MASTER THESIS

TITLE: LAPI project
SUBTITLE: Mobile Location Protocol implementation

DEGREE: Telecommunications Engineering

AUTHOR: Luis Alberto Gómez Domínguez

DIRECTOR: Lukáš Kenel, Robert Beštak

DATE: May 2009
RESEARCH AND DEVELOPMENT CENTRE
Overview

This project carries out a part of a new location application. This service consists of Location Based Application that connects with a Location Server. That server provides the necessary location information to run the service.

The goal of this project is to develop an interface between the Location Based Application and the Location Server.
ACKNOWLEDGEMENTS

This project would not have been possible without the support of many people.

First of all I want to mention my father, without whose support it would have been impossible to arrive where I am now.

I also want to thank the people at RDC with a special mention to Petr who has been always ready to help. And also Pep who “suffered” me during the first semester.

The last mention is for my friends, the ones I have met in Prague and the people in Barcelona.
CONTENTS

ACRONYMS ........................................................................................................................................... 12

1. INTRODUCTION ............................................................................................................................ 13
   1.1. SCOPE .................................................................................................................................. 13
   1.2. DOCUMENT CONTENT ....................................................................................................... 13

2. TECHNOLOGIES USED .................................................................................................................. 14
   2.1. MOBILE LOCATION PROTOCOL ....................................................................................... 14
       2.1.1. MLP structure ........................................................................................................... 14
       2.1.2. Transport layer .......................................................................................................... 15
           2.1.2.1. Element Layer .................................................................................................. 15
           2.1.2.2. Service Layer .................................................................................................. 15
       2.1.3. MLP request and response messages ....................................................................... 17
           2.1.3.1. Header .............................................................................................................. 17
           2.1.3.2. Standard Location Immediate Service ............................................................ 18
       2.1.4. MLP Result codes ....................................................................................................... 21
   2.2. HTTP ..................................................................................................................................... 21
       2.2.1. HTTP in MLP ............................................................................................................ 22
       2.2.2. Abyss Server ............................................................................................................. 23

3. LAPI IMPLEMENTATION .................................................................................................................. 24
   3.1. INTRODUCTION .................................................................................................................. 24
   3.2. MLP MODULE ..................................................................................................................... 24
       3.2.1. Integration in the SS7Box ........................................................................................ 24
       3.2.2. Module MLP architecture ........................................................................................ 24
           3.2.2.1. Transport Layer .............................................................................................. 26
           3.2.2.1.1. HttpServer ................................................................................................. 26
           3.2.2.1.2. MlpHandler .............................................................................................. 27
           3.2.2.2. Parser Layer .................................................................................................. 28
           3.2.2.2.1. MlpParser ................................................................................................. 28
           3.2.2.2.2. xmlParser ................................................................................................. 29
           3.2.2.2.3. SessionManager ....................................................................................... 30
           3.2.2.2.4. SessionDataMLP ...................................................................................... 31
           3.2.2.2.5. MLPServices .............................................................................................. 31
           3.2.2.2.6. ELayer ....................................................................................................... 32
           3.2.2.3. Application Layer .......................................................................................... 33
           3.2.2.3.1. SendertoLBS ............................................................................................ 33
           3.2.2.3.2. ProcMlpQuery ........................................................................................... 34
   3.3. MLP CLIENT ........................................................................................................................... 35
       3.3.1. MlpClient ................................................................................................................... 35
       3.3.2. MlpTemplates ............................................................................................................ 36

4. RESULTS OBTAINED ........................................................................................................................ 37
   4.1. INTRODUCTION .................................................................................................................. 37
   4.2. RUNNING MLP CLIENT ....................................................................................................... 37
   4.3. RUNNING MLP MODULE .................................................................................................... 39

5. CONCLUSIONS AND FUTURE LINES ............................................................................................. 42
   5.1. CONCLUSIONS ..................................................................................................................... 42
   5.2. FUTURE LINES ....................................................................................................................... 42
REFERENCES .......................................................................................................................... 43
APPENDIX A: MLP CLIENT CODE .............................................................................................. I
APPENDIX B: MODULE MLP CODE ........................................................................................... VI

TABLES
TABLE 1: MLP SERVICES (1) .................................................................................................... 16
TABLE 2: MLP ERROR CODES .................................................................................................. 21
TABLE 3: MLP PORTS ................................................................................................................ 22
TABLE 4: CLASSES OF THE MODULE MLP ............................................................................. 26
TABLE 5: ELEMENTS IMPLEMENTED BY THE LAPI .................................................................. 33

FIGURES
FIGURE 1: MLP ENVIRONMENT ............................................................................................... 14
FIGURE 2: MLP LAYERS (1) ....................................................................................................... 14
FIGURE 3: SLIS MESSAGE FLOW (1) .......................................................................................... 18
FIGURE 4: MODULE MLP STRUCTURE ...................................................................................... 25
FIGURE 5: MLPPARSER FLOW DIAGRAM ............................................................................... 28
FIGURE 6: SENDERTOLBS FLOW DIAGRAM .......................................................................... 34
FIGURE 7: PROCMLPQUERY FLOW DIAGRAM ..................................................................... 35
FIGURE 8: RUNNING MLP CLIENT ............................................................................................. 37
FIGURE 9: MLP CLIENT RECEIVING SLIA WITH ERROR ....................................................... 38
FIGURE 10: MLP CLIENT RECEIVING SLIA WITH LOCATION .............................................. 38
FIGURE 11: EXECUTING MLP_LOCAL.SH ............................................................................. 39
FIGURE 12: RUNNING THE MODULE ......................................................................................... 40
FIGURE 13: MLP WAITING FOR THE LBS MODULE RESPONSE ........................................... 40
FIGURE 14: MODULE MLP RETURNING SLIA WITH LOCATION ........................................... 41
## ACRONYMS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPI</td>
<td>Location Application Programming Interface</td>
</tr>
<tr>
<td>MLP</td>
<td>Mobile Location Protocol</td>
</tr>
<tr>
<td>OMA</td>
<td>Open Mobile Alliance</td>
</tr>
<tr>
<td>SS7</td>
<td>Signaling System #7</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>SLIS</td>
<td>Standard Location Immediate Service</td>
</tr>
<tr>
<td>SLIR</td>
<td>Standard Location Immediate Request</td>
</tr>
<tr>
<td>SLIA</td>
<td>Standard Location Immediate Answer</td>
</tr>
<tr>
<td>SLIREP</td>
<td>Standard Location Immediate Report</td>
</tr>
</tbody>
</table>
1. Introduction

1.1. Scope

Mobile communications are in constant change. Users need for more services every day and that is why companies think in Location Based Services (LBS). This services use the location of the Mobile Subscriber to provide any kind of service.

The purpose of this project is to develop a Location API (LAPI). This API is an interface between a Location Based Application and a Location Server.

This LAPI is a implementation of the Mobile Location Protocol (MLP) created by the Open Mobile Alliance (OMA). It is programmed using C++.

The development of this LAPI was divided in two parts. One is concerning the communication between the Location Server and the Location Based Application, and another concerning the implementation into the SS7Box Location Server.

1.2. Document content

This document is about the implementation of the communication between the Location Server and the Location Based Application and the implementation of the LAPI. The implementation into the SS7Box Server was developed by Pep Soler and can be found in (1)

The document is divided in different sections described next.

Chapter 2 discusses the technologies used for the design and implementation of the LAPI. Chapter 3 explains the design and implementation of the LAPI. In Chapter 4 are shown the results obtained by running the LAPI. Finally, Chapter 5 has the conclusions and future lines to be followed by the project.
2. Technologies used

2.1. Mobile Location Protocol

The MLP (2) is an application-level protocol for getting the position of mobile stations (mobile phones, wireless personal digital assistants, etc.) independent of underlying network technology. The MLP serves as the interface between a Location Server and a Location Based Application.

The MLP is a request/response based protocol. The client starts the communication with the location server sending a location query which is responded by the server. The query and the response consist in XML (3) messages defined in the MLP protocol.

2.1.1. MLP structure

The MLP structure is divided in three different layers which permit implement the protocol in many different ways.

Figure 1: MLP environment

Figure 2: MLP Layers (2)
The Figure 2 shows the different layers in which the MLP is divided: transport layer, element layer and Service layer.

2.1.2. Transport layer

The Transport Layer defines the kind of communication between the Location Server and the Location based Application. It is independent from the XML content.

This layer describes how the XML is transported. Possible MLP transport protocols could be HTTP, WSP, SOAP among others.

In the implementation defined in this document the transport layer uses HTTP. Nowadays, only HTTP mapping is provided by the MLP specifications.

The Transport Layer implementation is explained in the section 3.2.2.1

2.1.2.1. Element Layer

The Element Layer defines all the common elements used by the different services defined in the Service layer. The elements are defined in the following document type definitions (DTD):

- **MLP_ID.DTD**: Identify Element Definitions.
- **MLP_FUNC.DTD**: Function Element Definitions.
- **MLP_LOC.DTD**: Location Element Definitions.
- **MLP_RES.DTD**: Result Element Definitions.
- **MLP_SHAPE.DTD**: Shape Element Definitions.
- **MLP_QOP.DTD**: Quality of Position Element Definitions.
- **MLP_GSM_NET.DTD**: GSM Network Parameters Element Definitions.
- **MLP_CTXT.DTD**: Context Element Definitions.

All these DTD can be found in the web of the OMA.

2.1.2.2. Service Layer

The Service Layer defines the different location services offered by the MLP. Basic MLP services are based on the Location Services defined by 3GPP specification (4) and are described by the OMA specification (2). Advanced MLP Services and Other MLP Services are additional services that will be specified by other organization or forums.

The Service Layer is divided in two sub-layers, one lower defining the common elements of the Services and the top sub-layer defines the specific elements of each services. The different Basic Location Services are briefly defined in the Table 1.
<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard Location Immediate Service</strong></td>
<td>This is a standard query service with support for a large set of parameters. This service is used when a (single) location response is required immediately (within a set time) or the request may be served by several asynchronous location responses (until a predefined timeout limit is reached).</td>
</tr>
<tr>
<td><strong>Emergency Location Immediate Service</strong></td>
<td>This is a service used especially for querying of the location of a mobile subscriber that has initiated an emergency call. The response to this service is required immediately (within a set time) or the request may be served by several asynchronous location responses.</td>
</tr>
<tr>
<td><strong>Standard Location Reporting Service</strong></td>
<td>This is a service that is used when a mobile subscriber wants an LCS Client to receive the MS location. The position is sent to the LCS Client from the location server. Which LCS application and its address are specified by the MS or defined in the location server.</td>
</tr>
<tr>
<td><strong>Emergency Location Reporting Service</strong></td>
<td>This is a service that is used when the wireless network automatically initiates the positioning at an emergency call. The position and related data is then sent to the emergency application from the location server. Which LCS application and its address are defined in the location server.</td>
</tr>
<tr>
<td><strong>Triggered Location Reporting Service</strong></td>
<td>This is a service used when the mobile subscriber's location should be reported at a specific time interval or on the occurrence of a specific event.</td>
</tr>
</tbody>
</table>

This service consists of the following messages:
- Standard Location Immediate Request
- Standard Location Immediate Answer
- Standard Location Immediate Report

This service consists of the following messages:
- Emergency Location Immediate Request
- Emergency Location Immediate Answer
- Emergency Location Immediate Report

This service consists of the following message:
- Standard Location Report
- Standard Location Report Answer

This service consists of the following message:
- Emergency Location Report

Table 1: MLP services (2)

Each implementation of location server can select which services would be supported. In the case of the LAPI implementation only the Standard Location Immediate Service (SLIS) is supported. See section 2.1.3.2 for a detailed description of SLIS
2.1.3. MLP request and response messages

The MLP messages are XML messages that should follow a predefined structure. An MLP request message is composed by a header part and a body part. A MLP response message may have a header part but must have a body part.

In a request the header and the body are encapsulated into the same Service Initiation DTD, MLP_SVC_INIT_320.dtd. The MLP Service Initiation XML has the following structure:

```xml
<?xml version="1.0" ?>
<!DOCTYPE svc_result SYSTEM "MLP_SVC_INIT_320.DTD">
<svc_init ver="3.2.0">
  <hdr ver="3.2.0">
    .........
  </hdr>
  <slir ver="3.2.0">
    .........
  </slir>
</svc_init>
```

The context header holds the authentication and authorization data pertinent to a particular location request. In a MLP message the body part specifies the service used in the request/response. In the case of the LAPI only the SLIS is implemented.

An MLP response is also encapsulated into a DTD, the MLP_SVC_RESULT_320.dtd. The XML message has the following structure:

```xml
<?xml version="1.0" ?>
<!DOCTYPE svc_result SYSTEM "MLP_SVC_RESULT_320.DTD">
<svc_result ver="3.2.0">
  <slia ver="3.0.0">
    .........
  </slia>
</svc_result>
```

2.1.3.1. Header

The MLP header is described in the MLP_HDR_320.DTD. The header carries the information about the LCS Client and identifies it to the server. The Location server should be capable of identifying the Client-Mobile Location Application which is acceding.

For more detail about the header and the use in the LAPI see the section 3.2.2.2.2
2.1.3.2. Standard Location Immediate Service

The SLIS is the service implemented in the LAPI. The SLI Service is a standardized service used for requesting the location of one or more Mobile Subscribers.

