imsi | tmsi
```

imsi()

```
imsi() = {imsi, integer()}
```

lai()

```
lai() = {lai, integer()}
```

lu_rej_cause()

```
lu_rej_cause() = integer()
```

lu_type()

```
lu_type() = {update_type, FOR::boolean(), lu_normal | lu_periodic | lu_attach}
```

mscref()

```
mscref() = integer()
```

octets()

```
octets() = [integer()]
```

packet()

```
packet() = binary()
```

rp_err_cause()

```
rp_err_cause() = integer()
```

service_type()

```
service_type() = {service_type, Type}

    Type = mocall | emergency | sms | supplementary | groupcall | broadcastcall |
    location
```

tmsi()

```
tmsi() = {tmsi, integer()}
```

Function Index

channel_release/1	(MSC -> BSC) Channel release.
classmark_change/2	(BSC -> MSC) Classmark change.
clear_cmd/1	(MSC -> BSC) Clear command.
clear_cmp/1	(BSC -> MSC) Clear request.
clear_req/2	(BSC -> MSC) Clear request.
cp_ack_resp/2	(MSC -> BSC) CP-ACK for a CP-DATA for MO.
cp_ack_send/2	(MSC -> BSC) CP-ACK for a CP-DATA for MT.
cp_error_resp/3	(MSC -> BSC) CP-ERROR for a CP-DATA for MO.
cp_error_send/3	(MSC -> BSC) CP-ERROR for a CP-DATA for MT.
id_req/2	(MSC -> BSC) Handle ID request.
id_rsp/2	(BSC -> MSC) Handle ID Reponse.
imsi_detach_ind/4	(BSC -> MSC) (CR) Imsi detach indication.
lu_cnf/3	(MSC -> BSC) Handle Location Update confirmation.

lu_rej/2	(MSC -> BSC) Handle Location Update Reject.
lu_req/7	(BSC -> MSC) (CR) Handle location Update Request.
mo_cm_service_acc/1	(MSC -> BSC) Handle mo cm service accept.
mo_cm_service_rej/2	(MSC -> BSC) Handle mo cm service reject.
mo_cm_service_req/5	
mo_cm_service_req/6	(BSC -> MSC) (CR) Handle mo cm service request.
page_req/1	(MSC -> BSC) (UDT) Handle page request with only imsi.
page_req/2	(MSC -> BSC) (UDT) Handle page reques with imsi and (lai or lac).
page_rsp/5	(BSC -> MSC) (CR) Handle page response.
reset/1	(BSC -> MSC) Reset.
reset_ack/0	(MSC -> BSC) Reset acknowledge.
rp_mo_ack/3	(MSC -> BSC) Handle rp mo acknowledge.
rp_mo_data/6	(BSC -> MSC) Handle mo rp mo data.
rp_mo_error/4	(MSC -> BSC) Handle rp mo error.
rp_mt_ack/3	(BSC -> MSC) Handle rp acknowledge.
rp_mt_data/6	(MSC -> BSC) Handle RP MT data.
rp_mt_error/4	(BSC -> MSC) Handle rp Error.
rp_smma/3	(BSC -> MSC) Handle rp short message memory available.
sccp_cc/2	(MSC -> BSC) Sends CC (Connection confirm) to the BSC as a response to CR (Connection request).
sccp_cref/1	(MSC -> BSC) Sends CREF (Connection Refuse) to the BSC as a response CR (Connection request).
sccp_rlsd/2	(MSC -> BSC) Sends RLSD (Connection Release) to the BSC.
tmsi_realloc_cmd/3	(MSC -> BSC) Handle TMSI Reallocation.
tmsi_realloc_comp/1	(BSC -> MSC) TMSI Reallocation complete.

Function Details

channel_release/1

```
channel_release(DLR::bscref()) -> ok
```

(MSC -> BSC) Channel release.
Documentation: GSM 04.08 § 9.1.7

classmark_change/2

```
classmark_change(DLR::mscref(), Classmark2::classmark2()) -> ok
```

(BSC -> MSC) Classmark change.
Documentation: GSM 04.08 § 9.1.11

clear_cmd/1

```
clear_cmd(DLR) -> any()
```

(MSC -> BSC) Clear command.

Documentation: GSM 08.08 § 3.2.1.21

clear_cmp/1

```
clear_cmp(DLR) -> any()
```

(BSC -> MSC) Clear request.
Documentation: GSM 08.08 § 3.2.1.22

clear_req/2

```
clear_req(DLR, Cause) -> any()
```

(BSC -> MSC) Clear request.
Documentation: GSM 08.08 § 3.2.1.20

cp_ack_resp/2

```
cp_ack_resp(DLR::bscref(), TIO::integer()) -> ok
```

(MSC -> BSC) CP-ACK for a CP-DATA for MO.
Documentation: GSM 04.11 § 7.2.2

cp_ack_send/2

```
cp_ack_send(DLR::bscref(), TIO::integer()) -> ok
```

(MSC -> BSC) CP-ACK for a CP-DATA for MT.
Documentation: GSM 04.11 § 7.2.2

cp_error_resp/3

