Bachelor's Thesis

Degree Programme in Information Technology

2010

Antti Siirilä

Specification of a Remote Monitoring Protocol

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR´S THESIS | ABSTRACT

UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Information Technology

21 May 2010| Total number of pages: 95

Instructors

Patric Granholm, TUAS

Hannu Rytilä, Kone Oy

Author

Antti Siirilä

Specification of a Remote Monitoring Protocol

The purpose of this bachelor's thesis was to specify a generic data communication protocol for remotely performed condition monitoring purposes in service business. The remote monitoring system includes a vast number of reporting client devices and a single centralized or several decentralized monitoring centre(s). The reporting client devices are capable of independently determining their condition using a self-diagnostics system and reporting the results to the monitoring centre. Information transfer within the system is done over a private IP network that combines wireless mobile telecommunication and mobile operator backbone networks.

The thesis is divided in three logical sections. The first section describes the remote monitoring system and specifies the requirements for the Remote Monitoring Protocol (RMP). The second section discusses the system specific network technologies and introduces the protocol engineering method relevant for RMP. The third section includes the specification and the demonstration of RMP.

The functional requirements for RMP were specified as follows: Mutual authentication between the client and the RMC gateway, the client must perform registration to a pre-defined remote monitoring centre, the client devices shall be able to download configuration parameters from the remote monitoring centre, the client shall immediately report the faults and warning and status changes once occurred, the client must send a Bill-of-Material (BOM) message to Remote Monitoring Server (RMS) when requested, and the client shall disable itself if the authorized RMC is not found within a configurable time.

RMP was specified to follow the layered protocol model located on top of the UDP/IP stack and providing an Application Programming Interface (API) for the client and server applications. RMP can be driven in reliable and unreliable packet exchange modes. In addition, RMP specification includes the message structures and the exchange procedures for communication between the analyzer application and the remote monitoring server. Furthermore, RMP supports message sizes up to 64 KB and offers fragmentation and reassembling services for the messages passing the configured path Maximum Transmission Unit (MTU) size. RMP features also include the automatic discovery of the path MTU size.

As conclusion, the functionality and features of the RMP specification were demonstrated in a reference implementation. The implementation was carried out in JAVA using the Eclipse development environment.

KEY WORDS:

Information Technology, Telecommunication Protocol, Protocol Design, Remote Monitoring, Maintenance, Service Business

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Degree Programme in Information Technology

21.5.2010| Sivumäärä: 95

Ohjaajat

Patric Granholm, Turku AMK

Hannu Rytilä, Kone Oy

Tekijä

Antti Siirilä

Kaukovalvontaprotokollan määritys

Tämän opinnäytetyön tavoitteena oli määrittää avoin tietoliikenneprotokolla käytettäväksi huoltoliiketoiminnassa ja tarkemmin sen asiakaslaitteiden etänä suoritettavassa kunnonvalvonnassa. Kunnossapidon kaukovalvontajärjestelmä sisältää vaihtelevan määrän raportoivia asiakaslaitteita ja keskitetyn palvelinyksikön tai useita hajautettuja palvelinyksiköitä globaalia tiedon kasaamista varten. Valvottavat laitteet ovat kykeneviä omatoimiseen kunnon valvontaan ja analysointiin ja tarvittaessa raportoivat tilamuutoksista palvelinyksikölle. Tiedonsiirto laitteistojen välillä tapahtuu teleoperaattorin tarjoaman suojatun IP-verkon yli, jossa yhdistyy mobiili telekommunikaatiotekniikka sekä operaattorin ydintietoliikenneverkko.

Opinnäytetyö jakautuu kolmeen osaan. Ensimmäinen osio kuvaa kaukovalvontajärjestelmää ja määrittää vaatimukset toteutettavalle kaukovalvontaprotokollalle. Toinen osio tarjoaa teoreettista taustatietoa liittyen järjestelmän sisältämään verkkoteknologiaan sekä käsittelee protokollan suunnittelua koskevia aiheita. Kolmas osio sisältää protokollan dokumentoinnin ja demonstraation.

Tärkeimmät toiminnalliset vaatimukset kaukovalvontaprotokollalle voidaan määrittää seuraavasti: välitön raportointi vikatilanteissa, automatisoitu laitekonfiguraatio, laitekohtaisten tietojen lähetys valvontakeskukseen pyydettäessä, laitteistoa koskevien tilastotietojen säännöllinen raportointi valvontakeskukseen, kaksisuuntainen todentaminen asiakaslaitteen ja valvontakeskuksen välillä, sekä asiakaslaitteen automaattinen inaktivoituminen tilanteessa, jossa asiakaslaite ei onnistu todentamaan valvontakeskusta tai toisin päin.

Määritetty kaukovalvontaprotokolla suunniteltiin modulaariseksi siten, että se sijoittuu UDP/IP protokollien päälle ja tarjoaa käyttöliittymän yläpuolellaan olevalle laitteistokohtaiselle sovellukselle. Käyttöliittymä sekä protokollan sisältämät viestirakenteet mahdollistavat protokollan geneerisen käytön eri laitetyyppien välillä. Protokolla voi toimia kahdessa eri moodissa, joista toinen tukee luotettavaa ja toinen epäluotettavaa tiedonsiirtoa. Lisäksi protokolla määrittää vaatimusten mukaiset viestirakenteet käytettäväksi kaukovalvontajärjestelmän tiedonsiirtoon sallien viestikoot aina 64 kt saakka.