As it is described in the Table 1, the SLIS consist of three different kinds of messages: Standard Location Immediate request (SLIR), Standard Location Immediate Answer (SLIA) and Standard Location Immediate Report (SLIREP).

SLIR message contains the query and is followed by one or more SLIA with the location or locations of the different Mobile Subscribers and zero, one or more optional SLIREP. The procedure is shown in the next schema.

![Figure 3: SLIS message flow (2)]

**Standard Location Immediate Request**
The SLIR is sent in the body of the Service Initiation message. The SLIR is specified in the MLP_SLIR_320.DTD.

As mandatory elements a SLIR message should include the msid of the Mobile Subscriber. The msid can have different types as it is described in the MLP_ID_320.DTD.

```xml
<msid (#PCDATA)>
</msid>
```

In the LAPI is only supported the International Mobile Subscriber Identity (IMSI). The IMSI is a 15 digits long number. The first 3 digits are the Mobile country Code (MCC),
the next digits are the Mobile Network Code (MNC) and the remaining digits are the Mobile station Identification Number (MSIN).

Following example shows simple SLIR message body.

```xml
<slir ver='3.2.0' res_type='SYNC'>
  <msids>
    <msid type="IMSI">303555100312375</msid>
  </msids>
</slir>
```

The description of the supported elements of SLI in the LAPI is detailed in the section 3.2.2.2.2

**Standard Location Immediate Answer**

The SLIA is sent in the body of the Service Result. The SLIR is specified in the MLP_SLIA_320.DTD.

As mandatory elements the SLIA should include the element **pos**. The element **pos** is defined in the MLP_LOC_320.DTD and determines the location of the Mobile subscriber.

```xml
<!ELEMENT pos (msid, (pd | poserr), gsm_net_param?, trans_id?)>
<!ATTLIST pos
  pos_method (CELL | OTDOA | GPS | A-GPS | E-OTD | U-TDOA | AFLT | EFLT | UNKNOWN | OTHER) #IMPLIED>
```

In the same time the element **pos** should contain the elements **msid** and **pd**, if the location has been successful, or **poserr** in case that has been returned an error during the process. These two elements are defined also in the MLP_LOC_320.DTD.

An example of successful SLIA:

```xml
<slia ver="3.0.0">
  <pos>
    <msid type="IMSI"> 303555100312375</msid>
    <pd>
      <time utc_off="+0200">20090402172443</time>
      <shape>
        <Point>
          <coord>
            <X>X</X>
            <Y>Y</Y>
          </coord>
        </Point>
      </shape>
    </pd>
  </pos>
</slia>
```
Example of simple error message:

```xml
<slia ver='3.0.0'>
  <pos>
    <msid type="IMSI">303555100312375</msid>
    <result resid=error>message</result>
    <add_info>info</add_info>
  </pos>
</slia>
```

Where error is indicated by the result code resid and message is the result code slogan described in the section 2.1.4

The implementation of the SLIA in the LAPI is described in the section 3.2.2.2 of this document

**Standard Location Immediate Report**

The SLIREP is defined in the MLP_SLIREP_300.DTD. A SLIREP is sent to the Location Based Application if the location of multiple targets has been requested and the response is sent back in individual reports. Each report can contain the result of one or more targets.

The following example shows a possible SLIREP:

```xml
<slirep ver="3.0.0">
  <req_id>25267</req_id>
  <pos>
    <msid type="IPv6">10:A1:45::23:B7:89</msid>
    <pd>
      <time utc_off="+0300">20020813010423</time>
      <shape>
        <CircularArea srsName="www.epsg.org#4326">
          <coord>
            <X>35 03 28.244N</X>
            <Y>135 47 08.711E</Y>
          </coord>
          <radius>15</radius>
        </CircularArea>
      </shape>
    </pd>
  </pos>
</slirep>
```

The SLIREP is not supported by the LAPI.
2.1.4. MLP Result codes

The MLP result codes define the possible situations which can occur while processing an MLP request. These results are divided into different ranges:

- 0 - 99 Location server specific errors
- 100 - 199 Request specific errors
- 200 - 299 Network specific errors
- 300 - 499 Reserved for future use
- 500 - 599 Vendor specific errors
- 600 - 699 MLS Client specific errors

In the section 5.4 of (2) are described all the errors. The ones implemented by the LAPI are:

<table>
<thead>
<tr>
<th>Resid</th>
<th>Slogan</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK</td>
<td>No error occurred while processing the request</td>
</tr>
<tr>
<td>2</td>
<td>UNSPECIFIED ERROR</td>
<td>An unspecified error used in case none of the other errors apply. This can also be used in case privacy issues prevent certain errors from being presented.</td>
</tr>
<tr>
<td>105</td>
<td>FORMAT ERROR</td>
<td>A protocol element in the request has invalid format. The invalid element is indicated in ADD_INFO.</td>
</tr>
<tr>
<td>106</td>
<td>SYNTAX ERROR</td>
<td>The position request has invalid syntax. Details may be indicated in ADD_INFO.</td>
</tr>
<tr>
<td>107</td>
<td>PROTOCOL ELEMENT NOT SUPPORTED</td>
<td>A protocol element specified in the position request is not supported by the Location Server, or the position result is not supported by the LCS Client. The element is indicated in ADD_INFO.</td>
</tr>
<tr>
<td>108</td>
<td>SERVICE NOT SUPPORTED</td>
<td>The requested service is not supported in the Location Server. The service is indicated in ADD_INFO.</td>
</tr>
<tr>
<td>113</td>
<td>PROTOCOL ELEMENT ATTRIBUTE VALUE NOT SUPPORTED</td>
<td>A specific value of a protocol element attribute is not supported in the Location Server. The attribute and value are indicated in ADD_INFO.</td>
</tr>
<tr>
<td>203</td>
<td>CONGESTION IN MOBILE NETWORK</td>
<td>The request cannot be handled due to congestion in the mobile network.</td>
</tr>
</tbody>
</table>

Table 2: MLP Error Codes

2.2. HTTP

In the MLP the kind of transport, defined in the Transport Layer, is independent from the XML part, Element Layer and Service Layer. The OMA specification defines only the HTTP mapping for the MLP.

HTTP (4) is a request/response protocol involving a server and a client. It is defined in the Application Layer of the OSI standard though in the MLP protocol it is defined in the Transport Layer.
Typically, an HTTP client initiates a request. It establishes a Transmission Control Protocol (TCP) connection to a particular port on a host. An HTTP server listening on that port waits for the client to send a request message. Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 OK", and a message, body which is the requested resource, an error message, or some other information.

### 2.2.1. HTTP in MLP

In the context of the MLP, the client is referred to as the LCS Client and the server is the Location Server (GMLC/MPC).

For the MLP protocol the server may provide two different ports, one for encrypted communications and other without encryption. At the Internet Assigned Numbers Authority (IANA) there are registered the port numbers used for the MLP protocol. These port numbers are detailed in the Table 3.

<table>
<thead>
<tr>
<th>MLP PORTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lif-mlp</td>
<td>9210/tcp</td>
<td>LIF Mobile Locn Protocol</td>
</tr>
<tr>
<td>lif-mlp</td>
<td>9210/udp</td>
<td>LIF Mobile Locn Protocol</td>
</tr>
<tr>
<td>lif-mlp-s</td>
<td>9211/tcp</td>
<td>LIF Mobile Locn Secure</td>
</tr>
<tr>
<td>lif-mlp-s</td>
<td>9211/udp</td>
<td>LIF Mobile Locn Secure</td>
</tr>
</tbody>
</table>

**Table 3: MLP ports**

An LCS Client shall request a Location Service by issuing an HTTP POST request towards the Location Server. For more information about HTTP POST, see (4). The request line syntax is shown below.

```
Request-line = POST path HTTP/1.1
```

The request must include the entity-header Content-length field as part of the request. The message body of the request should include the XML formatted request and should have the length specified by the LCS Client in the Content-length field.

The response to the invocation of a Location Service should be returned using an HTTP response.

When an LCS client attempts to invoke a service request that is not defined in this specification, the Location Server should return a General Error Message (GEM) in a HTTP '404' error response:

```
Status-Line= HTTP/1.1 404 Not Found
```

In the LAPI the Transport Layer has been implemented using the Abyss Server into the XMLRPC-C libraries, using also some facilities from this protocol.
2.2.2. Abyss Server

Abyss is an HTTP server developed by Moez Mahfoudh, see (5). It was open source until 2001 when became closed source. In the MLP module Abyss Server for XMLRPC-C is used (-version 1.06, see (6)). This version is still open source and has been upgraded together with this protocol.

The MLP module uses Abyss among all the different alternatives because of three main reasons:

1. Abyss is Open Source, which means that we can accede to the code of the server and modify, use, or adapt it to our own purpose.
2. It is light comparing with other HTTP servers as for example Apache so it will not take many resources from the system.
3. It is already installed in the SS7Box because is used by the XML-RPC protocol, in the management modules.

With the Abyss server some facilities provided by the XML-RPC for C/C++ are used. These are the libraries libxmlrpc_abyss and libxmlrpc_server_abyss. With these libraries it is possible to add a new handler to the server that process the MLP queries. It is described in the section 3.2.2.1.2
3. LAPI implementation

3.1. Introduction

The LAPI is based on the MLP v.3.2, so for its implementation has followed the specifications of this protocol detailed in (2).

The LAPI implements the SLIS of the MLP. There are implemented the SLIR with support for the location of an individual target and the SLIA to provide the location of a single target. The SLIREP is not implemented in this version of the LAPI.

The LAPI runs in the SS7Box location server. This location served is placed in the RDC. The LAPI has been programmed to be integrated into the server architecture. In the SS7Box the LAPI is integrated as the Module MLP.

3.2. MLP module

3.2.1. Integration in the SS7Box

The MLP module is a module in the SS7Box Location Server. To work into the Server the module has to follow the architecture of the server.

The SS7Box consists of three layers. The lowest layer is the DataKinetics layer which provides SS7 stack functionalities. The next is the Core layer which provides communication between DataKinetics and the highest layer, the Application layer. The LAPI is implemented as an Application Module and is called MLP module.

In the location server the LAPI works as an interface between the module LBS which provides the location information and the Location Based Application. The basic function of the Module MLP is to receive the MLP queries, process them and obtain its location information from the Module LBS to send the MLP response to the Location Based Application.

For more information about the SS7Box structure see (7)

3.2.2. Module MLP architecture

The module MLP can be divided in three different layers depending on the different functions that the module develops. The three layers define the basic functionality of the module. These three layers are:
- **Transport layer.** Defines the communication between the Location Server and the Location Based Application through the network using HTTP.
- **Parser layer.** Defines the functions to process the MLP messages. That functions code and decode MLP messages, getting the necessary information from a message received or making the response message from the information obtained from LBS module.
- **Application layer.** Defines the communication with the LBS module, receiving or sending messages.

In the Figure 4 there is a description of the layers and the functions developed in each.

The Module MLP runs in two different threads. One thread corresponds to the communications in the SS7Box and the other thread corresponds to the HTTP Server and the communication with the Location Based Applications.

To perform the functions described, the Module MLP is composed by the following C++ classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MlpModule</strong></td>
<td>Class where the Module is started. It manages the communication with the SS7Box and defines the state of the module.</td>
</tr>
<tr>
<td><strong>HttpServer</strong></td>
<td>Starts and runs the Abyss server.</td>
</tr>
<tr>
<td><strong>MlpHandler</strong></td>
<td>Process the HTTP query</td>
</tr>
<tr>
<td><strong>MlpParser</strong></td>
<td>Process the MLP message</td>
</tr>
<tr>
<td><strong>MlpValidate</strong></td>
<td>Validate the MLP against DTD</td>
</tr>
<tr>
<td><strong>xmlParser</strong></td>
<td>Parse the XML document using libxml2 libraries</td>
</tr>
<tr>
<td><strong>SendertoLBS</strong></td>
<td>Send the IMSI to the LBS module</td>
</tr>
<tr>
<td><strong>SessionDataMLP</strong></td>
<td>Stores the information about the MLP query. Inherits from ELayer.</td>
</tr>
</tbody>
</table>
### Table 4: Classes of the Module MLP

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SessionManager</td>
<td>Stores the <code>SessionDataMLP</code> to be accessed by both threads.</td>
</tr>
<tr>
<td>MLPServices</td>
<td>Defines the MLP response message</td>
</tr>
<tr>
<td>ELayer</td>
<td>Stores the information about the MLP message.</td>
</tr>
<tr>
<td>ProcMlpQuery</td>
<td>Processes the response of the LBS module. It queries a DataBase for the</td>
</tr>
<tr>
<td></td>
<td>coordinates of the cell</td>
</tr>
<tr>
<td>MlpQueryMsg</td>
<td>Get the incoming message from the LBS module</td>
</tr>
<tr>
<td>MyAppConstants</td>
<td>Defines the module constants</td>
</tr>
<tr>
<td>NetConstants</td>
<td>Defines the error constants returned from the LBS module</td>
</tr>
</tbody>
</table>

The code for the classes of the MLP module can be found in the appendix B.

### 3.2.2.1. Transport Layer

The transport layer defines the communication of the module with the possible Location Based Applications.