```
cp_error_resp(DLR::bscref(), TIO::integer(), Cause::integer()) -> ok
```

(MSC -> BSC) CP-ERROR for a CP-DATA for MO.
Documentation: GSM 04.11 § 7.2.3

cp_error_send/3

```
cp_error_send(DLR::bscref(), TIO::integer(), Cause::integer()) -> ok
```

(MSC -> BSC) CP-ERROR for a CP-DATA for MT.
Documentation: GSM 04.11 § 7.2.3

id_req/2

```
id_req(DLR::bscref(), Type::id_type()) -> ok
```

(MSC -> BSC) Handle ID request.
Documentation: GSM 04.08 § 9.2.10

id_rsp/2

```
id_rsp(DLR::mscref(), Id::id()) -> ok
```

(BSC -> MSC) Handle ID Reponse.
Documentation: GSM 04.08 § 9.2.11

imsi_detach_ind/4

```
imsi_detach_ind(SLR::bscref(), Cell_info::cell_info(), Classmark::classmark(),
Id::id()) -> ok
```

(BSC -> MSC) (CR) Imsi detach indication.
Documentation: GSM 04.08 § 9.2.14

lu_cnf/3

```
lu_cnf(DLR::bscref(), LAI::lai(), Id::tmsi()) -> ok
```

(MSC -> BSC) Handle Location Update confirmation.
Documentation: GSM 04.08 § 9.2.13

lu_rej/2

```
lu_rej(DLR::bscref(), Cause::lu_rej_cause()) -> ok
```

(MSC -> BSC) Handle Location Update Reject.
Documentation: GSM 04.08 § 9.2.14

lu_req/7

```
lu_req(SLR::bscref(), Cell_info::cell_info(), Cipher::cipher(),
Type::lu_type(), Prev_LAI::lai(), Classmark::classmark(), ID::id()) -> ok
```

(BSC -> MSC) (CR) Handle location Update Request. And notice that the current LAI is in the Cell info.
Documentation: GSM 04.08 § 9.2.15

mo_cm_service_acc/1

```
mo_cm_service_acc(DLR::bscref()) -> ok
```

(MSC -> BSC) Handle mo cm service accept.
Documentation: GSM 04.08 § 9.2.5

mo_cm_service_rej/2

```
mo_cm_service_rej(DLR::bscref(), Cause::integer()) -> ok
```

(MSC -> BSC) Handle mo cm service reject.
Documentation: GSM 04.08 § 9.2

mo_cm_service_req/5

```
mo_cm_service_req(SLR, Cipher, Type, Classmark, ID) -> any()
```

mo_cm_service_req/6

```
mo_cm_service_req(SLR::bscref(), Cell_info::cell_info(), Cipher::cipher(),
Type::lu_type(), Classmark::classmark2(), ID::id()) -> ok
```

(BSC -> MSC) (CR) Handle mo cm service request.
Documentation: GSM 04.08 § 9.2.9

page_req/1

```
page_req(Imsi::imsi()) -> ok
```

(MSC -> BSC) (UDT) Handle page request with only imsi.
Documentation: GSM 08.08 § 3.2.1.19

page_req/2

```
page_req(Imsi::imsi(), Area) -> ok

    Area = lac() | lai()
```

(MSC -> BSC) (UDT) Handle page reques with imsi and (lai or lac).
Documentation: GSM 08.08 § 3.2.1.19

page_rsp/5

```
page_rsp(SLR::bscref(), Cell_info::cell_info(), Cipher::cipher(),
Classmark::classmark2(), Id::id()) -> ok
```

(BSC -> MSC) (CR) Handle page response.
Documentation: GSM 04.08 § 9.1.25

reset/1

```
reset(Cause::cause()) -> ok
```

(BSC -> MSC) Reset.

reset_ack/0

```
reset_ack() -> ok
```

(MSC -> BSC) Reset acknowledge.

rp_mo_ack/3

```
rp_mo_ack(DLR::bscref(), TIO::integer(), Ref::integer()) -> ok
```

(MSC -> BSC) Handle rp mo acknowledge.

rp_mo_data/6

```
rp_mo_data(DLR::mscref(), TIO::integer(), Ref::integer(), OA::oa(), DA::da(),
UD::binary()) -> ok
```

(BSC -> MSC) Handle mo rp mo data.

rp_mo_error/4

```
rp_mo_error(DLR::bscref(), TIO::integer(), Cause::rp_err_cause(),
Ref::integer()) -> ok
```

(MSC -> BSC) Handle rp mo error.

rp_mt_ack/3

```
rp_mt_ack(DLR::mscref(), TIO::integer(), Ref::integer()) -> ok
```

(BSC -> MSC) Handle rp acknowledge.

rp_mt_data/6

```
rp_mt_data(DLR::bscref(), TIO::integer(), OA::oa(), DA::da(), UD::binary(),
Ref::integer()) -> ok
```

(MSC -> BSC) Handle RP MT data. OA is the sc_addr but not sure. Documentation: GSM 04.11 § TODO

rp_mt_error/4

```
rp_mt_error(DLR::mscref(), TIO::integer(), Cause::rp_err_cause(),
Ref::integer()) -> ok
```

(BSC -> MSC) Handle rp Error.

rp_smma/3