Lopuksi protokollan toimivuutta demonstroitiin JAVA ohjelmointikielellä toteutetun referenssimallin avulla, jossa se niiltä osin todettiin vaatimusten mukaiseksi. Lisäksi loppupohdiskelussa tarkennettiin huomioon otettavia asioita, jos protokolla implementoidaan järjestelmään, jossa tietoliikenne yhteydet toteutetaan suojaamattomanverkon (esim. Internet) yli.

ASIASANAT:

Tietoliikenne, Telekommunikaatio protokolla, Kaukovalvonta, Kunnossapito

CONTENTS

FIGURES

TABLES

ABBREVIATIONS

3G	Third-Generation Cell-Phone Technology
AAA	Authentication Authorization Accounting
ACK	Acknowledgement Packet
AF	Action Flag
API	Application Programming Interface
APN	Access Point Network
AS	Authentication Server
BOM	Bill-of-Material
BSS	Base Station Subsystem
CE	Customer Edge
CHAP	Challenge Handshake Authentication Protocol

DEP	Data Endpoint
DIP	Data Integration Point
EAP	Extensible Authentication Protocol
EFFIMA	Energy and Life-cycle Cost-efficient Machines
FIMECC	Finnish Metals and Engineering Competence Cluster
FEC	Forwarding Equivalent Class
GGNS	Gateway GPRS Support Node
GPRS	General Packet Radio Service
IP	Internet Protocol
LAN	Local Area Network
LIB	Label Information Base
LSR	Label Switched Routers
M2M	Machine-to-Machine
MD5	Message Digest 5 Algorithm
MPLS	Multi-protocol Label Switching
MS	Mobile Station
MTU	Maximum Transmission Unit
NACK	Negative Acknowledgement Packet
NAPT	Network Address Port Translation
NAS	Network Access Server
NAT	Network Address Translation
PDU	Protocol Data Unit
PE	Provider Edge
PMTU	Path MTU
PT	PDU Type

QoS	Quality of Service
RACK	ACK Required
RADIUS	Remote Authentication Dial In User Service
RMC	Remote Monitoring Centre
RMG	Remote Monitoring Gateway
RMP	Remote Monitoring Protocol
RMS	Remote Monitoring Server
SAG	Site Access Gateway
TDM	Time Division Multiplexing
TLV	Type-Length-Value
UDP	User Datagram Protocol
UDPCP	UDP-based Communication Protocol
VPN	Virtual Private Network

Introduction

1.1 Background

The subject of the study, Generic Remote Monitoring Protocol, is a part of the Global Interconnectivity Architecture of Fimecc InterSync project. The project is further part of WP2 of the EFFIMA program, which belongs to the FIMECC Intelligent Solutions thematic area of FIMECC. The Generic Remote Monitoring Protocol can be used for condition monitoring and maintenance purposes of customer devices in service business.

Finnish Metals and Engineering Competence Cluster (FIMECC Ltd.) works in conjunction with metals and engineering industry companies, universities, and research institutions to elevate the research co-operation between the participants [1]. The target is to produce revolutionary Finnish engineering through various top quality research programs utilizing international research networks, and application-driven research contents. FIMECC has defined five strategic research themes to address the research problems and questions to be dealt in different research programs.

The Intelligent Solution theme is one of the five strategic themes of FIMECC. The mission of the theme is to improve the global competitiveness of Finnish metal and machinery industry by producing novel strategic knowledge for intelligent solutions. Furthermore, this will lead to the development of sustainable and productive user processes and operations. The sustainability in Intelligent Solutions theme refers to efficiency in raw materials and energy utilization as well as minimized environmental footprint. The productivity is measured in performance, availability, stability, and flexibility of products and processes. Safety is one of the key factors of the productivity and affects all levels. Additionally, expenses are taken into consideration in all phases of the mission, so that the lifecycle cost of products and processes becomes

compatible. To achieve the objectives, the Intelligent Solution theme functions in close conjunction with the other FIMECC themes utilizing both the international and national research networks. [2]

The Energy and Life-cycle Cost-efficient Machines (EFFIMA) program is based on the mission statements of the FIMECC Intelligent Solutions theme. The goal of EFFIMA is to develop energy-efficient machines, whose life span cost is drastically lower than today's leading technologies allow. To make this possible, the EFFIMA program includes research projects of novel technologies and solutions in the area of energy efficiency. The program is structured in three parts, each of which has dedicated objects. The following describes these divisions: "WP1 Low energy consumption and environmental Emissions, WP2 Technologies for Life-cycle Cost Management, and WP3 Efficiency by means of human compatible multi-machine systems" [3].