The functions of this layer can be divided depending on the way of communication. For the incoming messages the Transport Layer gets the incoming HTTP POST and verifies if the HTTP query accomplish the requirements of the MLP protocol described in the section 2.2.1 of this document. For the outgoing messages it sent the MLP response inside an HTTP response message.

The C++ classes that define the Transport Layer in the Module MLP are: `HttpServer` and `MlpHandler`.

#### 3.2.2.1.1. HttpServer

The class `HttpServer` is the class where the Abyss Server is started. This class runs a server in a new thread different from the core of the MLP module. The call for the initialization of the HTTP server thread is done by the class `MlpModule` in its initialize method.

The different methods of the class set the server properties and specify the server handlers. These methods are described in the following subsection.

**HttpServer::run()**

In this method is where the Abyss Server is set and run. It starts creating the server and defining its different parameters as port, the 9210 defined for the MLP protocol, or the server log.
Before running the server it defines the Handlers. The Abyss Server will have a Handler called MLP Handler which could process the MLP queries and a default handler called when the HTTP doesn’t fit the MLP requirements.

Finally the server is initialized and run.

**HttpServer::SetServerHandler(TServer *const srvP, const char * uriPath)**

This method set the MLP handler. This Handler permits process the MLP queries arriving to the Module MLP.

The handler is added to the server using the function ServerAddHandler2 present in the abyss.h header.

The function ServerAddHandler2 has the following header

```c
ServerAddHandler2(TServer * srvP, URIHandler2 * handlerP, 
                    abyss_bool * success)
```

The struct URIHandler2 contains the a variable call void*userdata. This user data contains the shared elements by all the queries arriving to the server. This data is defined as a structure call uriHandlerMLP with the following elements:

```c
struct uriHandlerMlp{
    SessionManager* sessionManager;
    const char *uriPath;
    int src;
};
```

For more detailed information about the use of this function see: [http://xmlrpc-c.sourceforge.net/doc/libxmlrpc_abyss.html#ServerAddHandler2](http://xmlrpc-c.sourceforge.net/doc/libxmlrpc_abyss.html#ServerAddHandler2).

### 3.2.2.1.2. MlpHandler

The MlpHandler is called when a query arrives to the server. The Handler verifies that the HTTP post accomplishes all the requirements of an MLP request described in the section 2.2.1 of this document. If the request is correct, it is passed to the MlpParser class if not, error is returned using the Default Handler.

**MlpHandler::MlpHandlerRun(struct URIHandler2 * urihandler,TSession * abyssSessionP,abyss_bool * handledP)**

This method processes the message entering to the Abyss Server. In case the message is correct it calls the parseMLP method in the MlpParser class, otherwise it returns an error to the client.
3.2.2.2. Parser Layer

The Parser Layer implements the SLIR service from the MLP described in the section 2.1.3.2 of this document. To perform the parsing of the MLP message the LAPI uses the libxml2 libraries, (8)

This layer extracts the MLP message from the HTTP request and processes it. It returns an MLP response or an MLP error depending on the situation. It is implemented by three different classes: *MlpParser, xmlParser, MlpValidator*.

3.2.2.2.1. MlpParser

This class receives the correct HTTP query from the MlpHander and extracts the MLP message from it. Afterwards call the *MlpValidator* to verify the message against the DTD. Depending on the result of the validator the message is parsed by the *xmlParser* or an error is returned to the client.

In the Figure 5 it is shown a flow diagram of the operation of this class.

![Figure 5: MlpParser flow diagram](image)
Chapter 3

The MlpParser gets the MLP response message calling an instance from the SessionDataMLP which generates the kind of response message depending on the result of processing the message.

3.2.2.2.2. xmlParser

If the message has been validated against the DTD it arrives to the xmlParser class. This class parses the MLP message and extracts its information. The information is stored in an SessionDataMlp object. The header and the body of the message are processed in the separate methods.

xmlParser::processHdr(SessionDataMLP* m_sessionDataMLP, bool *res)

This method processes the header of the MLP message. The information extracted is stored into the SessionDataMLP object. The variable bool *res indicates to the MlpParser if the header has been processed correctly (TRUE), or not (FALSE).

The header of the MLP message is described in the MLP_HDR_320.DTD and defines the following element.

<!ELEMENT hdr ((client | sessionid | (client, sessionid)), subclient*, requestor?)>
<!ATTLIST hdr
   ver CDATA #FIXED "3.2.0"
>

From this only client is supported. Client is defined in MLP_CTXT_320.DTD as follows.

<!ELEMENT client (id, pwd?, serviceid?, requestmode?)>

From client are only supported id and pwd. In case the MLP message is composed by any other element the MLP is not processed, *res=false, and the error code 107, PROTOCOL ELEMENT NOT SUPPORTED is indicated into the SessionDataMLP.

The possibility of validate the client identity is not yet implemented in the LAPI but should be done in this method

xmlParser::processSvc(SessionDataMLP* m_sessionDataMLP, bool* res)

This method processes the body or service of the MLP message. The information extracted is stored into the SessionDataMLP object. The variable bool *res indicates to the MlpParser if the service has been processed correctly (TRUE), or not (FALSE).

The LAPI only supports the SLIR. In case the body of the MLP message corresponds to any other service the MLP is not processed. The SLIR defines the following elements into the MLP_SLIR_320.DTD.
3.2. SessionManager

The SessionManager is an object shared between both threads. It stores the SessionDataMLP objects with contains the particular information of each request. The SessionManager assign a key to each sessionDataMLP and stores it to be accessed by both threads.

The main methods used in this class are:

- **insertSession**: it is used to add a new Session into the SessionManager. the SessionDataMLP object pointer and its key have to be specified.
- **deleteSession**: it is used to delete an old Session of the SessionManager. The identifier has to be specified. Therefore, this function searches in the m_session the key which is equals to the identifier. After finding the key; it deletes the associated SessionDataMLP object.
- **getKeyData**: it is used to search the SessionDataMLP object. The identifier has to be specified. Then this function searches in the m_session the key which is equals to the identifier. After finding the key, it returns a pointer to the associated SessionDataMLP object.
3.2.2.2.4. SessionDataMLP

In the SessionDataMLP is stored the information about the MLP request. Each request has its own SessionDataMLP object stored into the SessionManager object. The SessionDataMLP should be accessible from both threads because contains common information about the requests.

The SessionDataMLP creates an object of the class MLPServices. In this object is where the information of the MLP is stored.

The sessionDataMLP also control the synchronization between both threads. It is done by the methods WaitComplete and SignalComplete. The first function is called by the Handler MLP to keep waiting for the signal sent by the processAppResponse when it has filled the SessionDataMLP with the location information. That signal is sent with the function SignalComplete.

3.2.2.2.5. MLPServices

The MLPServices class defines the elements supported by the LAPI. This class inherits from the class ELayer.

The MLP response message is generated by the MlpServices object by the method string DoMLPMessage(). This method reads the Result Code returned by processing the message and generate an error message or a successful response message.

In case of successful location it returns the following message

```xml
<?xml version="1.0" ?>
<!DOCTYPE svc_result SYSTEM MLP_SVC_RESULT_320.DTD">
<svc_result ver="3.2.0">
  <slia ver="3.0.0">
    <pos>
      <msid type="IMSI"> 303555100312375</msid>
      <pdp>
        <time utc_off="+2000">20090511114847</time>
        <shape>
          <Point>
            <coord>
              <X>16.34400</X>
              <Y>49.07200</Y>
            </coord>
          </Point>
        </shape>
      </pdp>
    </pos>
  </slia>
</svc_result>
```
In case of error it returns this other message indicating the kind of error:

```xml
<?xml version="1.0" ?>
<!DOCTYPE svc_result SYSTEM MLP_SVC_RESULT_320.DTD">
<svc_result ver="3.2.0">
  <slia ver="3.0.0">
    <pos>
      <msid type="IMSI">303555100312375</msid>
      <poserr>
        <result resid=error>message</result>
        <time utc_off="+2000">2009051114847</time>
      </poserr>
    </pos>
  </slia>
</svc_result>
```

3.2.2.2.6. ELayer

This class defines the elements and attributes of the Element Layer supported by the LAPI.

The elements implemented can be found as a structure. For example: there is one element called result from the Result Elements in the Element layer. That element has an attribute called resid. So, in the *ELayer* class the structure that represents this element is:

```c
struct resultTag{
    string resid;
    string message;
}fillResultTag;
```

Where message, is the value of the *result* element. Then, a summary of the elements implemented is showed in the Table 5.

<table>
<thead>
<tr>
<th>Element</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msid</td>
<td>Identity Elements Definitions</td>
<td>Represents an identifier of a mobile subscriber</td>
</tr>
<tr>
<td>loc_type</td>
<td>Function Elements Definitions</td>
<td>Defines the type of location Requested.</td>
</tr>
<tr>
<td>result</td>
<td>Result Elements Definitions</td>
<td>Indicates the result of a request.</td>
</tr>
<tr>
<td>pos</td>
<td>Location Element Definition</td>
<td>Contains the elements: msid, pd or poser, gsm_net_param(optional) and trans_id(optional).</td>
</tr>
<tr>
<td>pd</td>
<td>Location Element Definition</td>
<td>Contains the elements: time, shape and alt, alt_unc, speed, direction, lev conf and gos_not_met optionally.</td>
</tr>
<tr>
<td>time</td>
<td>Location Element Definition</td>
<td>Indicates the time when the positioning was performed.</td>
</tr>
<tr>
<td>shape</td>
<td>Shape Element Definition</td>
<td>Contains the elements: Point or LineString or Polygon or Box or CircularArea or</td>
</tr>
</tbody>
</table>
CircularArcArea or EllipticalArea or MultiLineString or MultiPoint or MultiPolygon or LinearRing

**point**  Shape Element Definition Contains the element: coord

**coord**  Shape Element Definition it contains the elements: X, Y (optional) and Z (optional).

**X**  Shape Element Definition X is the first ordinate in a coordinate system, and denotes the latitude.

**Y**  Shape Element Definition Y is the second ordinate in a coordinate system, this is optional in a linear coordinate system, and denotes the longitude.

**Z**  Shape Element Definition Third ordinate in a coordinate system which has at least three ordinates

**poserr**  Location Element Definition Contains the elements: result, add_info(optional) and time.

**hdr**  Header DTD Contains the elements: sessionid or client and sessionid (optional), subclient and requestor (optional)

**client**  Context Element Definition Contains the elements: id, pwd(optional), serviced(optional) and request mode(optional).

**id**  Context Element Definition A string defining the name of a registered user performing a location request. In an answer the string represents the name of a location server

**pwd**  Context Element Definition The password for the registered user performing a location request. In this answer the string represents the password for a location server

<table>
<thead>
<tr>
<th>Table 5: elements implemented by the LAPI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>3.2.2.3. Application Layer</strong></td>
</tr>
</tbody>
</table>

The Application Layer defines the communication from and to the module LBS. It is basically developed by the classes *SendertoLBS, ProcMlpQuery* and *MlpQueryMsg*.

The communication with the module LBS is only established in case the MLP query has been correctly processed by the Parser Layer. The *SendertoLBS* is called from the *MlpParser*. The *sendertoLBS* send the *msid* to the module LBS and remains waiting until the *ProcMlpQuery* process the response with the location.

**3.2.2.3.1. SendertoLBS**

The *SendertoLBS* class is called from the *MlpParser* if the MLP query is supported by the LAPI. The *sendertoLBS* extracts the *msid* from the *SessionDataMLP* and send a message with it to the module LBS calling the method *bool SendtoLBS(string msid, int key)* of this class.
In case the message is sent correctly to the LBS module, the code is waiting until receiving a signal from the `ProcMlpQuery` class that means the dialog with the LBS module has finished.

After the `sendertoLBS` obtain the response message from the `SessionDataMLP` object and returns it to the `MLPParser` to be sent back to the LocationBased Application.

The Figure 6 shows a flow diagram of the `sendertoLBS` class.

![Figure 6: SendertoLBS flow diagram](image)

The return MLP message is obtained using a call to the `DoMLPMessage` method in the class `MLPServices` through the `SessionDataMLP` object.

### 3.2.2.3.2. ProcMlpQuery

The `ProcMlpQuery` receives and processes the response from the LBS module. The figure 7 shows a flow diagram of this class. For detailed information about this class see (1).
3.3. MLP Client

The MLP Client has been developed to test the Module MLP. It only implements the MLP Transport Layer. That means that only establishes the connection with the Location Server using the specified port in the MLP protocol and sends a HTTP POST encapsulating the message in the body of the HTTP. This is implemented using the XMLRPC-C libraries.

When running the MLP Client, it waits for the introduction of an msid value by the standard input (keyboard) and then, generates and sends an MLP message to the Module MLP and receives the MLP response message. It runs in a loop, so after performing a query it ask for an msid again. The execution finishes when the user introduces ‘q’ or ‘quit’ by the keyboard.

The MLP Client is formed by the classes MlpClient and MlpTemplates. The code of this classes can be found in the Appendix A.

3.3.1. MlpClient

The MlpClient class has to main functions. First, waits for the introduction of and msid and after, establishes the communication with the Location Server.

For the communication the MLP Client uses some functions provided by XMLRPC-C. These functions define the kind of transport between the client and the server. The lines of code which sets the transport are:
carriageParm_http0 * carriage = new carriageParm_http0(serverUrl);
carriageParm_http0 carriageParm(*carriage);

clientXmlTransportPtr transportP(clientXmlTransport_http::create());
carriageParm_http0 carriageParm((*carriage));

Detailed information for these functions can be found in the XMLRPC-C documentation in the URL (10)

3.3.2. MlpTemplates

The class MlpTemplates generates the SLIR message. The class generates the message as a string. The header and the body of the MLP message are generated by different methods. This adds modularity to the design.