```
rp_smma(DLR::mscref(), TIO::integer(), Ref::integer()) -> ok
```

(BSC -> MSC) Handle rp short message memory available.

sccp_cc/2

```
sccp_cc(SLR::bscref(), DLR::mscref()) -> ok
```

(MSC -> BSC) Sends CC (Connection confirm) to the BSC as a response to CR (Connection request).
Documentation: ITU-T RecQ.713 4.3

sccp_cref/1

```
sccp_cref(SLR::bscref()) -> ok
```

(MSC -> BSC) Sends CREF (Connection Refuse) to the BSC as a response CR (Connection request).
Documentation: ITU-T RecQ.713 4.4

sccp_rlsd/2

```
sccp_rlsd(SLR::bscref(), DLR::mscref()) -> ok
```

(MSC -> BSC) Sends RLSD (Connection Release) to the BSC.
Documentation: ITU-T RecQ.713 4.5

tmsi_realloc_cmd/3

```
tmsi_realloc_cmd(DLR::bscref(), LAI::lai(), TMSI::id()) -> ok
```

(MSC -> BSC) Handle TMSI Reallocation.
Documentation: GSM 04.08 § 9.2.17

tmsi_realloc_comp/1

```
tmsi_realloc_comp(DLR::mscref()) -> ok
```

(BSC -> MSC) TMSI Reallocation complete.
Documentation: GSM 04.08 § 9.2.18

Overview

Generated by EDoc, Jan 14 2011, 11:01:57.

Module mo_child

Gen server child that forwards the mobile originated short message and alerts the SC when memory is available in the mobile station.

Version: '1.0'.

Authors: openMSC.

Description

Gen server child that forwards the mobile originated short message and alerts the SC when memory is available in the mobile station.

Data Types

id_type()

```
id_type() = imsi | tmsi
```

lu_rej_cause()

```
lu_rej_cause() = integer()
```

lu_type()

```
lu_type() = {update_type, FOR::boolean(), lu_normal | lu_periodic | lu_attach}
```

rp_err_cause()

```
rp_err_cause() = integer()
```

service_type()

```
service_type() = {service_type, Type}

    Type = mocall | emergency | sms | supplementary | groupcall | broadcastcall |
    location
```

Function Index

code_change/3	Convert process state when code is changed.
handle_call/3	handles call messages.
handle_cast/2	handles cm_serv_req request from the mo_server.
handle_info/2	Handling all non call/cast messages.
init/1	initates the gen server child.
start_link/3	starts a new gen server child.
start_link/5	starts a new gen server child.
terminate/2	called when the server is about to terminate.

Function Details

code_change/3

```
code_change(OldVsn::any(), State::any(), Extra::any()) -> {ok, any()}
```

Convert process state when code is changed

handle_call/3

```
handle_call(A::any(), B::any(), State::any()) -> {noreply, any()}
```

handles call messages

handle_cast/2

```
handle_cast(X1::tuple(), X2::atom() | tuple()) -> Result

returns: Result = {noreply,tuple()} | {stop,normal,any()}
```

handles cm_serv_req request from the mo_server

handle_info/2

```
handle_info(Info::any(), State::any()) -> {noreply, any()}
```

Handling all non call/cast messages

init/1

```
init(X1::List) -> {ok, null}

returns: List = [id(),integer(),lai(),ci(),bscref()]
```

initates the gen server child

start_link/3

```
start_link(ID::id(), CM_serv_type::integer(), DLR::bscref()) -> {ok, pid()}
```

starts a new gen server child

start_link/5

```
start_link(ID::id(), CM_serv_type::integer(), CLAI::lai(),
Serving_cell_id::ci(), DLR::bscref()) -> {ok, pid()}
```

starts a new gen server child

terminate/2

```
terminate(Reason::any(), State::any()) -> ok
```

called when the server is about to terminate

Overview

Generated by EDoc, Jan 14 2011, 11:01:57.

Module mo_server

Gen server that recieve mobile originating short message request from bsc and spawn a gen server child process for every such request.

Version: '1.0'.

Behaviours: gen_server.

Authors: openMSC.

Description

Gen server that recieve mobile originating short message request from bsc and spawn a gen server child process for every such request.

Data Types

id_type()

```
id_type() = imsi | tmsi
```

lu_rej_cause()

```
lu_rej_cause() = integer()
```

lu_type()

```
lu_type() = {update_type, FOR::boolean(), lu_normal | lu_periodic | lu_attach}
```

rp_err_cause()

```
rp_err_cause() = integer()
```

service_type()

```
service_type() = {service_type, Type}

    Type = mocall | emergency | sms | supplementary | groupcall | broadcastcall |
    location
```

Function Index

cm_service_req/3	cm service request from bsc.
cm_service_req/5	cm service request from bsc.
code_change/3	Convert process state when code is changed.
handle_call/3	handles call messages.
handle_cast/2	handles cast mesaages from bsc.
handle_info/2	Handling all non call/cast messages.
init/1	initates the gen server child.
start_link/0	starts a new gen server child.
terminate/2	called when the server is about to terminate.

Function Details

cm_service_req/3