The WP2 work package of EFFIMA sharpens on systems possessing self-monitoring, diagnosing and repairing capabilities that allows them to be maintained with minimal lifespan cost. Life-cycle management offers a globally relevant, durable, and economically interesting brand for enterprises in Finnish machine industry. [3]

The WP2: InterSync project bases on the analysis of industrial requirements within the EFFIMA program. Its aim is to improve global networking of production systems and machinery by enhancing their energy efficiency, capacity, and material usage. To achieve these objectives, The InterSync project will study the opportunities and implement novel solutions for wireless and wired communication and sensor networks. Industrial service business and applications will benefit from the new solutions that allow collecting real-time multi-sensory information through decentralized wireless sensor networks. In addition, the systems will be based on modular architecture, which provides backward and forward compatibility for all system components from different life cycle phases. The compatibility includes optimal configurations, which can be performed without demanding and costly installation tasks. Finally, the InterSync project answers to the research and development question of

"maintenance-free and cost-effectively configurable wireless or wired multi-sensor platforms with reliable real-time data transfer capability in industrial environment", which has been identified in the WP3 of EFFIMA program as "Efficiency by means of automation" [3].

1.2 Research Problem and Focus

This thesis proposes a solution for a generic remote monitoring protocol of Machine-to-Machine system that combines globally interconnected condition-aware mechatronics devices and centralized or distributed monitoring centre(s). The condition awareness refers to a device that is able to perform self-diagnostics and inform the monitoring centre in case of a weakened condition. Because the raw sensor data is not transferred over the network, the data transfer between the end systems can be messages based without real-time requirements. The interconnectivity network provides wireless and wired end-to-end IP connectivity through a network platform obtained from mobile telecommunication operators. The objective of the thesis is to define the requirements and produce a specification for a remote monitoring protocol. Furthermore, the protocol is demonstrated with a simple reference implementation, which is realized in a JAVA programming environment.

1.3 Chapter Organization

The organization of this thesis continues with the discussion of the usage of remote monitoring in service business. Additionally, Chapter 2 introduces the remote monitoring system for which RMP shall be designed, and finally defines the requirements for the protocol.

Chapter 3 discusses the network infrastructure and technologies of the lower layers. Both versions of IP are discussed according to their relevance for RMP. Additionally, the chapter describes how IP is implemented into mobile telecommunication technology. Finally, the chapter discusses the effects of these technologies on RMP.

Chapter 4 gives a background for protocol engineering and introduces some existing protocols. The basic sequence control, fragmentation, encoding, and authentication mechanisms are discussed. The features of connectionless transport layer protocol User Datagram Protocol (UDP) are introduced. UDP-based Communication Protocol (UDPCP) is introduced as a protocol example for RMP. As a result, the chapter closes with a discussion of the RMP design.

Chapter 5 specifies the remote monitoring protocol. The results of the JAVA demonstration are presented in Chapter 6.

Chapter 7 concludes the thesis and determines whether the proposed solution fulfilled the specified requirements for RMP or not. In addition, the chapter discusses some future improvement concerning the RMP.

2 Remote Monitoring in Service Business

The traditional use for remote monitoring in service business has mainly been fault reporting. This is, when the remote equipment breaks, the equipment detects the faulty state and sends a notification to the Remote Monitoring Centre (RMC) of the service company. The fault notification usually includes a fault code and the identification information of the equipment. Based on the fault report, the personnel of the RMC is able to launch the necessary actions to restore the equipment back to the normal state. This usually means that a trouble-shooter is sent to visit the equipment. The trouble-shooter may use the fault report data to localize the cause for the brake down. The previously described scenario refers to a maintenance model called reactive model. The reactive maintenance model suits for pieces of equipment, whose function is not susceptible to unavailability caused by breakdowns. [4]

Better availability is achieved if the equipment is maintained regularly at pre-defined intervals. The purpose of the regular service visits is to maintain the condition of the equipment at the level that the possible break-downs can be prevented. This kind of approach is refers to as the preventive maintenance model. Different factors are used to determine the need of maintenance

operations for an individual device or a group of devices. First, the economic factors can be used to define the amount of preventive maintenance operations. Shortly, the cost of the preventive maintenance operations should be less than the estimated expenditures caused by the possible break-down of the equipment without being maintained. The frequency of the preventive tasks can also be determined from the statistics. This method relies on the information given by number of the faults and the density of their occurrence for a group of devices. The remote monitoring has an important role in collection of these statistical data. The fault reports can be stored into a database and analyzed with suitable tools. The equipment may also send periodical reports including statistics on its own functions. These data can also be used when the preventive maintenance model is being constructed. Despite the methods described in the previous paragraph, it is difficult to optimize the preventive model so that neither overhead nor shortage in maintenance operations occurs. This is because the preventive model is based on estimations and not on measured data. [4]