The message is generated as a string and has the structure shown below.

```xml
<?xml version = "1.0"?>
<!DOCTYPE svc_init SYSTEM "dtd/MLP_SVC_INIT_320.DTD">
<svc_init ver="3.2.0">
  <hdr ver="3.2.0">
    <client>
      <id>MlpClient</id>
    </client>
  </hdr>
  <slir ver="3.2.0" res_type="SYNC">
    <msids>
      <msid type="IMSI">msid</msid>
    </msids>
  </slir>
</svc_init>
```

The msid element has the value indicated by the standard input.
4. Results Obtained

4.1. Introduction

In this section are shown the results obtained by testing the LAPI.

4.2. Running MLP Client

The MLP client is run executing the instruction `./MlpClient` in the command line. The MLP Client ask for the introduction of an msid value and then generates and send the message to the Location Server.

The Figure 8 shows an execution of the MLP client waiting for the response from the Location Server.

In the Figure 9 we can see a successful Location SLIA message arriving to the MLP Client. The Figure 10 shows an unsuccessful Location SLIA reporting an error to the MLP Client.
Chapter 4

Figure 9: MLP Client receiving SLIA with error

Figure 10: MLP Client receiving SLIA with location
4.3. Running MLP module

The MLP Module is running inside the SS7Box so, to start it some specific instructions have to be followed. To start the running this three instructions have to be written in the command line:

1. `python mlp_create_queue.py`
   This script return -2 if the queue was already created, 0 if successful. Any other value indicates an error.

2. `./mlp_local mlpconfig.xml`
   This script starts the module. It should be running in background every time. When the module starts read the configuration XML. It set the queue which connect with the LBS, connect with the database which contain the locations and finally runs the server.

3. `python mlp_run.py`
   Run the module. Returns 0 if successful. Any other number indicates error.

The next figures show the procedure for running the MLP module.

![Figure 11: Executing mlp_local.sh](image)
Once running, the module is ready to receive Location requests. The execution follows the procedure explained in the section 3. The next captures show an example of query processing.
Figure 14: Module MLP returning SLIA with Location
5. Conclusions and Future Lines

5.1. Conclusions

Working in the design and implementation of the LAPI has provided me a great learning in the fields of Mobile Communications, C++ programming or Unix among others.

The LAPI is a valid interface to provide the location of a mobile subscriber. Though there are many possibilities of design a LAPI based in the MLP, in this implementation the main goal was to implement an appropriate LAPI for the SS7Box location server, that mean fitting the server architecture and using as many resources available in the server as for example the Abyss web server or the libxml2 libraries.

The use of free software to implement the LAPI has helped during the development of the project due to the quantity of documentation available and the help given by the Internet community.

5.2. Future Lines

The LAPI implements the basic features of the SLIS in the MLP. The implementation of the SLIS could be extended adding new elements to be supported. Another option could be to implement other services of the MLP.

The LAPI needs also to have implemented some kind of client authentication. This authentication should be performed while processing the header of the MLP message. By the moment this has not been implemented because there are no clients.

The LAPI should implement some parsing in the Location Based Application. Due to the limitation of time the MLP Client only implement the MLP Transport Layer. This parsing can be based in the already existing parser in the Location Server.

For the maintenance of the LAPI, is also convenient to update the libxml2 and the XMLRPC-C libraries when new versions of them will be released.
REFERENCES

8. **RDC.** SS7Box - Architecture design version 1.1.
APPENDIX A: MLP CLIENT CODE

MlpClient

/**
 * \file MlpClient.h
 * \Author: Luis Alberto Gomez
 */

#ifndef CLIENT_H
#define CLIENT_H

#include <sstream>
using namespace std;
class MlpClient{
    private:
    protected:
    public:
        void CreateClient();
};
#endif
/**
 * \file MlpClient.cpp
 * \n * Author: Luis Alberto Gomez
 */

/**
 * C++ Libraries
 */
#include <cassert>
#include <cstdlib>
#include <string.h>
#include <iostream>
#include <stdio.h>

/*XMLRPC-C Libraries*/
#include <xmlrpc-c/girerr.hpp>
#include <xmlrpc-c/base.hpp>
#include <xmlrpc-c/client.hpp>
#include <xmlrpc-c/client_transport.hpp>

#include "MlpClient.h"
#include "MlpTemplates.h"

using namespace std;
using namespace xmlrpc_c;

void MlpClient::CreateClient()
{

MlpTemplates m_mlptemplate;

string const serverUrl("http://172.25.192.72:9210/RPC2");
string msid;

cout << "Insert msid: " << endl;
cin >> msid;

while (!strcmp(msid.c_str(), "quit")){

    string callXml = m_mlptemplate.DoMlpSvcInit(msid);
    string responseXmlP;

    carriageParm_http0 * carriage = new carriageParm_http0(serverUrl);
    carriageParm_http0 carriageParm(*carriage);
    cout << callXml << endl;
    try{
        transportP->call(&carriageParm, callXml, &responseXmlP);
    }catch (std::exception const e) {
        cerr << "call failed because " << e.what() << endl;
    }
}
cout << responseXmlP << endl;
cout << "Insert msid: " << endl;
cin >> msid;
}
MlpTemplates

/**
 * \file MlpTemplates.h
 * \Author: Luis Alberto Gomez
 */

#ifndef MLPTEMPLATES_H_
#define MLPTEMPLATES_H_

#include <iostream>
#include <string>
using namespace std;

class MlpTemplates{
private:
    string kindof; // right now: kindof could be timeout or coordinates
    string MlpMessage, Slir, Hdr;

protected:
    string DoSlir(string);
    string DoMlpHdr();

public:
    string GetKindof();
    void SetKindof(string);

    string GetMlpMessage();
    void SetMlpMessage(string);

    string DoMlpSvcInit(string);

};

#endif
/**
 * \file MlpTemplates.cpp
 * \Author: Luis Alberto Gomez
 */
#include "MlpTemplates.h"

string MlpTemplates::GetKindof()
{
    return kindof;
}

void MlpTemplates::SetKindof(string kind)
{
    kindof=kind;
}

string MlpTemplates::GetMlpMessage()
{
    return MlpMessage;
}

void MlpTemplates::SetMlpMessage(string msg)
{
    MlpMessage=msg;
}

string MlpTemplates::DoMlpHdr()
{
    Hdr.append("<hdr ver="3.2.0"/>");
    Hdr.append("<client>");
    Hdr.append("<id>MlpClient</id>");
    Hdr.append("</client>");
    Hdr.append("</hdr>");
    return Hdr;
}

string MlpTemplates::DoSlir(string msisdn)
{
    Slir.append("<slir ver="3.2.0" res_type="SYNC"/>");
    Slir.append("<msids>");
    Slir.append("<msid type="IMSI"">"+msisdn+"</msid>");
    Slir.append("</msids>");
    Slir.append("</slir>");
    return Slir;
}

string MlpTemplates::DoMlpSvcInit(string msisdn)
{
    MlpMessage="<?xml version = "1.0"?><!DOCTYPE svc_init SYSTEM 
"dtd/MLP_SVC_INIT_320.DTD"><svc_init ver="3.2.0"/>";
    MlpMessage.append(DoMlpHdr());
    MlpMessage.append(DoSlir(msisdn));
    MlpMessage.append("</svc_init>");
    return MlpMessage;
}
APPENDIX B: MODULE MLP CODE

TRANSPORT LAYER

HTTPServer

/**
 * \file HttpServer.h
 * \Author: Luis Alberto Gomez
 */

#ifndef HTTPSERVER_H
#define HTTPSERVER_H

#include <sstream>
#include "Configurable.h"
#include "Thread.h"
#include <xmlrpc-c/abyss.h>
#include <xmlrpc-c/server_abyss.h>
#include "uriHandlerMlp.h"

using namespace std;

class MlpHandler;
class SessionManager;

class HttpServer : public Configurable, public Thread{

private:
    int port; /**< port value should be 9120 */
    const char* uriPath;
    const char* log;
    SessionManager* m_sessionManager;
    int m_src;
    MlpHandler* m_mlphandler;

protected:
    virtual const char* getXConfigName(){ return "http_server";}

    virtual void configure(const XmlConfig &config);
    int FromString(const std::string& s, std::ios_base& (*f)(std::ios_base&));
    void SetServerHandler(TServer *const srvP, const char* uriPath);

public:
    HttpServer (SessionManager* sessionManager, int src);
};

#endif
/**
 * \file HttpServer.cpp
 *
 * \Author: Luis Alberto Gomez
 */

#include <cassert>
#include <stdexcept>
#include <iostream>
#include <string>

#include "Log.h"
#include "HttpServer.h"
#include "SessionManager.h"
#include "MlpHandler.h"
#include "SendertoLBS.h"
#include "XmlConfig.h"
#include <xmlrpc.h>

/**
 * Constructor
 * @param sessionManager is the SessionManager object shared by both threads
 * @param src is the MLP module queue
 */
HttpServer::HttpServer(SessionManager* sessionManager, int src)
{
    m_sessionManager = sessionManager;
    m_src = src;
    m_mlphandler = new MlpHandler();
}

/**
 * Read rom the mlpconfigure.xml the parameters of the server
 */
void
HttpServer::configure(const XmlConfig &config)
{
    std::string sport = config.readString("port");
    port = FromString(sport, std::dec);
    log = "abyss.log"; //config.readString("log");
    uriPath = "/RPC2";
}

/**
 * Converts a string to a int
 * @param s the string to convert
 * @param *f the base of the number
 * @return the int value of the string
 */
int
HttpServer::FromString(const std::string& s,
                        std::ios_base& (*f)(std::ios_base&))
{
    int t;
    std::istringstream iss(s);
    iss >> f >> t;
    return t;
}
/**
 * This method Set the Mlp handler using ServerAddHandler2 function from abyss
 * @param srvP a pointer to the server
 * @param uriPath the URI of the server
 * @see uriHandlerMlp
 */
void HttpServer::SetServerHandler(TServer *const srvP, const char *uriPath)
{
    struct uriHandlerMlp * uriHandlerMlpP;
    URIHandler2 uriHandler;
    abyss_bool success;

    uriHandlerMlpP->sessionManager = m_sessionManager;
    uriHandlerMlpP->uriPath       = strdup(uriPath);
    uriHandlerMlpP->src = m_src;

    uriHandler.handleReq2 = m_mlphandler->MlpHandlerRun;
    uriHandler.handleReq1 = NULL;
    uriHandler.userdata   = uriHandlerMlpP;
    uriHandler.init       = NULL;
    uriHandler.term       = NULL;

    ServerAddHandler2(srvP, &uriHandler, &success);

    if (!success)
        LOG_ERROR(ExInfo("Abyss failed to register the mlp "
                         "handler. ServerAddHandler2() failed.");
    }
/**
 * This method starts the running of the Abyss server. It is a
 * virtual method inherited from the class Thread
 */
void HttpServer::run()
{
    try {
        LOG_DEBUG(ExInfo("INTO RUNSERVER"));
        TServer abyssServer;

        ServerCreate(&abyssServer, "MlpServer", port, uriPath, log);
        ServerDefaultHandler(&abyssServer, m_mlphandler->MyDefaultHandler);
        SetServerHandler(&abyssServer, uriPath);
        ServerInit(&abyssServer);

        LOG_DEBUG(ExInfo("SERVER RUNNING"));
        ServerRun(&abyssServer);
    } catch (exception const& e) {
        cerr << "Something failed. " << e.what() << endl;
    }
}
MlpHandler

/**
 * ile MlpHandler.h
 * 
 * \Author: Luis Alberto Gomez
 */

#ifndef MLP_HANDLER_H
#define MLP_HANDLER_H

#include <sstream>
#include "uriHandlerMlp.h"
#include <xmlrpc-c/abyss.h>
#include <xmlrpc-c/server_abyss.h>

using namespace std;

class MlpHandler{

protected:

private:

    static void ValidateContentType(TSession * const abyssSessionP, unsigned int * const httpErrorP);
    static void ProcessContentLength(TSession * const httpRequestP, size_t * const inputLenP, unsigned int * const httpErrorP);

public:
    static abyss_bool MyDefaultHandler(TSession * const sessionP);
    static void SendError(TSession * const abyssSessionP, unsigned int const status);
    static void MlpHandlerRun(struct URIHandler2 *, TSession * abyssSession, abyss_bool * handledP);

};

#endif
/**
* \file MlpHandler.cpp
* Authors: Luis Alberto Gomez
*/

#include <cassert>
#include <stdexcept>
#include <iostream>
#include <string>
#include <stdio.h>
#include "Log.h"
#include "MlpHandler.h"
#include "SessionManager.h"
#include "SendertoLBS.h"
#include "XmlConfig.h"
#include "uriHandlerMlp.h"
#include "MlpParser.h"

/**
* This method is the modified default handler for the abyss server
* It will return an 404 HTTP error to the client in case the request
* arriving to the server cannot be handled by any Handler of the
* server
*/
abyss_bool
MlpHandler::MyDefaultHandler(TSession * const sessionP)
{
    LOG_DEBUG(ExInfo("INTO DEFAULT HANDLER"));
    ResponseStatus(sessionP, 404); /* HTTP server error */
    ResponseError(sessionP);
    return TRUE;
}