```
cm_service_req(DLR::bscref(), CM_serv_type::integer(), ID::id()) -> {ok,
pid()}
```

cm service request from bsc

cm_service_req/5

```
cm_service_req(DLR::bscref(), CM_serv_type::integer(), CLAI::lai(), ID::id(),
Serving_cell_id::ci()) -> {ok, pid()}
```

cm service request from bsc

code_change/3

```
code_change(OldVsn::any(), State::any(), Extra::any()) -> {ok, any()}
```

Convert process state when code is changed

handle_call/3

```
handle_call(A::any(), B::any(), State::any()) -> {noreply, any()}
```

handles call messages

handle_cast/2

```
handle_cast(X1::Tuple, State::null) -> {noreply, null}

returns: {Tuple = access_request, id(),integer(),lai(),ci(),bscref()}
```

handles cast mesaages from bsc

handle_info/2

```
handle_info(Info::any(), State::any()) -> {noreply, any()}
```

Handling all non call/cast messages

init/1

```
init(X1::[]) -> {ok, null}
```

initates the gen server child

start_link/0

```
start_link() -> {ok, pid()}
```

starts a new gen server child

terminate/2

```
terminate(Reason::any(), State::any()) -> ok
```

called when the server is about to terminate

Overview

Generated by EDoc, Jan 14 2011, 11:01:57.

Module msc_map_api

The API to the MAP node for communication with SMSC and HLR.

Version: '1.0'.

Authors: openMSC.

Description

The API to the MAP node for communication with SMSC and HLR.
Documentation: 3GPP GSM 09.02, Mobile application part specification.

Function Index

cancel_location/3	(HLR -> MSC) Cancel location,.
delete_subscriber_data/2	(HLR -> MSC) Delete subscriber data about a IMSI in VLR.
insert_subscriber_data/3	(HLR -> MSC) Insert subscriber data about a IMSI in VLR.
insert_subscriber_data/4	(HLR -> MSC) Insert subscriber data about a IMSI in VLR.
mo_forward_ack/1	(SMSC -> MSC) MO forward acknowledge, sms is recieved in smsc.
mt_forward/5	(SMSC -> MSC) MT forward, sms is going to be sent.
purge_ms_ack/1	(HLR -> MSC) Purge VLR information in HLR for a IMSI.
ready_for_sms_ack/1	(HLR -> MSC) Ready for sms acknowledge.
update_location_ack/1	(HLR -> MSC) Update location acknowledge,.

Function Details

cancel_location/3

```
cancel_location(Imsi, Lmsi, Ref) -> any()
```

(HLR -> MSC) Cancel location,
Documentation: GSM 09.02 § 8.1.3

delete_subscriber_data/2

```
delete_subscriber_data(List_args, Ref) -> any()
```

(HLR -> MSC) Delete subscriber data about a IMSI in VLR.
Documentation: GSM 09.02 § 8.8.2

insert_subscriber_data/3

```
insert_subscriber_data(Imsi, List_args, Ref) -> any()
```

(HLR -> MSC) Insert subscriber data about a IMSI in VLR.
Documentation: GSM 09.02 § 8.8.1

insert_subscriber_data/4

```
insert_subscriber_data(Imsi, Msisdn, List_args, Ref) -> any()
```

(HLR -> MSC) Insert subscriber data about a IMSI in VLR.
Documentation: GSM 09.02 § 8.8.1

mo_forward_ack/1

```
mo_forward_ack(Message) -> any()
```

(SMSC -> MSC) MO forward acknowledge, sms is recieved in smsc.
Documentation: GSM 09.02 § 12.2

mt_forward/5

```
mt_forward(Dest_addr, Sc_addr, Smrpui, More_msg, Ref) -> any()
```

(SMSC -> MSC) MT forward, sms is going to be sent.
Documentation: GSM 09.02 § 12.9

purge_ms_ack/1

```
purge_ms_ack(X1) -> any()
```

(HLR -> MSC) Purge VLR information in HLR for a IMSI.
Documentation: GSM 09.02 § 8.1.6

ready_for_sms_ack/1

```
ready_for_sms_ack(X1) -> any()
```

(HLR -> MSC) Ready for sms acknowledge
Documentation: GSM 09.02 § 12.4

update_location_ack/1

```
update_location_ack(X1) -> any()
```

(HLR -> MSC) Update location acknowledge,
Documentation: GSM 09.02 § 8.1.2

Module mt_child

A process which controls the behaviour of mobile termination.

Version: '1.0'

Behaviours: gen_fsm.

Authors: openMSC.

Description

A process which controls the behaviour of mobile termination.

Data Types

bscref()

```
bscref() = integer()
```

cell_info()

```
cell_info() = {lai() | lac() | undefined, ci()}
```

dest()

```
dest() = integer()
```

imsi()

```
imsi() = {imsi, integer()}
```

lai()

```
lai() = {lai, integer()}
```

more_msg()

```
more_msg() = atom()
```

ref()

```
ref() = pid()
```

sc()

```
sc() = integer()
```

smrpui()