Proactive and condition-based maintenance models take advantage of the remote monitoring system where the condition of the equipment is constantly measured and the data are sent to RMC. In condition-based model, various different techniques can be used to measure the condition of a component. Some examples of these techniques are vibration measurement, testing of lubricant, thermography, and noise detections. The measurements in each technique are carried out with suitable sensor devices. The results of the measurements are compared with reference information, which indicates the normal behaviour of the component. A possible fault is detected if the measured data bypasses certain pre-defined boundaries. The previous analysis can be performed locally so that only the result of the analysis is sent to RMC or controversially the raw sensor data can be sent to RMC for further analysis. Transportation of such bulk data in real-time over a packet switched network requires the use of data streaming techniques. When local analysis is used, the amount of transferred data is reduced and the transfer can usually be carried as message-based. This is convenient in mobile telecommunication networks,

where the data transfer cost is generally based on the amount of data transferred and in some cases, bandwidth availability issues may occur. [4], [5]

2.1 Remote Monitoring System Description

The key components of remote monitoring system on which this thesis work focuses include the client sites, the data transportation network, the default monitoring centre, and the optional local RMCs. (Figure 1). The purpose of the system is to locally detect the condition of the devices and report the result of the conditions analysis to the monitoring centre. The data transfer between the end systems include authentication and registration traffic, fault, status, and statistics reporting, configuration downloading, and sending a BOM message.

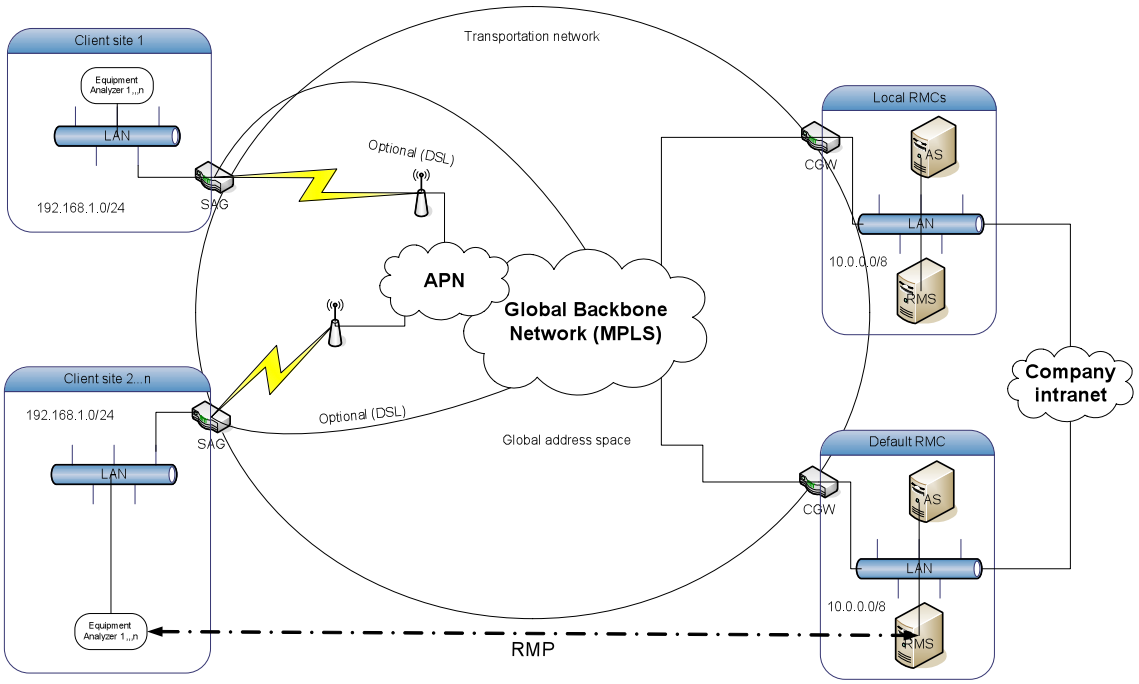

Figure 1. Network architecture and the key components of the remote monitoring system.

The client site contains the Site Access Gateway (SAG), the Local Area Network (LAN) in which the client equipment resides. The client site LAN can implement several client equipments. Furthermore, the remote monitoring system may include many client sites, which can share the same private address space. SAG performs routing, Network Address Translation (NAT), and

packet filtering for data transfer between the site network and the transportation network. SAG has a wireless interface through which it connects to the mobile telecommunication network that is part of the bigger transportation network provided by a mobile telephone operator. Optionally, SAG may connect directly to the global backbone network through a digital subscriber line. The client equipment is an intelligent embedded device including an analyzer component that is capable of determining the real-time condition of the equipment by analyzing the data provided by the embedded operating system and various different sensors of the equipment. The in-built database of the equipment stores the results of the condition analysis. Each new occurrence of an event updates the database and triggers the sending of an information message to RMS. The analyzer can be configured to determine which type of events shall be sent to RMC.

The transportation network is a private Machine-to-Machine (M2M) network combining a mobile telecommunication network, and a wired global backbone network. The transportation network uses Internet Protocol (IP) as a network layer protocol throughout the whole network. Privacy over the transportation network is based on two different technologies. The global backbone network is a virtual private network based on Multi-protocol Label Switching (MPLS) technology. The Access Point Network (APN) is a private network over the mobile wireless network based on SIM and global SIM solutions provided by the mobile operator. Due to the global private network, the remote monitoring messages can be sent without encryption. However, the system shall implement an entity authentication schema. The system has three visible IP-address spaces. The site network, the transportation network, and the remote monitoring centre network. The Network Address Translation is dynamically performed on SAG.