/**
* Method from xmlrpc_server_abyss.c
* If the client didn't specify a content-type of "text/xml", return
* "400 Bad Request". We can't allow the client to default this
* header,
* because some firewall software may rely on all XML-RPC requests
* using the POST method and a content-type of "text/xml"
* @param abyssSessionP identifies the connection to the server
* @param httpErrorP
*/
void
MlpHandler::ValidateContentType(TSession * const abyssSessionP,
    unsigned int * const httpErrorP) {

    const char * const content_type =
        RequestHeaderValue(abyssSessionP, "content-type");

    if (content_type == NULL)
        *httpErrorP = 400;
    else {
        const char * const sempos = strchr(content_type, ';');
        unsigned int baselen;
LOG_DEBUG(ExInfo("CONTENT TYPE").addInfo("content-type", content_type));
if (slen)
    baselen = slen - content_type;
else
    baselen = strlen(content_type);
if (strncmp(content_type, "text/xml", baselen) != 0)
    *httpErrorP = 400;
else
    *httpErrorP = 0;
}
/**
 * Method from xmlrpc_server_abyss.c
 * Make sure the content length is present and non-zero. This is
 * technically required by XML-RPC, but we only enforce it because we
 * don't want to figure out how to safely handle HTTP < 1.1 requests
 * without it. If the length is missing, return "411 Length Required"
 */
void
MlpHandler::ProcessContentLength(TSession * const httpRequestP, size_t * const inputLenP,
                                    unsigned int * const httpErrorP)
{
    const char * const content_length =
        RequestHeaderValue(httpRequestP, "content-length");
    if (content_length == NULL)
        *httpErrorP = 411;
    else {
        if (content_length[0] == '\0')
            *httpErrorP = 400;
        else {
            unsigned long contentLengthValue;
            char * tail;

            contentLengthValue = strtoul(content_length, &tail, 10);
            if (*tail != '\0')
                *httpErrorP = 400;
            else if (contentLengthValue < 1)
                *httpErrorP = 400;
            else if (((unsigned long)(size_t)contentLengthValue
                          != contentLengthValue)
                        *httpErrorP = 400;
            else {
                *httpErrorP = 0;
                *inputLenP = (size_t)contentLengthValue;
            }
        }
    }
}
/**
 * Method form xmlrpc_server_abyss.c
 * Send an error response back to the client.
 */

void MlpHandler::SendError(TSession * const abyssSessionP, unsigned int const status) {
    ResponseStatus(abyssSessionP, (uint16_t) status);
    ResponseError(abyssSessionP);
}

/**
 * This method is de MLP handler for the abyss server
 *
 * Make sure that the HTTP fit the MLP requirements.
 * In case the requirements are accomplished it call the method
 * parseMLP from the class MlpParser
 *
 * @see MlpParser
 */

void MlpHandler::MlpHandlerRun(struct URIHandler2 * urihandler,TSession * abyssSessionP, abyss_bool * handledP) {
    MlpParser m_mlpParser;
    uriHandlerMlp * const uriHandlerMlp = (uriHandlerMlp *) urihandler->userdata;
    const TRequestInfo * requestInfoP;
    SessionGetRequestInfo(abyssSessionP, &requestInfoP);
    LOG_DEBUG(ExInfo("INTO MLP HANDLER").addInfo("uriPath", requestInfoP->uri));
    if (strcmp(requestInfoP->uri, uriHandlerMlp->uriPath) != 0){
        *handledP = FALSE;
    } else {
        *handledP = TRUE;
        LOG_DEBUG(ExInfo("POST?").addInfo("method", requestInfoP->method));
        if (requestInfoP->method != m_post)
            MlpHandler::SendError(abyssSessionP, 405);
        else {
            unsigned int httpError;
            MlpHandler::ValidateContentType(abyssSessionP, &httpError);
            LOG_DEBUG(ExInfo("CONTENT TYPE").addInfo("httperror", httpError));
            if (httpError)
                MlpHandler::SendError(abyssSessionP, httpError);
            else {
                size_t contentSize;
                MlpHandler::ProcessContentLength(abyssSessionP, &contentSize, &httpError);
                LOG_DEBUG(ExInfo("CONTENT LENGTH").addInfo("httperror", httpError));
                if (httpError)
                    MlpHandler::SendError(abyssSessionP, httpError);
                else {
                    LOG_DEBUG(ExInfo("CALLING PARSER"));
                } else {
                    LOG_DEBUG(ExInfo("CALLING PARSER"));
                }
            }
        }
    }
}

/*
m_mlpParser.parseMLP(abyssSessionP,
    contentSize, uriHandlerMlpP->sessionManager, uriHandlerMlpP->src);
}
}
}
}

UriHandlerMLP

#ifndef URI_HANDLER_MLP_H
#define URI_HANDLER_MLP_H

using namespace std;

class MlpHandler;
class SessionManager;

/**
  * urihandlerMLp struct.
  * It defines the necessary information to work
  * with the MLP message in the module_mlp
  * This struct contains the information passed to
  * the handler
  */

struct uriHandlerMlp{
    SessionManager* sessionManager; /**< SessionManager object
            passed from the server to the handler */
    const char * uriPath; /**< url for the RPC */
    int src; /**< module_mlp queue*/
};

#endif
PARSER LAYER

MlpParser

/**
   * \file MlpParser.h
   * \Author: Luis Alberto Gomez
   */

#ifndef MLP_PARSER_H
#define MLP_PARSER_H

#include <sstream>

/**
   * libxml2 headers
   */
#include <libxml/tree.h>
#include <libxml/parser.h>
#include <libxml/parserInternals.h>

using namespace std;

class SendertoLBS;
class SessionDataMLP;

class MlpParser{

private:
    xmlDocPtr doc; /**< MLP tree*/
SendertoLBS* LbsDialog;
SessionDataMLP* m_sessionDataMLP;

protected:
    void getBody(xmlrpc_env * const envP, TSession * const abyssSessionP, size_t const contentSize, xmlrpc_mem_block ** const bodyP);
    void refillBufferFromConnection(TSession *const abyssSessionP);
    void genResponse(TSession* abyssSessionP, const char* body, size_t const len);

public:
    void parseMLP(TSession* abyssSessionP, size_t contentSize, SessionManager* sessionManager, int src);
    MlpParser();
};

#endif
/**
 * \file MlpParser.cpp
 * \Author: Luis Alberto Gomez
 */

#include <cassert>
#include <stdexcept>
#include <iostream>
#include <string>

/**
 * SS7 Box Headers
 */
#include "Log.h"
#include "HttpServer.h"
#include "SessionManager.h"
#include "SendertoLBS.h"
#include "MlpParser.h"
#include "MlpHandler.h"
#include "xmlParser.h"
#include "XmlConfig.h"
#include "MlpValidate.h"
#include "MLPServices.h"

/**
 * XML-RPC Headers
 */
#include <xmlrpc.h>
#include <xmlrpc-c/abyss.h>
#include <xmlrpc-c/server_abyss.h>
#include <xmlrpc-c/util.h>

using namespace std;

/**
 * Constructor.
 * It creates a SessionDataMLP object
 */
MlpParser::MlpParser(){
    m_sessionDataMLP = new SessionDataMLP(1);
}

/**
 * Get the next chunk of data from the connection into the buffer.
 */
void MlpParser::refillBufferFromConnection(TSession * const abyssSessionP)
{
    abyss_bool succeeded;

    succeeded = SessionRefillBuffer(abyssSessionP); //abyss.h

    if (!succeeded)
        LOG_ERROR(ExInfo("Timed out waiting for client to send its
        POST data"));
}
/**
 * GetBody from xmlrpc_server_abyss.c
 * Get the entire body, which is of size 'contentSize' bytes, from the
 * Abyss session and return it as the new memblock *bodyP.
 * 
 * The first chunk of the body may already be in Abyss's buffer. We
 * retrieve that before reading more.
 */

void MlpParser::getBody(xmlrpc_env * const envP, TSession * const abyssSessionP, size_t const contentSize, xmlrpc_mem_block ** const bodyP)
{
    LOG_DEBUG(ExInfo("INSIDE GETBODY"));
    xmlrpc_mem_block * body;
    body = xmlrpc_mem_block_new(envP, 0);

    if(!envP->fault_occurred){
        size_t bytesRead;
        const char * chunkPtr;
        size_t chunkLen;
        bytesRead = 0;
        while (!envP->fault_occurred && bytesRead < contentSize) {
            SessionGetReadData(abyssSessionP, contentSize - bytesRead, &chunkPtr, &chunkLen); //abyss.h
            bytesRead += chunkLen;
            assert(bytesRead <= contentSize);
            XMLRPC_MEMBLOCK_APPEND(char, envP, body, chunkPtr, chunkLen);
            if(bytesRead < contentSize)
                refillBufferFromConnection(abyssSessionP);
        }
        LOG_DEBUG(ExInfo("BODY READ"));
        if(envP->fault_occurred)
            xmlrpc_mem_block_free(body);
        else
            *bodyP = body;
    }
}

/**
 * Generates the response message to the client
 * @param abyssSessionP identifies the session where to return the
 * response
 * @param body identifies the message
 * @param len identifies the length of the message
 */

void MlpParser::genResponse(TSession* abyssSessionP, const char* body, size_t const len)
{
    LOG_DEBUG(ExInfo("INSIDE genResponse"));
    ResponseStatus(abyssSessionP, 200);

    if ((size_t)(uint32_t)len != len)
        LOG_DEBUG(ExInfo("Response too large"));

    uint32_t const abyssLen = (uint32_t)len;
}
ResponseContentType(abyssSessionP, "text/xml; charset="utf-8");
ResponseContentLength(abyssSessionP, abyssLen);
ResponseWriteStart(abyssSessionP);
ResponseWriteBody(abyssSessionP, body, abyssLen);
ResponseWriteEnd(abyssSessionP);

/**
 * This method obtain the MLP message from the HTTP request
 * @param abyssSessionP identifies the connection
 * @param contentSize identifies the length
 * @param m_sessionManager is the SessionManager object
 * @param m_src identifies de MLP module queue
 */
MlpParser::parseMLP(TSession* abyssSessionP, size_t contentSize,
SessionManager* m_sessionManager, int m_src)
{
    LOG_DEBUG(ExInfo("INSIDE PARSER"));
    /** Local variables */
    xmlrpc_mem_block * body;
    xmlParser m_xmlParser;
    LbsDialog = new SendToLBS(m_sessionManager, m_src);
    /**
     * XML-RPC Error Handler. As we are using some methods from XML-RPC
     * in this part. We use its error handler.
     */
    xmlrpc_env env;
    xmlrpc_env_init(&env);
    LOG_DEBUG(ExInfo("INSIDE MLPPARSER"));
    getBody(&env, abyssSessionP, contentSize, &body);
    if(!env.fault_occurred){
        XMLRPC_MEMBLOCK_RESIZE(char, &env, body, contentSize);
        const char * mlpmsg = XMLRPC_MEMBLOCK_CONTENTS(char, body);
        LOG_DEBUG(ExInfo("AFTER GETBODY").addInfo("body", mlpmsg));
    }
    /**
     * Before process the MLP message we should validate it
     * against the DTD
     * Using the validator object we validate and obtain the
     * parsed document
     */
    LIBXML_TEST_VERSION
    MlpValidate validator;
    int flag = 0;
    doc = validator.parseMlp(mlpmsg, contentSize, &flag, true);
    cout << "Flag" << flag << endl;
    if(doc==NULL){
        LOG_DEBUG(ExInfo("Unable to validate and parse MLP.
       Returning error to the client"));
    }
if(flag==1){
    m_sessionDataMLP->GetMLPServices()->
    SetResultTag("105","FORMAT ERROR");
    string resp = m_sessionDataMLP->
    GetMLPServices()->DoMLPMessage();
    genResponse(abyssSessionP, resp.c_str(),
    resp.length());
}
else if(flag==2){
    m_sessionDataMLP->GetMLPServices()->
    SetResultTag("104","SYNTAX ERROR");
    string resp = m_sessionDataMLP->
    GetMLPServices()->DoMLPMessage();
    genResponse(abyssSessionP, resp.c_str(),
    resp.length());
} else{
    m_sessionDataMLP->GetMLPServices()->
    SetResultTag("2","UNSPECIFIED ERROR");
    string resp = m_sessionDataMLP->
    GetMLPServices()->DoMLPMessage();
    genResponse(abyssSessionP, resp.c_str(),
    resp.length());
}
else{
    LOG_DEBUG(ExInfo("CALLING XMLPARSER"));
    /**
     * Here we call the xmlParser which will process the
     * MLP XML obtained from the
     * body of the HTTP POST if everything is correct we
     * will receive the IMSI else we receive
     * NULL.
     */
    bool processed = m_xmlParser.processMlp(doc,
    m_sessionDataMLP);
    if(processed)
    {
        /**
         * Here we call the SendertoLBS which will ask
         * for the location
         * The program wait here until the dialog
         * between MLP & LBS modules have finished
         */
        string resp=LbsDialog->
        sender(m_sessionDataMLP);
        // Response to the client
        LOG_DEBUG(ExInfo("RESPONSE MESSAGE").addInfo("mlpresp", resp));
        genResponse(abyssSessionP, resp.c_str(),
        resp.length());
    }
    else{ // Error processing return error message
        string resp = m_sessionDataMLP->
        GetMLPServices()->DoMLPMessage();
        genResponse(abyssSessionP, resp.c_str(),
        resp.length());
    }
}
** In case of error we should report to the client */

if (env.fault_code == XMLRPC_TIMEOUT_ERROR) {
    m_sessionDataMLP->GetMLPServices()->SetResultTag("7", "TIMEOUT");
    string resp = m_sessionDataMLP->GetMLPMessage();
    genResponse(abyssSessionP, resp.c_str(), resp.length());
}
else{
    m_sessionDataMLP->GetMLPMessage()->SetResultTag("2", "UNSPECIFIED ERROR");
    string resp = m_sessionDataMLP->GetMLPMessage();
    genResponse(abyssSessionP, resp.c_str(), resp.length());
}
xmlParser