```
smrpui() = binary()
```

Function Index

code_change/4	Convert process state when code is changed.
handle_event/3	
handle_info/3	
handle_sync_event/4	
idle/2	handle mt_forward sent from init function.
init/1	Whenever a gen_fsm is started using gen_fsm:start/[3,4] or gen_fsm:start_link/3,4, this function is called by the new process to initialize.
start_link/6	Creates a gen_fsm process which calls Module:init/1 to initialize.
terminate/3	This function is called by a gen_fsm when it is about to terminate.
wf_answer_from_hlr/2	next state of wf_search_rsp, wf_page_rsp or wf_answers_from_hlr.
wf_answer_from_vlr/2	next state of wf_search_rsp,wf_page_rsp, wf_answer_from_hlr and wf_answers_from_hlr.
wf_answers_from_hlr/2	next state of wf_search_rsp or wf_page_rsp.
wf_more_msg/2	next state of wf_sms_cnf.
wf_page/2	next state of idle.
wf_page_rsp/2	next state of wf_page.
wf_search_rsp/2	next state of wf_page.
wf_sms_cnf/2	next state of wf_answer_from_vlr.

Function Details

code_change/4

```
code_change(OldVsn, StateName, StateData::State, Extra) -> {ok, StateName,
NewState}
```

Convert process state when code is changed

handle_event/3

```
handle_event(Event, StateName, StateData) -> any()
```

handle_info/3

```
handle_info(Info, StateName, StateData) -> any()
```

handle_sync_event/4

```
handle_sync_event(Event, From, StateName, StateData) -> any()
```

idle/2

```
idle(X1::{mt_forward, dest(), sc(), smrpui(), more_msg(), ref()}, State) ->
{next_state, wf_page, NextState}
```

handle mt_forward sent from init function

init/1

```
init(X1::list(dest(), sc(), smrpui(), more_msg(), pid(), from())) -> {ok,
idle, NewState}
```

Whenever a gen_fsm is started using gen_fsm:start/[3,4] or gen_fsm:start_link/3,4, this function is called by the new process to initialize.

start_link/6

```
start_link(Dest_addr::dest(), Sc_addr::sc(), Smrpui::smrpui(),
More_msg::more_msg(), Ref::pid(), From::from()) -> {ok, Pid}
```

Creates a gen_fsm process which calls Module:init/1 to initialize. To ensure a synchronized start-up procedure, this function does not return until Module:init/1 has returned.

terminate/3

```
terminate(X1::normal | shutdown, X2::any(), X3::any()) -> ok
```

This function is called by a gen_fsm when it is about to terminate. It should be the opposite of Module:init/1 and do any necessary cleaning up. When it returns, the gen_fsm terminates with Reason. The return value is ignored.

wf_answer_from_hlr/2

```
wf_answer_from_hlr(X1::{update_location_ack | ready_for_sms_ack, ok} |
{update_location_ack | ready_for_sms_ack, error(), reason()}, State) ->
{next_state, wf_answer_from_vlr, NewState}
```

next state of wf_search_rsp, wf_page_rsp or wf_answers_from_hlr. Handle message from hlr. Messages include update_location_ack or ready_for_sms_ack. When finished, go to wf_answer_from_vlr.

wf_answer_from_vlr/2

```
wf_answer_from_vlr(X1::{send_info_for_mt_sms_ack, any()}, State) ->
{next_state, wf_sms_cnf, NewState, Timeout} | {stop, normal, State}
```

next state of wf_search_rsp,wf_page_rsp, wf_answer_from_hlr and wf_answers_from_hlr. If everything goes fine send msg to ms, and goes to wf_sms_cnf state, otherwise terminate.

wf_answers_from_hlr/2

```
wf_answers_from_hlr(X1::{update_location_ack | ready_for_sms_ack, ok} |
{update_location_ack | ready_for_sms_ack, error(), reason()}, State) ->
{next_state, wf_answer_from_hlr, NewState} | {next_state, wf_answer_from_vlr,
```

```
NextState}
```

next state of wf_search_rsp or wf_page_rsp. Handle messages from hlr. Messages include update_location_ack and ready_for_sms_ack. When finished, go to wf_answer_from_hlr or wf_answer_from_vlr.

wf_more_msg/2

```
wf_more_msg(X1::{mt_forward, dest(), sc(), smrpui(), more_msg(), ref()} |
timeout, State) -> {next_state, wf_sms_cnf, NewState} | {stop, normal, State}
```

next state of wf_sms_cnf. If more_msg flag is set to true by smsc, which means there are more msgs to be sent to the ms, it will come to this state, waiting for more mt_forward from the smsc. It will return to wf_sms_cnf after the msg is sent to the ms. Timeout after 30 secs.

wf_page/2

```
wf_page(X1::Event, State) -> {stop, normal, NewState} | {next_state,
wf_page_rsp, NextState, Timeout} | {next_state, wf_search_rsp, NextState,
Timeout}

     Event = {send_info_for_mt_sms_ack, tuple()} | {page, imsi(), lai()} |
     {search_for_ms, imsi()}
```

next state of idle. Send page_req to the ms to get mobile's location

wf_page_rsp/2