The remote monitoring centre contains a LAN including the following key components: Remote Monitoring Gateway (RMG), Authentication Server (AS), and RMS. RMG is a router performing NAT and routing tasks between the RMC network and the transportation network. In addition, RMG acts as an

authenticator between the AS and the client requesting network access. AS is responsible for authenticating the clients before they access the RMC network. RMS handles the registration of client devices and stores the data received from the equipment analyzers. RMS has a database where all the data and the identification information of the client devices are stored. The remote monitoring system may consist of many RMCs. However, one of the RMCs functions as the default RMC, from where the new clients can request the system registration using their default configuration. The default RMC has a possibility to dynamically re-direct a client to another RMC. This may occur because of the device configuration or a load balancing need between the RMCs.

2.2 Requirements for the Remote Monitoring Protocol

Based on the previous description about the remote monitoring system, the following functional requirements for the remote monitoring protocol can be defined:

- Mutual authentication between the client and the RMC gateway
- The client must perform registration to the pre-defined remote monitoring centre
- Client devices shall be able to download configuration parameters from the remote monitoring centre
- The client shall immediately report the faults, warnings, and status changes once occurred
    - The fault reporting shall include the equipment status information
- The client must send a BOM message to the RMS when requested
- The client shall disable itself if the authorized RMC is not found within a configurable time

Additionally, the following non-functional requirements are specified:

- The protocol shall be compatible with both IP version 4 and IP version 6

- The client device shall be able to send fault, status, and usage information messages to the monitoring centre
- The maximum message size shall conform the maximum UDP-packet size (65,535 bytes)
- The communication shall be message-based, therefore, no streaming is required
- No message encryption is required
- Both the client and the monitoring centre shall be authenticated before any data transfer between the peers
- Protocol shall work over several IP address spaces with dynamic NAT
    - Client originated transactions
- The protocol shall be able to operate without any manual configuration in the site.
- The protocol shall implement message fragmentation for the messages exceeding the network MTU
- The protocol shall be connectionless
- The protocol shall be light

3 Network Technologies

The underlying network topology and the different technologies in use have an impact on the design of RMP that shall be discussed in this chapter. This impact can be concretised into the following questions:

- How can RMP handle routing issues caused by different IP address spaces and NAT?
- Can IP fragmentation be trusted as sole fragmentation service in the system?
- How does the presence of mobile telecommunication networks in the system affect RMP?

In addition, the chapter shortly introduces some basic features of IPv4 and IPv6 and discusses their differences. Moreover, the machine-to-machine concept shall be explained as well as the main components of mobile telecommunication networks. Finally, some conclusions concerning the RMP design are made in the last paragraph of this chapter.

3.1 Internet Protocol

IP is used as a network layer protocol for RMP. Some key features of IP (versions 4 and 6) include packet routing between network nodes, packet fragmentation, Quality of Service (QoS), and connectionless packet delivery [6]. The following sections will give a short description of these features.

3.1.1 IP packet

IP transfers data between hosts in units called IP packets. The IP packet consists of IP header and the payload from the upper layer. The header contains the information needed for routing and other services (source and destination IP-addresses, priority, etc.) and has a fixed length of 20 bytes in IPv4 and 40 bytes in IPv6 (Figure 2) [6], [7]. Additionally, the IPv4 header may include optional fields, which are used to carry control, measurement, or debugging information. The length of the optional fields may vary and increase the size of the header and thus the size of the IP packet accordingly. The IPv4 header length must always terminate at the 32-bit boundary. Therefore, IP uses the padding field to reach the boundary. Whereas IPv4 uses optional fields, IPv6 has optional headers called extension headers for the optional services. As in IPv4, the optional header of IPv6 increases the packet size, or, in other words, decreases the size of the payload a packet is able to carry. The payload refers to the data that the IP carries and it can size up to 64 KB in both versionsof IP. However, to be able to transfer packets as large as that, the IP must fragment the packet into smaller packets. IP fragmentation will be discussed in Section 3.1.5. [8]