/**
 * \file xmlParser.h
 * \Author: Luis Alberto Gomez
 */

#ifndef XML_PARSER_H
#define XML_PARSER_H

#include <sstream>
#include <libxml/tree.h>
#include <libxml/parser.h>
#include <libxml/parserInternals.h>

using namespace std;

class xmlParser{

private:
    xmlNodePtr root_node; /**< Root node from the XML tree*/
    xmlNodePtr hdr_node; /**< The hdr node from the XML tree*/
    xmlNodePtr svc_node; /**< The svc node from the XML tree*/

protected:
    void getMlpNodes(xmlDocPtr);
    void processHdr(SessionDataMLP*, bool*);
    void processSvc(SessionDataMLP*, bool*);

public:
    bool processMlp(xmlDocPtr, SessionDataMLP*);
};

#endif
/**
 * \file xmlParser.cpp
 * \Author: Luis Alberto Gomez
 */

#include <cassert>
#include <stdexcept>
#include <iostream>
#include <string>

/**
 * SS7 Box Headers
 */
#include "Log.h"
#include "HttpServer.h"
#include "SessionManager.h"
#include "SessionDataMLP.h"
#include "SendertoLBS.h"
#include "xmlParser.h"
#include "XmlConfig.h"
#include "MLPServices.h"

using namespace std;

/**
 * Get header and service nodes
 * from the XML tree and stor them in hdr_node and svc_node
 * @param doc the XML tree
 */
void xmlParser::getMlpNodes(xmlDocPtr doc)
{
    root_node = xmlDocGetRootElement(doc);
    LOG_DEBUG(ExInfo("ROOT ELEMENT").addInfo("root_node_name",
        (const char*)root_node->name));
    hdr_node = root_node->xmlChildrenNode;
    svc_node = hdr_node->next;
}

/**
 * Procees the header node
 * @param m_sessionDataMLP stores the info from the node
 * @param res indicates true if the node has been processed
 */
void xmlParser::processHdr(SessionDataMLP* m_sessionDataMLP, bool *res)
{
    if(!strcmp((const char*)hdr_node->xmlChildrenNode->name, "client")){
        if(!strcmp((const char*)hdr_node->xmlChildrenNode->xmlChildrenNode->name, "id")){
            const char* id = (const char*)
                xmlNodeGetContent(hdr_node->xmlChildrenNode->xmlChildrenNode);
            m_sessionDataMLP->GetMLPServices()->SetClientTag("id", id);
            LOG_DEBUG(ExInfo("ID").addInfo("id", id));
            if(hdr_node->xmlChildrenNode->xmlChildrenNode->next!=NULL){
                LOG_DEBUG(ExInfo("PWD"));
            }
        }
    }
}
const char* pwd = (const char*)xmlNodeGetContent(hdr_node->xmlChildrenNode->xmlChildrenNode->next);
if (m_sessionDataMLP->GetMLPServices()->SetClientTag("pwd",pwd);
} else{
    LOG_DEBUG(ExInfo("NO PWD"));
}
else{
    m_sessionDataMLP->GetMLPServices()->SetResultTag("107","PROTOCOL ELEMENT NOT SUPPORTED");
    *res = false;
}
else{
    m_sessionDataMLP->GetMLPServices()->SetResultTag("107","PROTOCOL ELEMENT NOT SUPPORTED");
    *res = false;
}

void xmlParser::processSvc(SessionDataMLP* m_sessionDataMLP, bool* res)
{
    m_sessionDataMLP->GetMLPServices()->SetService((const char*)svc_node->name);
    if (!m_sessionDataMLP->GetMLPServices()->GetService().compare("slir")){
        if (!strcmp((const char*)svc_node->xmlChildrenNode->name, "msids"){
            if (!strcmp((const char*) svc_node->xmlChildrenNode->name, "msid"){
                const char* msid = (const char*)xmlNodeGetContent(svc_node->xmlChildrenNode->xmlChildrenNode);
                const char* type = (const char*)xmlGetProp(svc_node->xmlChildrenNode->xmlChildrenNode, (xmlChar*)"type");
                if (!strcmp(type, "IMSI")){
                    m_sessionDataMLP->GetMLPServices()->SetMsids(msid,type);
                    *res = true;
                }
            } else{
                m_sessionDataMLP->GetMLPServices()->SetResultTag("113"," PROTOCOL ELEMENT ATTRIBUTE VALUE NOT SUPPORTED");
                *res = false;
            }
        } else{
            m_sessionDataMLP->GetMLPServices()->SetResultTag("107","PROTOCOL ELEMENT NOT SUPPORTED");
            *res = false;
        }
    }
} else{
m_sessionDataMLP->GetMLPServices()->SetResultTag("107", "PROTOCOL ELEMENT NOT SUPPORTED");
*res = false;
}
}
else{
    m_sessionDataMLP->GetMLPServices()->SetResultTag("107", "PROTOCOL ELEMENT NOT SUPPORTED");
    *res = false;
}
}
else{
    LOG_DEBUG(ExInfo("INSIDE ELSE").addInfo("node_name", (const char*) svc_node->xmlChildrenNode->name));
    m_sessionDataMLP->GetMLPServices()->SetService((const char*) svc_node->name);
    m_sessionDataMLP->GetMLPServices()->SetResultTag("108", "SERVICE NOT SUPPORTED");
    *res = false;
}

/**
 * Process the XML tree
 * @return true if the tree has been processed
 */
bool xmlParser::processMlp(xmlDocPtr doc, SessionDataMLP* m_sessionDataMLP) {
    bool result = false;
    getMlpNodes(doc);
    processHdr(m_sessionDataMLP, &result);

    if(result)
        processSvc(m_sessionDataMLP, &result);
    LOG_DEBUG(ExInfo("AFTER PROCESS SVC").addInfo("result", result));

    return result;
}
SessionDataMLP

/**
 * \file SessionDataMLP.h
 * \author Pep Soler
 */

#ifndef SESSIONDATAMLP_H_
#define SESSIONDATAMLP_H_

#include "SessionData.h"
#include <string>
#include <iostream>
#include <pthread.h>
#include <sstream>
#include <map>

using namespace std;

class MLPServices;

class SessionDataMLP : public SessionData {

private:
    pthread_mutex_t mutex; /**< Mutual exclusion. Used to fill the Session Data without data inconsistencies */
    pthread_cond_t condition_cond; /**< used to stop the HTTP thread while the dialog with the LBS */
    string cgi; /**< cgi value used to search in the ddbb*/
    string lac; /**< lac value used to search in the ddbb*/
    string ci; /**< ci value used to search in the ddbb*/
    MLPServices *m_MLPServices; /**< Pointer to the MLPServices object used to create the MLP message */

public:
    SessionDataMLP(int);
    ~SessionDataMLP();

    virtual void closeSession();
    void WaitComplete();
    void SignalComplete();
    MLPServices* GetMLPServices();
    string GetCgi();
    string SetCgi(string);
    string GetLac();
    string SetLac(string);
    string GetCi();
    string SetCi(string);
    int LockMutex();
    int UnlockMutex();
};

#endif /* SESSIONDATAMLP_H_ */
/**
 * \file SessionDataMLP.h
 * \author Pep Soler
 */

#ifndef SESSIONDATAMLPH_
#define SESSIONDATAMLPH_

#include "SessionData.h"
#include <string>
#include <iostream>
#include <pthread.h>
#include <sstream>
#include <map>

using namespace std;

class MLPServices;

class SessionDataMLP : public SessionData {
    private:
        pthread_mutex_t mutex; /**< Mutual exclusion. Used to fill the Session Data without data inconsistencies */
        pthread_cond_t condition_cond; /**< used to stop the HTTP thread while the dialog with the LBS */
        string cgi; /**< cgi value used to search in the ddbb*/
        string lac; /**< lac value used to search in the ddbb*/
        string ci; /**< ci value used to search in the ddbb*/

        MLPServices *m_MLPServices; /**< Pointer to the MLPServices object used to create the MLP message */

    public:
        SessionDataMLP(int);
        ~SessionDataMLP();

        virtual void closeSession();

        void WaitComplete();
        void SignalComplete();

        MLPServices* GetMLPServices();

        string GetCgi();
        string SetCgi(string);

        string GetLac();
        string SetLac(string);

        string GetCi();
        string SetCi(string);

        int LockMutex();
        int UnlockMutex();
};
SessionDataMLP::SessionDataMLP(int invoke): SessionData(invoke) {
    m_MLPServices = new MLPServices();
}

SessionDataMLP::~SessionDataMLP() {
    delete m_MLPServices;
}

void SessionDataMLP::closeSession(){
    //Delete the session!!!
}

/* Getters and Setters of all the variables*/
MLPServices*
SessionDataMLP::GetMLPServices(){
    return m_MLPServices;
}

string SessionDataMLP::GetCgi(){
    return cgi;
}

string SessionDataMLP::SetCgi(string id){
    cgi=id;
    return cgi;
}

string SessionDataMLP::GetLac(){
    return lac;
}

string SessionDataMLP::SetLac(string id){
    lac=id;
    return lac;
}

string SessionDataMLP::GetCi(){
    return ci;
}

string SessionDataMLP::SetCi(string id){
    ci=id;
    return ci;
}
void SessionDataMLP::WaitComplete(){
    pthread_cond_wait(&condition_cond, &mutex);
}

void SessionDataMLP::SignalComplete(){
    pthread_cond_signal(&condition_cond);
}

/**
 * It is used to lock and unlock the SessionData when it is filled
 */
int SessionDataMLP::LockMutex()
{
    pthread_mutex_lock(&mutex);
    return 0;
}

int SessionDataMLP::UnlockMutex()
{
    pthread_mutex_unlock(&mutex);
    return 0;
}
MLPServices

/** \ile MLPServices.h
 * \author Pep Soler
 */

#ifndef MLPSERVICES_H_
define MLPSERVICES_H_
#include <string>
#include "ELayer.h"

using namespace std;

class MLPServices : public ELayer {
public:
    string DoMLPMessage();
    void SetService(const char* ser){service=ser;}
    string GetService() {return service;}
    int PutDocType();
    int PutSvcResult();
    int PutSlir();
    int PutSlia();
    void SetslirTag(string, string);
    string GetslirTag(string);
    void SetslirTag(string ver){ fillSlia.version=ver;}
    string GetslirTag(){return fillSlia.version;}
}
#endif /* MLPSERVICES_H_ */
/**
 * \file MLPServices.cpp
 * \author Pep Soler
 */

#include "MLPServices.h"
#include <string>
#include <iostream>

using namespace std;

/**
 * \brief description of the constructor
 * \description of the constructor
 * \The constructor fill the version of the SLIR and SLIA.
 */
MLPServices::MLPServices(): ELayer()
{
    fillSlir.version="3.2.0";
    fillSlia.version="3.0.0";
}

/* The Put functions fill the mlpMsg with the MLP tags */
int MLPServices::PutSlir()
{
    SetmlpMsg("<slir ver="3.2.0"
res_type=""+fillSlir.res_type+">
    PushstackTags("slir");
    return 0;
}

int MLPServices::PutDocType()
{
    SetmlpMsg("<!DOCTYPE svc_result SYSTEM
"dtd/MLP_SVC_RESULT_320.DTD">
    return 0;
}

int MLPServices::PutSvcResult()
{
    SetmlpMsg("<svc_result ver=""+fillSlir.version+">
    PushstackTags("svc_result");
    return 0;
}

int MLPServices::PutSlia()
{
    SetmlpMsg("<slia ver=""+fillSlia.version+">
    PushstackTags("slia");
    return 0;
}
/** Getters and Setters */
void 
MLPServices::SetslirTag(string res, string ver){
    fillSlir.version=ver;
    fillSlir.res_type=res;
}

string 
MLPServices::GetslirTag(string which){
    if (!which.compare("res_type"){ return fillSlir.res_type;}
    else if (!which.compare("version")){ return fillSlir.version;}
}

/**
* This function develope the MLP message.
*/
string 
MLPServices::DoMLPMessage(){
    cout<<"Inside DoMLPMessage"<<endl;
    if (!GetResultTag("resid").compare("0"))
    