```
wf_page_rsp(X1::{page_rsp, imsi(), lai(), cell_info(), bscref()} | timeout,
State) -> {next_state, wf_answer_from_vlr, NewState} | {next_state,
wf_answer_from_hlr, NewState} | {next_state, wf_answers_from_hlr, NewState}
```

next state of wf_page. Handle page_rsp from the ms. Timeout after 30 sec. If send update_location and ready_for_sms to hlr in this state, it goes to wf_answers_from_hlr, otherwise it goes to wf_answer_from_hlr. If it doesn't send messages to hlr, it goes to wf_answer_from_vlr.

wf_search_rsp/2

```
wf_search_rsp(X1::tuple(page_rsp, imsi(), lai(), cell_info(), bscref()) |
timeout, State) -> {next_state, wf_answer_from_vlr, NewState} | {next_state,
wf_answer_from_hlr, NewState} | {next_state, wf_answers_from_hlr, NewState}
```

next state of wf_page. Handle page_rsp from the ms. Timeout after 30 sec. If send update_location and ready_for_sms to hlr in this state, it goes to wf_answers_from_hlr, otherwise it goes to wf_answer_from_hlr. If it doesn't send messages to hlr, it goes to wf_answer_from_vlr.

wf_sms_cnf/2

```
wf_sms_cnf(X1::Event, State) -> {stop, normal, State} | {next_state,
wf_more_msg, State, Timeout}

     Event = {rp_mt_ack, ref()} | {rp_mt_error, cause(), ref()} | timeout
```

next state of wf_answer_from_vlr. Handle msg from ms after sendig rp_data. If ms sends rp_mt_ack, the msg sent to the ms is confirmed. If there is no more msgs to be sent to the ms, send cp_ack and release the sccp connection. If there are more msgs to be sent to the ms,

goto wf_more_msg state. In both case, send mt_forward_ack to the mapp node. Timeout after 1 min.

Generated by EDoc, Jan 14 2011, 11:01:57.

Module mt_server

A gen_server receiving requests from MAPP node and page_rsp from MS and cast them to the right mt_child process.

Version: '1.0'

Behaviours: gen_server.

Authors: openMSC.

Description

A gen_server receiving requests from MAPP node and page_rsp from MS and cast them to the right mt_child process.

Data Types

bscref()

```
bscref() = integer()
```

ci()

```
ci() = {ci, integer()}
```

dest()

```
dest() = integer()
```

imsi()

```
imsi() = integer()
```

more_msg()

```
more_msg() = true | false
```

reason()

```
reason() = any()
```

ref()

```
ref() = pid()
```

sc()

```
sc() = integer()
```

smrpui()

```
smrpui() = binary()
```

state()

```
state() = any()
```

Function Index

add_pid/3	Add pid with destination address and more_msg flag to the ets table when a mt_child spawned.
code_change/3	Convert process state when code is changed.
del_pid/1	Delete pid and other configuration to the table when a mt_child died.
handle_call/3	Handle mt_forward message from ss7 stack.
handle_cast/2	Handle cast msgs such as stop, add_pid, del_pid and page_rsp.
handle_info/2	Trap Exit message.
init/1	Initiate the server.
mt_forward/5	Receive mt_forward from smsc.
page_rsp/4	Lookup the pids with same destination address when receiving page_rsp, and send page_rsp msg to each of these pids.
start_link/0	Starts the server.
stop/0	Stop the server.
terminate/2	Terminate the server.

Function Details

add_pid/3

```
add_pid(Imsi::imsi(), Pid::pid(), More_Msg::more_msg()) -> ok
```

Add pid with destination address and more_msg flag to the ets table when a mt_child spawned

code_change/3

```
code_change(OldVsn::any(), State::any(), Extra::any()) -> {ok, any()}
```

Convert process state when code is changed

del_pid/1

```
del_pid(Pid::pid()) -> ok
```

Delete pid and other configuration to the table when a mt_child died

handle_call/3

```
handle_call(X1::{mt_forward, dest(), sc(), smrpui(), more_msg, ref()},
From::any(), LoopData::any()) -> {noreply, any()}
```

Handle mt_forward message from ss7 stack. Check ets table with same destination address and is waiting for msgs. If any, send msg to the corresponding pid, otherwise, create a new mt_child.

handle_cast/2

```
handle_cast(X1::Msg, LoopData::any()) -> {stop, normal, any()} | {noreply,
any()}

     Msg = stop | {add_pid, imsi(), pid(), more_msg()} | {del_pid, pid()} |
     {page_rsp, id(), lai(), ci(), bscref()}
```

Handle cast msgs such as stop, add_pid, del_pid and page_rsp

handle_info/2

```
handle_info(X1::{'EXIT', pid(), reason()}, LoopData::any()) -> {noreply,
any()}
```

Trap Exit message

init/1

```
init(X1::atom()) -> {ok, []}
```

Initiate the server

mt_forward/5

```
mt_forward(Dest_addr::dest(), Sc_addr::sc(), Smrpui::smrpui(),
More_msg::more_msg(), Ref::ref()) -> {ok, Pid} | ignore | {error, Error}
```

Receive mt_forward from smsc

page_rsp/4

```
page_rsp(ID::imsi(), LAI::lai(), CI::ci(), BSCRef::bscref()) -> ok
```

Lookup the pids with same destination address when receiving page_rsp, and send page_rsp msg to each of these pids

start_link/0