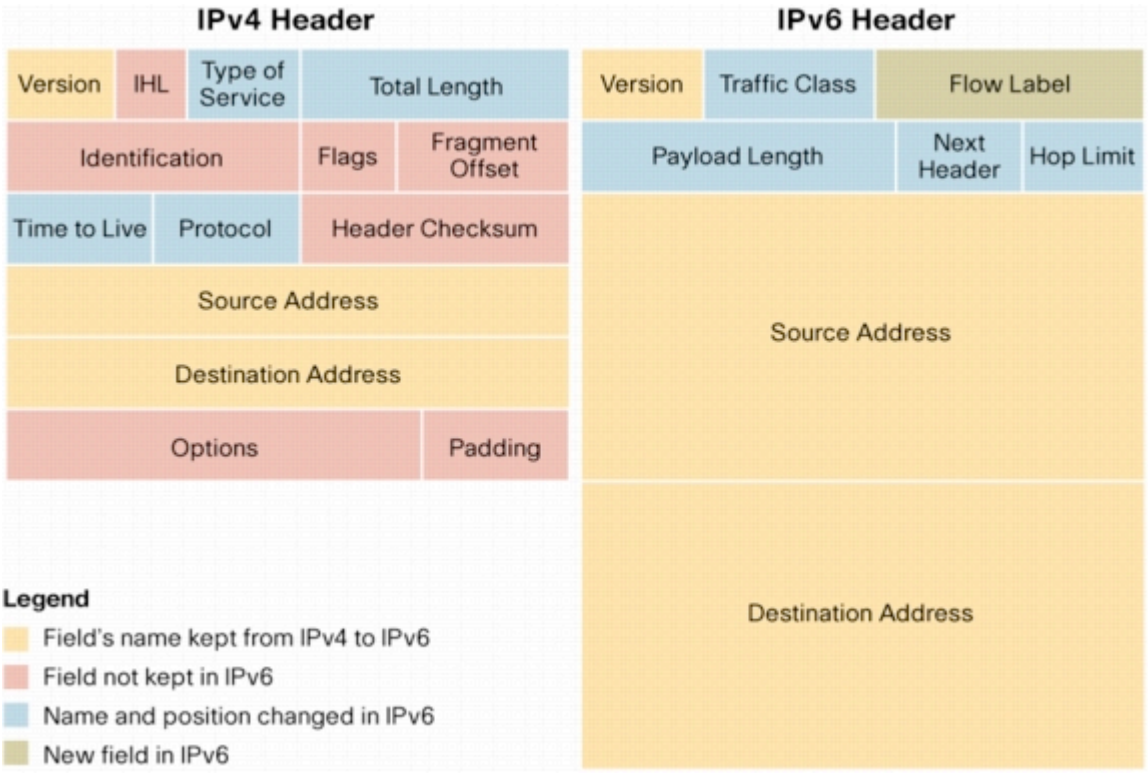

Figure 2. Comparison of IPv4 and IPv6 header format and fields [9]

3.1.2 IP address

Each host in the IPv4 network is identified by a 32-bits long unique identifier called IP address. The IP address is divided in four octets, which are usually represented by decimal numbers separated by dots. Furthermore, the octets of IP address are divided in network portion and host portion depending on which address class is in use. There are five address classes A, B, C, D, and E (Figure 4)[5]. Classes from A to C are used for unicast transmission. Class A addresses allow the largest hosts range for each network whereas class C offers the greatest number of networks allowing 254 hosts in each. Class D is used for multicasting and class E is reserved for future use. In addition, there are some addresses or address ranges that have a special function in the IPv4 address schema. In each address class, there are public addresses and private addresses. Public addresses are only used in the Internet and controlled by the Internet Assigned Numbers Authority. Private addresses are a special portion of addresses in each class from A to C, to be used only in private networks or

LAN. In other words, the nodes directly connecting to the Internet cannot use an address from the private address space. The addresses having all bits set to one in the host portion are used as a broadcast addresses. When an IP packet has a broadcast address as a destination address, the router forwards the packet to all nodes in the network. Additionally, local hosts use the last network address of class A (127.0.0.0). [10]

The designers of IPv4 were not able to predict the huge growth of the Internet, which has led to a situation where the IPv4 address space is not large enough for today's demand. Without some patching arrangements such as Network Address Translation, sub-netting, private address spaces, Classless Inter Domain Routing, etc, the IPv4 addresses would have been depleted a long time ago [11], [12]. Despite these attempts to alleviate the IPv4 address exhaustion, the real solution to the problem relies on IPv6. IPv6 supports an 128-bit address identifier allowing 2^{128} addresses in total, which covers around $4{,}9 \times 10^{28}$ times the world population. Such a vast number of addresses allows globally unique IP addresses for each network device so the private and public address space division of IPv4 becomes obsolete. The IPv6 address is presented in a hexadecimal format divided in 8 words (16-bits) separated by colons (:). The classification of IPv6 addresses differs from the IPv4 classifications. IPv6 addresses are divided in three different types: unicast, multicast, and anycast. RFCrelease number 2373 specifies these types [13]. [10]

The IPv6 specification was first introduced in 1998 by the Internet Society as a next generation IP protocol. It was designed to supersede the existing IPv4. However, the implementation of IPv6 has been slow and still today it has not been able to overcome the IPv4 [10]. Notwithstanding the slowness of the implementation, the IPv6 will eventually displace the IPv4 solving the issues of IP address exhaustion for good.