        PutHeaderXml();
        PutDocType();
        PutSvcResult();
        PutSlia();
        PutPos();
        PutMsid();
        PutPd();
        PutTime();
        PutShape();
        PutPoint();
        PutCoord();
        FinishTag();
    }
    else { /* then it is an error message */
        PutHeaderXml();
        PutDocType();
        PutSvcResult();
        PutSlia();
        PutPos();
        PutMsid();
        PutPoserr();
        FinishTag();
    }
    return GetmlpMsg();
}


ELayer

/**
 * \file ELayer.h
 * \author Pep Soler
 */

#ifndef ELAYER_H_
#define ELAYER_H_

#include <stack>
#include <string>
using namespace std;
typedef stack<string> StackStrings;

class ELayer {

private:
/**
 * A msid tag is defined by an ID and a type.
 * Defined in Identity Elements Definitions
 * in OMA-TS-MLP-V3_2-20051124-C
 */
struct msidsTag{
    const char* msidID;
    const char* msidType;
}fillMsidsTag;
/**
 * The coord tag is defined by a X, Y and Z.
 * Y and Z are optional. Defined in Shape Elements
 * Definitions in OMA-TS-MLP-V3_2-20051124-C
 */
struct coordTag{
    string X;
    string Y;
}fillCoordTag;
/**
 * Define the type of location request.
 * Defined in Function Elements Definitions
 * in OMA-TS-MLP-V3_2-20051124-C
 */
struct loc_typeTag{
    string loc_typeType;
}fillLoc_typeTag;
/**
 * Indicates the result of the request.
 * Defined in Result Elements Definitions
 * in OMA-TS-MLP-V3_2-20051124-C
 */
struct resultTag{
    string resid;
    string message;
}fillResultTag;
/**
 * Defines an id of the client and optionally a password,
 * serviceid and requestmode. Defined in Function Elements
 */
struct clientTag {
    const char * id;
    const char * pwd;
} fillClient;

/**
 * This is a stack of each open tag.
 * When we open a tag we push it,
 * when we close the tag we pop it
 */
StackStrings stackTags;

/**
 * This is the MLP message sent to the client
 */
string mlpMsg;

/**
 * Defines the indentation
 */
int steps;

public:
int SetCoord(string, string);
string GetCoord(string);

void SetMsids(const char*, const char*);
string GetMsids();

void Setloc_typeTag(string type){
    fillLoc_typeTag.loc_typeType=type;}
string Getloc_typeTag () {return
    fillLoc_typeTag.loc_typeType;}

int SetResultTag(string, string);
string GetResultTag(string);

string toString(int);
string GetTime();

void FinishTag();
void PutClient();
void PutHeaderMlp();
void PutHeaderXml();
void PutPos();
void PutMsid();
void PutPd();
void PutTime();
void PutShape();
void PutPoint();
void PutCoord();
void PutPoserr();

void SetClientTag(string, const char*);
string GetClientTag(string);

void PopstackTags (StackStrings){ stackTags.pop();}
void PushstackTags (string tag) {stackTags.push(tag);};
StackStrings GetstackTags(){ return stackTags;}


void SetmlpMsg (string msg){ mlpMsg.append(msg);}
string GetmlpMsg () {return mlpMsg;}
ELayer();

};

#endif /* ELAYER_H_ */
/**
* \file ELayer.cpp
* \authors:Pep Soler
* \ Luis Alberto Gomez
*/

#include "ELayer.h"
#include <string>
#include <iostream>
#include <sstream>
#include <time.h>

/**
* \brief description of the constructor
* \description of the constructor
* The constructor clear the mlpMsg string and does steps equal zero
*/
ELayer::ELayer()
{
    GetmlpMsg().clear();
}

/**
* \brief Getters and Setters *
*/

/**
* \brief Define the value of the fillMsidsTag structure
* \param msids is the ID of the user. Our implementation the ID is
defined by a IMSI.
* \param type defines the type of the ID, in our case equal IMSI.
*/
void
ELayer::SetMsids(const char* msids, const char* type)
{
    fillMsidsTag.msidID = msids;
    fillMsidsTag.msidType = type;
}

/**
* \brief Get the value of the msid
* \return the IMSI value.
*/
string
ELayer::GetMsids()
{
    return fillMsidsTag.msidID;
}

/**
* \brief Define the value of the fillCoord structure
* \param coordX is the latitude
* \param coordY is the latitude
*/

int
ELayer::SetCoord(string coordX, string coordY)
{
    fillCoordTag.X = coordX;
    fillCoordTag.Y = coordY;
    return 0;
}

/**
* Get the value of the coord
* @param coord the coordinate you want to obtain
* @return the coordinate value.
*/
string
ELayer::GetCoord(string coord)
{
    if (!coord.compare("latitude")){
        return fillCoordTag.X;
    } else if (!coord.compare("longitude")){
        return fillCoordTag.Y;
    }
}

/**
* Define the value of the fillResultTag structure
* @param rid is the result code of the MLP
* @param msg is the slogan of the MLP result code
*/
int
ELayer::SetResultTag(string rid, string msg)
{
    fillResultTag.resid = rid;
    fillResultTag.message = msg;
    return 0;
}

/**
* Get the value or the slogan of the result code
* @param which what you want to obtain
* @return the result code or the slogan.
*/
string
ELayer::GetResultTag(string which)
{
    if (!which.compare("resid")){
        return fillResultTag.resid;
    } else if (!which.compare("message")){
        return fillResultTag.message;
    }
}

string
ELayer::toString(int num)
{
    string s;
    stringstream out;
    out << num;
    s = out.str();
    return s;
}
/**
 * Obtain the time with the structure defined by MLP
 * @return the time in the form yyyyMMddhhmmss
 */
string
ELayer::GetTime()
{
    time_t rawtime;
    tm * ptm;

time(&rawtime);

ptm = gmtime(&rawtime);

string time = toString(ptm->tm_year+1900);
if(ptm->tm_mon < 9) time.append("0");
time.append(toString(ptm->tm_mon+1));
if(ptm->tm_mday < 10) time.append("0");
time.append(toString(ptm->tm_mday));
if(ptm->tm_hour < 10) time.append("0");
time.append(toString(ptm->tm_hour));
if(ptm->tm_min < 10) time.append("0");
time.append(toString(ptm->tm_min));
if(ptm->tm_sec < 10) time.append("0");
time.append(toString(ptm->tm_sec));

return time;
}

/**
 * Define the value of the fillClient structure
 * @param which is the parameter of the structure to fill
 * @param value is the id or pwd value
 */
void
ELayer::SetClientTag(string which, const char* value){
    if(!which.compare("id")){ fillClient.id=value;}
    else if (!which.compare("pwd")) {fillClient.pwd=value;}
}

string
ELayer::GetClientTag(string which){
    if (!which.compare("id")) { return fillClient.id;}
    else if (!which.compare("pwd")) { return fillClient.pwd;}
}

/**
 * The Put functions fill the mlpMsg with the MLP tags
 */
void
ELayer::PutHeaderXml()
{
    mlpMsg.append("<?xml version="1.0"?>");
    steps++;
void ELayer::PutMsid()
{  
    mlpMsg.append("<msid>" + GetMsids() + "</msid>"疏);  
}

void ELayer::PutPd()
{  
    /** open the tag */  
    mlpMsg.append("<pd>");  
    /** Push of the tag into the satck */  
    stackTags.push("pd");  
}

void ELayer::PutShape()
{  
    mlpMsg.append("<shape>");  
    stackTags.push("shape");  
}

void ELayer::PutPoint()
{  
    mlpMsg.append("<Point>");  
    stackTags.push("Point");  
}

void ELayer::PutPoserr()
{  
    mlpMsg.append("<poserr>");  
    mlpMsg.append("<result resid=" + GetResultTag("resid") + ">" + GetResultTag("message") + "</result>");  
    mlpMsg.append("<time utc_off="+0200">" + GetTime() + "</time>");  
    mlpMsg.append("</poserr>");  
}

void ELayer::PutTime()
{  
    mlpMsg.append("<time utc_off="+0200">" + GetTime() + "</time>");  
}

void ELayer::PutCoord()
{  
    mlpMsg.append("<coord>");  
    mlpMsg.append("<X>" + GetCoord("latitude") + "</X>");  
    mlpMsg.append("<Y>" + GetCoord("longitude") + "</Y>");  
    mlpMsg.append("</coord>");  
}

void ELayer::PutPos(){
    mlpMsg.append("<pos>");  
    stackTags.push("pos");  
}

void ELayer::PutHeaderMlp()
{    
    mlpMsg.append("<hdr ver="+3.2.0">");  
    stackTags.push("hdr");  
}
void
ELayer::PutClient()
{
    mlpMsg.append("<client>");
    mlpMsg.append("<id>"+GetClientTag("id")+"</id>");
    mlpMsg.append("<pwd>"+GetClientTag("pwd")+"</pwd>");
    mlpMsg.append("</client>");
}

/**
 * Close the open tags
 */
void
ELayer::FinishTag()
{
    while(stackTags.size() != 0)
    {
        mlpMsg.append("</");
        mlpMsg.append(stackTags.top().c_str());
        mlpMsg.append(">");
        stackTags.pop();
    }
}
APPLICATION LAYER

SendertoLBS

/**
 * \file SendertoLBS.h
 * \Author: Luis Alberto Gomez
 */

#ifndef HEADER_SAMPLEADDMETHOD_H
#define HEADER_SAMPLEADDMETHOD_H

#include <string>
#include "SessionManager.h"
#include "SessionDataMLP.h"
#include "BaseSender.h"

using namespace std;

class SessionManager;

class SessionDataMLP;

class SendertoLBS : public BaseSender{

private:
    SessionManager* m_sessionManager;
    string latitude, longitude;

protected:
    void toString(std::string &s, int const int_t);
    bool SendtoLBS(string, int);

public:
    SendertoLBS(SessionManager * sessionManager, int src);
    string sender(SessionDataMLP*);
};

#endif
/**
 * \file SendertoLBS.cpp
 * \Author: Luis Alberto Gomez
 */

#include <cassert>
#include <stdexcept>
#include <iostream>
#include <string>
#include <sstream>
#include <fronty/CAppMsg.h>
#include "HttpServer.h"
#include "SendertoLBS.h"
#include "SessionData.h"
#include "MyAppConstants.h"
#include "MLPServices.h"
#include "Log.h"

/**
 * Constructor
 */
SendertoLBS::SendertoLBS(SessionManager* sessionManager, int src) :
 BaseSender(src) {
    m_sessionManager = sessionManager;
}

void
SendertoLBS::toString(std::string &s, int const int_t) {
    std::stringstream out;
    out << int_t;
    s = out.str();
}

/**
 * This method generates and send the message
 * to the module LBS.
 * @return TRUE if the message is sent or FALSE if there is any error
 */
bool
SendertoLBS::SendtoLBS(string msid, int key) {
    CAppMsg* message = createMsg(LBS_QUERY_MSG_REQ, 1);
    addString(message, PRM_IMSI, msid);
    addUint32(message, PRM_MLP_SESSION_ID, (uint32_t) key);

    return sendMsg(LBS_QUEUE, message);
}

/**
 * This method perform the communication with
 * the LBS module
 * @return the MLP response message
 */
string SendtoLBS::sender(SessionDataMLP* m_sessionDataMLP)
{
    string responseMsg;

    /*Generate seesionId and Put the session data into the session manager */
    int key = m_sessionManager->keyGenerator();
    LOG_DEBUG(ExInfo("NEW KEY TO SESSIO DATA").addInfo("key",key));
    m_sessionManager->insertSession(key, m_sessionDataMLP);

    /* Send message to the LBS module */
    LOG_DEBUG(ExInfo("SENDING MESSAGE TO LBS MODULE").addInfo("key",key));

    if(!SendtoLBS(m_sessionDataMLP->GetMLPServices()->GetMsids(), key)){
        LOG_DEBUG(ExInfo("ERROR SENDING MESSAGE TO LBS. SENDING ERROR TO THE CLIENT").addInfo("key",key));
        m_sessionManager->deleteSession(key);
        m_sessionDataMLP->GetMLPServices()->SetResultTag("1", "SYSTEM FAILURE");
        responseMsg = m_sessionDataMLP->GetMLPServices()->DoMLPMessage();
    }else{
        LOG_DEBUG(ExInfo("WAITING_for_SDATA_WRITTEN").addInfo("key", key));
        m_sessionDataMLP->WaitComplete();

        LOG_DEBUG(ExInfo("SESSION DATA FILLED WITH RESPONSE").addInfo( "key", key));

        m_sessionDataMLP = m_sessionManager->getKeyData(key);
        responseMsg = m_sessionDataMLP->GetMLPServices()->DoMLPMessage();
    }

    m_sessionManager->deleteSession(key);

    return responseMsg;
}
ProcMlpQuery

/** \ile ProcMlpQuery.h
 * 
 * \author Pep Soler
 */

#ifndef HEADER_PROCMLPQUERY_H
#define HEADER_PROCMLPQUERY_H

class netResult;
class ParamValue;
class Dialog;
class SenderTimer;
class SessionManager;
class SessionDataMLP;
class DBConnection;
class DBResult;
class CMsg;
class MlpQueryMsg;