```
start_link() -> {ok, Pid} | ignore | {error, Error}
```

Starts the server

stop/0

```
stop() -> {ok, Pid} | ignore | {error, Error}
```

Stop the server

terminate/2

```
terminate(Reason::reason(), State::state()) -> {reason(), state()}
```

Terminate the server. Delete the server's ets table.

Overview

Generated by EDoc, Jan 14 2011, 11:01:57.

Module parse

Helper functions for parsing.

Version: '1.0'.

Authors: openMSC.

Description

Helper functions for parsing.

Function Index

applys/2	Recursivly applies a parsing function and continues to parse if there is something more left.
digits/2	Parse a list of numbers from N number of octets and return possible leftover.
digits_swapped/2	Parse a list in of numbers in swapped order from N number of octets and return possible leftover.
octet_digits/3	Parse N number of octets and treat each octet as a pair of digits.

Function Details

applys/2

```
applys(Bin::binary(), Fs::[Fun]) -> {[any()], binary()}

     Fun = 'fun'((binary()) -> {term(), binary()})
```

Recursivly applies a parsing function and continues to parse if there is something more left.

digits/2

```
digits(Bin::binary(), N::integer()) -> {[integer()], binary()}
```

Parse a list of numbers from N number of octets and return possible leftover.
Example: 16#50, 16#23 => [5, 0, 2, 3].

digits_swapped/2

```
digits_swapped(Bin::binary(), N::integer()) -> {[integer()], binary()}
```

Parse a list in of numbers in swapped order from N number of octets and return possible leftover.
Example: 16#50, 16#23 => [0, 5, 3, 2].

octet_digits/3

```
octet_digits(Bin::binary(), N::integer(), F) -> {[integer()], binary()}

     F = 'fun'([list()]) -> list())
```

Parse N number of octets and treat each octet as a pair of digits.
Example: 16#50, 16#23 => [[5, 0], [2, 3]].

Overview

Generated by EDoc, Jan 14 2011, 11:01:57.

Module parser

Behaviours: gen_server.

Function Index

code_change/3	
conn/0	Initialize the connection between msc and bsc by sending pid of parser to cnode.
handle_call/3	
handle_cast/2	
handle_info/2	
handle_parse/3	Applies the parsed arguments to a function in mbinterface depending on which sccp protocol class that is given.
init/1	
ping/0	Send ping to bsc.
pong/0	Send pong to bsc.
start_link/0	
terminate/2	

Function Details

code_change/3

```
code_change(OldVsn, State, Extra) -> any()
```

conn/0

```
conn() -> ok
```

Initialize the connection between msc and bsc by sending pid of parser to cnode.

handle_call/3

```
handle_call(X1, X2, State) -> any()
```

handle_cast/2

```
handle_cast(X1, State) -> any()
```

handle_info/2

```
handle_info(X1, State) -> any()
```

handle_parse/3

```
handle_parse(Bin::binary(), SLR::binary(), Prot::integer()) -> ok
```

Applies the parsed arguments to a function in mbinterface depending on which sccp protocol class that is given.

init/1

```
init(X1) -> any()
```

ping/0

```
ping() -> any()
```

Send ping to bsc.

pong/0

```
pong() -> any()
```

Send pong to bsc.

start_link/0

```
start_link() -> any()
```

terminate/2

```
terminate(Reason, State) -> any()
```

Overview

Module sccp

A small (and in no way complete) implementation of the SCCP protocol.

Version: '1.0'.

Authors: openMSC.

Description

A small (and in no way complete) implementation of the SCCP protocol.

Function Index

create_cc/2	Connection confirm with source local reference and destination local reference as parameters.
create_cref/1	Connection refuse with only the source local reference.
create_rlsd/2	Connection release with source local reference and destination local reference as parameters.
prepend_ipaccess_header/1	Prepend the ipaccess header to the sccp message.

Function Details

create_cc/2

```
create_cc(SLR::bscref(), DLR::mscref()) -> packet()
```

Connection confirm with source local reference and destination local reference as parameters.
Documentation: ITU-T RecQ.713 4.3

create_cref/1

```
create_cref(SLR::bscref()) -> packet()
```

Connection refuse with only the source local reference.
Documentation: ITU-T RecQ.713 4.4

create_rlsd/2

```
create_rlsd(SLR::bscref(), DLR::mscref()) -> packet()
```

Connection release with source local reference and destination local reference as parameters.
Documentation: ITU-T RecQ.713 4.5

prepend_ipaccess_header/1

```
prepend_ipaccess_header(SCCP::[integer()]) -> packet()
```

Prepend the ipaccess header to the sccp message.

Overview

Generated by EDoc, Jan 14 2011, 11:01:57.

Module service_sup

Behaviours: supervisor.

Function Index

init/1	Initialization of the service by adding server and child supervisor.
start_link/4	Starts a service supervisor that has two children, a Server and a Child supervisor that uses simple-one-for-one that will spawn children of the given type.

Function Details

init/1

```
init(X1::[atom()]) -> {ok, any()}
```

Initialization of the service by adding server and child supervisor.

start_link/4