3.1.3 IP routing

Routers route IP packets between different networks based on the destination IP address found in the packet header. The next hop for the packet is resolved from a database called routing table by matching the destination address against it. The routing table is updated either statically (static routes) or dynamically (routing protocols) depending on the configuration of the router. Static routing requires a great deal of administrative work, because each route to a destination must be manually configured in to the routing table. Therefore, static routes do not suit well for a network where routers may have to resolve routes to hundreds of thousands different destination. However, static routing may be used in a small network where the number of hosts is reasonable. Unnecessary traffic in the network is reduced with the use of static routes instead of routing protocols. Additionally, LAN gateway routers often use the static routes to point the external networks. Dynamic routing, on the other hand, uses routing protocols to dynamically resolve routes to destinations. Routers send messages defined in routing protocol to each other in order to recognize adjacent routers, share routing information, and notify changes in their link states. Based on the messaging data, routers calculate the best path to each destination using a path discovery algorithm. The algorithm used depends on the routing protocol. Only the best path to each destination is added to the routing table. Routers notify each other about the route changes in the network on either regular basis or after each change occurrence depending on the type of the routing protocol in use. [14]

3.1.4 Network Address Translation

As mentioned in Section 3.1.2, hosts in LAN use a private IPv4 address space. As a result, many private LANs can have an identical address schema. This is acceptable if the LANs do not need to communicate with outside networks. However, often this is not the case and some extra means is required to allow a LAN host to access i.e., the Internet. NAT is a technique that allows routers to translate a private IP address to a public IP address. This allows hosts in private

address space to communicate with a host in a public address space. Traditional NAT can be implemented as basic NAT or as Network Address Port Translation (NAPT) depending on the number of global IP addresses in use. [15]

A router configured for basic NAT replaces the private source address of the IP packet to a pre-configured public address. The translation data is saved in a translation table. The router uses the translation table to resolve the destination address to the host if return traffic from the outside (external) host occurs. The translation table stores the address bindings only for a certain period. For example, for UDP, the period should be two minutes [17]. The address binding is removed from the table if the binding is not used during that period. After the binding is removed, the return traffic from outside is dropped by the router. There may also be a pool of public addresses configured on the router. This allows the router to do as many simultaneous translations as there are public addresses in the pool. [15]

Whereas basic NAT does address translations on one-to-one basis, the NAPT does the same on many-to-one basis. NAPT uses transport layer protocol ports, such as Transmission Control Protocol or UDP ports, as a unique identifier for each address translation. NAPT replaces the private source address with the assigned public address in the IP header and the source UDP or Transmission Control Protocol port number in the transport layer header. The address-port binding is saved into the translation table. That is how two IP packets, originated from different LAN hosts, can have the same source address after the NAPT translation. NAPT also functions with Internet Control Message Protocol using the ICMP-query number as a unique identifier for the bindings. [15]

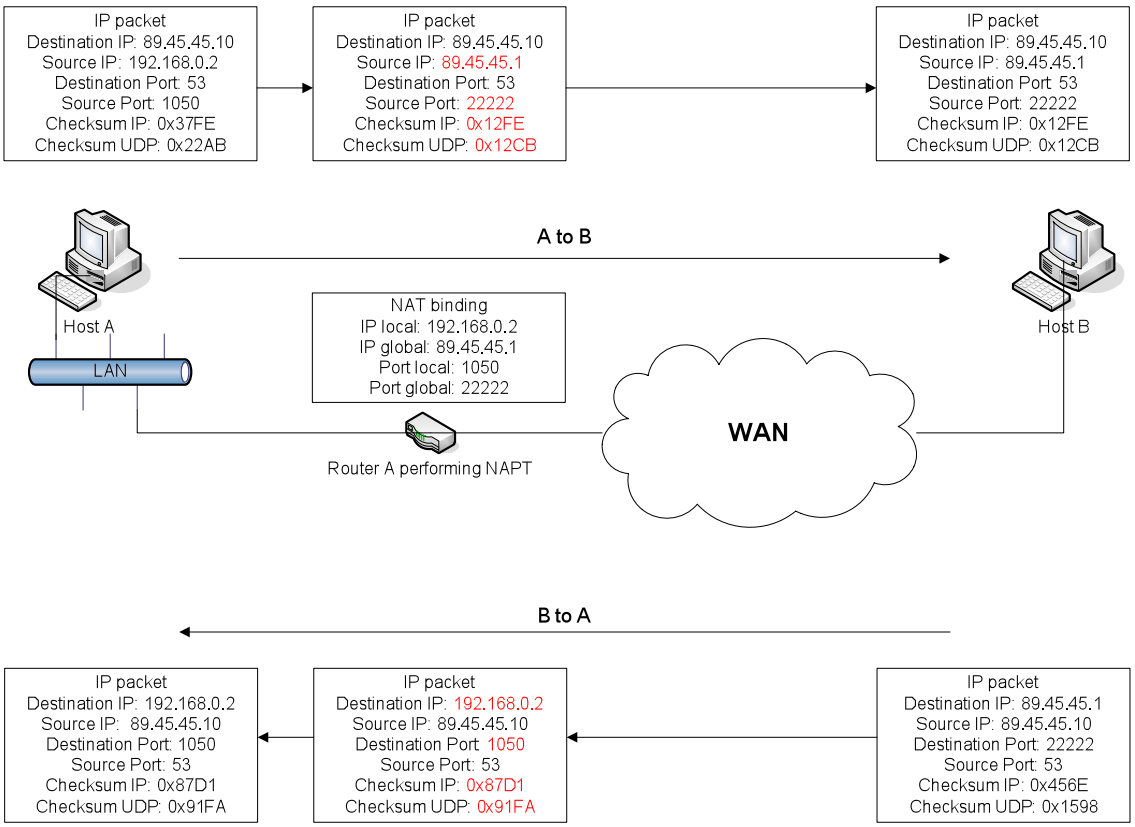

Figure 3. Example of a message exchange between private address space host and public address space host over a NAPT gateway router