#include <string>
#include <sstream>
#include <iostream>

using namespace std;

class ProcMlpQuery {

private:
    SessionManager *m_sessionManager; /**< SessionManager to manage the sessions*/
    SessionDataMLP *m_sessionDataMLP; /**< sessionData to manage the sessions of each users*/
    DBConnection *m_dBConnection; /**< To do the queries to the database*/
    DBResult *m_dBResult; /**< To process the information from the database*/
    unsigned int idTmp; /**< To save the ID of the user temporarily*/
    char buf[30]; /**< A buffer where the CGI is saved*/
    string query; /**< A string to save the query which is sent to the database*/
    MlpQueryMsg *m_mlpQueryMsg; /**< MlpQueryMsg is the arriving message*/

public:
    ProcMlpQuery (SessionManager* sessionManager, DBConnection* dBConnection);
    ~ProcMlpQuery();
    void processAppResponse(CMsg *new_incoming);
    void processErrCode(string, string);
    boolFromString(int & t, const std::string & s, std::ios_base& (*f)(std::ios_base&));

};

#endif
/**
 * \file ProcMlpQuery.cpp
 * \author Pep Soler
 */

#include "Hexa.h"
#include "ProcMlpQuery.h"
#include "SessionManager.h"
#include "SessionDataMLP.h"
#include "MLFServices.h"
#include "DBConnection.h"
#include "DBResult.h"
#include "DBException.h"
#include "MLFServices.h"
#include "Log.h"
#include "InMsg.h"
#include "MlpQueryMsg.h"
#include "MyAppConstants.h"
#include "StringTool.h"
#include "NetResult.h"

using namespace std;

/**
 * Brief description of the constructor
 * @param sessionManager is the instance to the Session Manager
 * @param dBConnection is the instance to a dBConnection
 * These two instance are created in MlpModule
 */
ProcMlpQuery::ProcMlpQuery( SessionManager* sessionManager,
DBConnection* dBConnection)
{
    m_dBConnection = dBConnection;
    m_sessionManager = sessionManager;
}

/**
 * The destructor delete the instances created
 */
ProcMlpQuery::~ProcMlpQuery()
{
    delete m_sessionManager;
    delete m_sessionDataMLP;
    delete m_dBConnection;
    delete m_dBResult;
}

/**
 * Turn a string-number into a integer.
 * @param t is the final integer
 * @param s is the string where the string-number is
 * @param f is base of the string-number, eg: hex if the string-number
 * is a hexadecimal number
 */
bool
ProcMlpQuery::FromString(int& t,
const std::string& s,
std::ios_base& (*f)(std::ios_base&))
```cpp
std::istringstream iss(s);
return !(iss >> f >> t).fail();
}

/**
* Fill the resultTag with the error code and the slogan. The
resultTag is located in ELayer
*/
void ProcMlpQuery::processErrCode(string code, string slogan)
{
    LOG_DEBUG(ExInfo("MAKING THE ERROR MESSAGE"));
    m_sessionDataMLP->GetMLPServices()->SetResultTag(code, slogan);
    m_sessionDataMLP->SignalComplete();
    LOG_DEBUG(ExInfo("MAKING THE ERROR MSG HAS FINISHED"));
}

/**
* Processes the messages from LBS if the messages are
LBS_QUERY_MSG_RSP
* @param the new_incoming is income message
*/

void ProcMlpQuery::processAppResponse(CMsg *new_incoming)
{
    LOG_DEBUG(ExInfo("ENTERING PROCAPPRESP"));
    InMsg inMsg(new_incoming);
    m_mlpQueryMsg = new MlpQueryMsg(new_incoming);
    /** The MSISDN is catched from the income message to get the
session Data of the user*/
    unsigned int id = m_mlpQueryMsg->getMlpSessionId();
    m_sessionDataMLP=m_sessionManager->getKeyData(id);
    if (NetResult::INDICATE_SUCCESS ==
        inMsg.readByte(PRM_ERR_INDICATOR)){
        LOG_DEBUG(ExInfo("SUCCEFUL RESPONSE"));
        /** The CGI is catched from the income message*/
        string cgi;
        if (inMsg.contains(PRM_LOCATION))
            cgi = Hexa::hexaData(inMsg.readData(PRM_LOCATION));
        /** the digits from the position 6 to 10 are caught. That is
the LAC.*/
        cgi.copy(buf,4,6);
        buf[5]='\0';
        m_sessionDataMLP->SetLac(string(buf));
        /** the digits from the position 10 to 14 are caught. That is
the CI.*/
        cgi.copy(buf,4,10);
        buf[5]='\0';
        m_sessionDataMLP->SetCi(string(buf));
        try{
            LOG_DEBUG(ExInfo("QUERYING DB"));
            query="SELECT longitude,latitude FROM cells WHERE
LAC=(select cast(0x";
            query.append(m_sessionDataMLP->GetLac());
            query.append(" as unsigned)) AND CellID=(select cast(0x";
            query.append(m_sessionDataMLP->GetCi());
            query.append(" as unsigned));
```
```
std::auto_ptr<DBResult> dBResult(m_dBConnection->query(query));

LOG_DEBUG(ExInfo("QUERY").addInfo("query",query));
if (dBResult->nextRow()) {
    /** The coordenates are saved in the Session Data of the user, and then a Signal is sent to HTTP server and notifying that it can carry on with the creation of the MLP message*/
    m_sessionDataMLP->GetMLPServices()->SetCoord(dBResult->getFieldAsString(0),dBResult->getFieldAsString(1));
    m_sessionDataMLP->GetMLPServices()->SetResultTag("0","OK");
    m_sessionDataMLP->SignalComplete();
}

} catch (DBException &e) {
    LOG_ERROR(e.info());
}
else if (NetResult::INDICATE_OPEN_REFUSED == inMsg.readByte(PRM_ERR_INDICATOR)) {
    LOG_DEBUG(ExInfo("THE MSG WAS AN ERROR"));
    switch (inMsg.readByte(PRM_ERR_CODE)) {
    case NetResult::OPEN_REFUSED_noReasonGiven:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
    case NetResult::OPEN_REFUSED_invalidDestinationReference:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
    case NetResult::OPEN_REFUSED_invalidOriginationReference:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
    case NetResult::OPEN_REFUSED_applicationContextNotSupported:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
    case NetResult::OPEN_REFUSED_potentialVersionIncompatibility:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
    case NetResult::OPEN_REFUSED_remoteNodeNotReachable:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
    default:
        processErrCode("2", "UNSPECIFIED ERROR");
        break;
    }
} else if (NetResult::INDICATE_USER_ABORT == inMsg.readByte(PRM_ERR_INDICATOR)) {
    LOG_DEBUG(ExInfo("THE MSG WAS AN ERROR"));
    switch (inMsg.readByte(PRM_ERR_CODE)) {
    case NetResult::ABORT_USER_REASON_userSpecificReason:
processErrCode("101", "CONGESTION IN LOCATION SERVER");
    break;
  case NetResult::ABORT_USER_REASON_userResourceLimitation:
    processErrCode("101", "CONGESTION IN LOCATION SERVER");
    break;
  case NetResult::ABORT_USER_REASON_resourceUnavailable:
    processErrCode("101", "CONGESTION IN LOCATION SERVER");
    break;
  case NetResult::ABORT_USER_REASON_applicationProcedureCancelled:
    processErrCode("101", "CONGESTION IN LOCATION SERVER");
    break;
  case NetResult::ABORT_USER_REASON_procedureError:
    processErrCode("101", "CONGESTION IN LOCATION SERVER");
    break;
  default:
    processErrCode("2", "UNSPECIFIED ERROR");
    break;
  }
}
else if (NetResult::INDICATE_PROVIDER_ABORT == inMsg.readByte(PRM_ERR_INDICATOR)) {
    LOG_DEBUG(ExInfo("THE MSG WAS AN ERROR"));
    switch (inMsg.readByte(PRM_ERR_CODE)) {
      case NetResult::ABORT_PROVIDER_REASON_providerMalfunction:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
      case NetResult::ABORT_PROVIDER_REASON_supportingDialogueReleased:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
      case NetResult::ABORT_PROVIDER_REASON_resourceLimitation:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
      case NetResult::ABORT_PROVIDER_REASON_maintenanceActivity:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
      case NetResult::ABORT_PROVIDER_REASON_versionIncompatibility:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
      case NetResult::ABORT_PROVIDER_REASON_abnormalMapDialogue:
        processErrCode("101", "CONGESTION IN LOCATION SERVER");
        break;
      default:
        processErrCode("2", "UNSPECIFIED ERROR");
        break;
    }
}
else if (NetResult::INDICATE_MAP_NOTICE == inMsg.readByte(PRM_ERR_INDICATOR)) {
    LOG_DEBUG(ExInfo("THE MSG WAS AN ERROR"));
    switch (inMsg.readByte(PRM_ERR_CODE)) {

case NetResult::NOTICE_PROBLEM_abnormalEventDetectedByPeer:
    processErrCode("101", "CONGESTION IN LOCATION SERVER");
    break;

case NetResult::NOTICE_PROBLEM_responseRejectedByPeer:
    processErrCode("101", "CONGESTION IN LOCATION SERVER");
    break;

case NetResult::NOTICE_PROBLEM_abnormalEventReceivedFromPeer:
    processErrCode("101", "CONGESTION IN LOCATION SERVER");
    break;

case NetResult::NOTICE_PROBLEM_messageCannotBeDeliveredToThePeer:
    processErrCode("101", "CONGESTION IN LOCATION SERVER");
    break;

default:
    processErrCode("2", "UNSPECIFIED ERROR");
    break;
}
}

else if (NetResult::INDICATE_CNF_USER_ERROR ==
inMsg.readByte(PRM_ERR_INDICATOR)){
    LOG_DEBUG(ExInfo("THE MSG WAS AN ERROR"));
    switch (inMsg.readByte(PRM_ERR_CODE)) {
    case NetResult::USER_ERROR_unknownSubscriber:
        processErrCode("4", "UNKNOWN SUBSCRIBER");
        break;
    case NetResult::USER_ERROR_numberChanged:
        processErrCode("4", "UNKNOWN SUBSCRIBER");
        break;
    case NetResult::USER_ERROR_callBarred:
        processErrCode("4", "UNKNOWN SUBSCRIBER");
        break;
    case NetResult::USER_ERROR_forwardingViolation:
        processErrCode("4", "UNKNOWN SUBSCRIBER");
        break;
    case NetResult::USER_ERROR_cugReject:
        processErrCode("4", "UNKNOWN SUBSCRIBER");
        break;
    case NetResult::USER_ERROR_absentSubscriber:
        processErrCode("5", "ABSENT SUBSCRIBER");
        break;
    case NetResult::USER_ERROR_bearerServiceNotProvisioned:
        processErrCode("4", "UNKNOWN SUBSCRIBER");
        break;
    case NetResult::USER_ERROR_teleserviceNotProvisioned:
        processErrCode("4", "UNKNOWN SUBSCRIBER");
        break;
    case NetResult::USER_ERROR_facilityNotSupported:
        processErrCode("4", "UNKNOWN SUBSCRIBER");
        break;
    case NetResult::USER_ERROR_systemFailure:
        processErrCode("1", "SYSTEM FAILURE");
        break;
    case NetResult::USER_ERROR_dataMissing:
        processErrCode("4", "UNKNOWN SUBSCRIBER");
        break;
    case NetResult::USER_ERROR_unexpectedDataValue:
        processErrCode("4", "UNKNOWN SUBSCRIBER");
        break;
    default:
        }
processErrCode("2", "UNSPECIFIED ERROR");
break;
}
}
}

else if (NetResult::INDICATE_CNF_PROVIDER_ERROR == 
inMsg.readByte(PRM_ERR_INDICATOR)) {
    LOG_DEBUG(ExInfo("THE MSG WAS AN ERROR"));
    switch (inMsg.readByte(PRM_ERR_CODE)) {
        case NetResult::PROVIDER_ERROR_duplicitedInvoke:
            processErrCode("203", "CONGESTION IN MOBILE NETWORK");
            break;
        case NetResult::PROVIDER_ERROR_notSupportedService:
            processErrCode("203", "CONGESTION IN MOBILE NETWORK");
            break;
        case NetResult::PROVIDER_ERROR_mistypedParameter:
            processErrCode("203", "CONGESTION IN MOBILE NETWORK");
            break;
        case NetResult::PROVIDER_ERROR_resourceLimitation:
            processErrCode("203", "CONGESTION IN MOBILE NETWORK");
            break;
        case NetResult::PROVIDER_ERROR_initiatingRelease:
            processErrCode("203", "CONGESTION IN MOBILE NETWORK");
            break;
        case NetResult::PROVIDER_ERROR_unexectedResponseFromPeer:
            processErrCode("203", "CONGESTION IN MOBILE NETWORK");
            break;
        case NetResult::PROVIDER_ERROR_serviceCompletionFailure:
            processErrCode("203", "CONGESTION IN MOBILE NETWORK");
            break;
        case NetResult::PROVIDER_ERROR_noResponseFromPeer:
            processErrCode("203", "CONGESTION IN MOBILE NETWORK");
            break;
        case NetResult::PROVIDER_ERROR_invalidResponseReceived:
            processErrCode("203", "CONGESTION IN MOBILE NETWORK");
            break;
        default:
            processErrCode("2", "UNSPECIFIED ERROR");
            break;
    }
}
else{
    /* This means that there is error*/
    LOG_DEBUG(ExInfo("THERE IS AN UNSPECIFIC ERROR"));
    processErrCode("2", "UNSPECIFIED ERROR");
}

LOG_DEBUG(ExInfo("PROCESS RESPONSE HAS FINISHED"));