```
start_link(Sup::atom(), Server::atom(), Child_sup::atom(), Child::atom()) ->
{ok, Pid} | ignore | {error, Error}
```

Starts a service supervisor that has two children, a Server and a Child supervisor that uses simple-one-for-one that will spawn children of the given type.

Generated by EDoc, Jan 14 2011, 11:01:57.

Module vlr

The interface between MSC and VLR database.

Version: '1.0'

Authors: openMSC.

Description

The interface between MSC and VLR database

Data Types

bscref()

```
bscref() = integer()
```

cell_info()

```
cell_info() = {lai(), ci()} | {{lac, integer()}, ci()} | {undefined, ci()}
```

cell_info()

```
cell_info() = {lai(), ci()} | {{lac, integer()}, ci()} | {undefined, ci()}
```

ci()

```
ci() = {ci, integer()}
```

id()

```
id() = tmsi() | imsi()
```

imsi()

```
imsi() = {imsi, integer()}
```

lai()

```
lai() = {lai, integer()}
```

lai()

```
lai() = {lai, integer()}
```

mscref()

```
mscref() = integer()
```

octets()

```
octets() = [integer()]
```

packet()

```
packet() = binary()
```

tmsi()

```
tmsi() = {tmsi, integer()}
```

Function Index

Function Details

create_database/1

```
create_database(DbName::Dbname) -> ok
```

```
    Dbname = string()
```

Create vlr database, the server is pre-defined as DB_HOST in vlr_server.hlr, Also create a view which match imsi with tmsi.

create_database/2

```
create_database(Server, DbName::Dbname) -> ok

    Server = {Ip::string(), Port::integer()}
    Dbname = string()
```

Create vlr database with a specified server

id_rsp/2

```
id_rsp(X1::{imsi, imsi()}, X2::{lai, lai()}) -> {wait_for_update_location_ack,
{imsi, imsi()}, {lai, lai()}} | {error, Reason} | {user_error, Reason} |
{badrpc, Reason}

    Reason = any()
```

the mobile station responds with the imsi for the request from MSC, and the update location procedure is continued

insert_sub_data/3

```
insert_sub_data(Imsi::imsi(), Msisdn::integer(), List::list()) -> ok |
{user_error, string()}
```

Invoke by HLR giving the corresponding imsi's detail info and VLR will update the Imsi with its msisdn and List.

page_ack/2

```
page_ack(Imsi::integer(), Data::any()) -> ok
```

Invoke by msc after recieving page_rsp from the ms, indicate the locaion of the ms

process_access_req/3

```
process_access_req(ID, Lai::lai(), Serving_Cell_id::cell_info()) -> {ok} |
{ok, wf_update_location_ack} | {ok, wf_ready_for_sms_ack} | {ok,
wf_update_location_ack, wf_ready_for_sms_ack} | {error, Reason::any()} |
{user_error, Reason::any()}

    ID = {imsi, imsi()} | {tmsi, tmsi()}
```

Initiate processing of an MS access to the network after being paged.

ready_for_sms_ack/2

```
ready_for_sms_ack(X1::{imsi, imsi()}, X2::{lai, lai()}) -> {lu_cnf, {tmsi,
tmsi()}, {lai, lai()}} | {wait_for_tmsi_realloc_comp, {tmsi, tmsi()}, {lai,
lai()}}
```

HLR replies after it is notified with that the mobile is reachable. Checking for allocation of new tmsi id done here.

search_for_ms_ack/2

```
search_for_ms_ack(Imsi::imsi(), Data) -> ok

     Data = lai() | {user_error, any()}
```

Invoke by msc after recieving page_rsp from the ms, indicate the locaion of the ms

send_info_for_mo_sms/1

```
send_info_for_mo_sms(ID::id()) -> ok | {user_error, Reason::any()}
```

Invoke by msc after receiving mo_forward from ss7 stack

send_info_for_mt_sms/1

```
send_info_for_mt_sms(Imsi::integer()) -> ok
```

Invoke by msc after recieving mt_forward from ss7 stack check config data in the database

update_location_area/2

```
update_location_area(X1::ID, X2::{lai, lai()}) -> {ok, {imsi, imsi()}, {lai,
lai()}} | {wait_for_update_location_ack, {imsi, imsi()}, {lai, lai()}} |
{wait_for_imsi, {lai, lai()}} | {error, Reason::any()} | {user_error,
Reason::any()} | {badrpc, Reason::any()}

     ID = {imsi, imsi()} | {tmsi, tmsi()}
```

Update location when ms move to a new lai and update the info in hlr. MS denotes itself via imsi or tmsi. If the tmsi is not in the database, ask the ms to send a imsi instead.

update_location_area_ack/2

```
update_location_area_ack(X1::{imsi, imsi()}, X2::{lai, lai()}) -> {ok, {imsi,
imsi()}, {lai, lai()}} | {wait_for_ready_for_sms_ack, {imsi, imsi()}, {lai,
lai()}} | {error, Reason::any()} | {badrpc, Reason::any()}
```

HLR successfully update the location and sends a confirm to VLR. Update the config for imsi and invoke sub_present_vlr to activate the imsi and start sms alert procedure.

Generated by EDoc, Jan 14 2011, 11:01:57.