As an example (Figure 3), host A with the address 192.168.0.2 is willing to send a packet to host B in the Internet. Router

A receives the packet, searches a route to the destination, and notices that the route points to the Internet. Router A replaces the source address in the IP packet header with the public address 89.45.45.1. Router A also replaces the source port number of transport layer header to a one, which is other than well-known port, and re-calculates the checksum values [10]. Finally, router A stores the address-port binding information into a database and sends the modified packet to host B. When host B sends a reply packet to host A, it uses the public address and port assigned by the router as the destination address and port to host A. Router A receives the packet and looks the route for it. Router A notices that the packet is to be directed into the local network. Before router A can send the packet, it must resolve the address by matching the packet's destination address-port data against the NAPT translation table. Once

the router finds a match, it replaces the corresponding fields in the packet header with the host B's address information. Finally, router A sends the packet to host B. [15]

NAT translation may also be configured statically on a router. Static NAT assigns a permanent address binding between a local private address and a public address. An advantage of this scenario is that the LAN host can be reached from outside at any given time. Therefore, LAN hosts, offering services for the outside world, use static NAT translations. [15]

3.1.5 IP fragmentation

MTU defines the maximum size that a single data unit may have when traversing through a network. MTU size varies depending on the link media and the data-link layer protocol [16]. IPv4 defines a minimum path MTU (PMTU) that each link carrying IP traffic on the path must support, to be 68 bytes. However, MTU is generally much higher value; for example, the MTU of an Ethernet link is 1500 bytes. IETF's RFC 791 defines the IPv4 MTU size as follows, "All hosts must be prepared to accept datagrams of up to 576 octets (whether they arrive whole or in fragments)"[4]. The minimum PMTU for IPv6 is 1280 bytes. The IPv6 specification defines that a link carrying IPv6 traffic must support the minimum of 1280 bytes of MTU. The link must provide a fragmentation/reassemble service for IPv6 packets, if its MTU is less than 1280. On the other hand, if the size of an IP packet is greater than the specified minimum and the link MTU is less than the packet size, the IP must fragment the packet into smaller packets before passing it to lower layer. [6], [7]

MTU size is configured for each network interface and used as a maximum packet size over that link. However, if a network path consists of many different physical media, the MTU size may respectively vary. Therefore, a host sending a packet over the path may not know the correct PMTU. This may result in an inefficient data transmission, if packets are to be fragmented at the sending device. The fragment size should be equal or less than the PMTU. Too small fragments generate more overhead in data transmission, because of the

number of packet headers per transmission increases. Two large fragments, on the other hand, may result in packet loss and delay caused by the re-transmission.

There are two different approaches to overcome this problem. First, the host can perform a procedure called PMTU discovery before the transmission [16], fragment the packet into smaller packets qualifying the PMTU, and send the fragments over the path. The receiving host re-assembles the fragments and reconstructs the packet. Another method is to send the packet respecting the MTU of the local link and fragment/reassemble it, if necessary, at any subsequent interface on the path. IPv4 uses the latter approach and the IPv6 does the path MTU discovery. [6], [7]

Because of the connectionless nature of IP, the IP fragmentation, as the whole IP transmission, in both versions of IP is unreliable. Missing one fragment at the receiver end results in the drop of all the other fragments of the packet. In case of data loss, the upper layer protocols shall handle the retransmission or other correction process. However, the IP is able to re-assemble the fragments received in disorder by using the fragment-offset field. [6], [7]

Despite the previously described methods for IP fragmentation, it may occur that some network links do not support them. This is mostly because IP fragmentation and path MTU discovery have been exploited in network attacks such as Denial of Service attacks. Therefore, some network nodes may not be configured to support the IP fragmentation. As a result, it is strongly recommended to rely on upper layer services when Protocol Data Unit (PDU) needs to be fragmented [17].

3.2 Mobile M2M

The global accessibility and mobility is the base for M2M concept. The core components of M2M include: Data Endpoint (DEP), communication network, and Data Integration Point (DIP). M2M implementation generally includes several different DEPs and a single DIP. [18]

DEP functions as an interface between a larger sub-system and the communication network. It is also responsible for delivering data between the Data Integration Point and the subsystem. The condition analyzer of our remote monitoring can be placed as DIP in the M2M concept. [18]

The communication network may be any kind of network capable of carrying data between the two end-points DEP and DIP. Typically, communication network uses the Internet as a core network combined with some private or public wireless networks to extend the geometrical coverage. When such public connections are used, it is crucial that the M2M application has applied the adequate security measures for data transmissions. Security may be provided by encrypting the application messages or using Virtual Private Network (VPN) tunnelling over the unsecured network connections. In contrast, mobile operators have launched a business concept offering connectivity platforms for M2M applications. Because of the modern mobile technologies and the vast geometrical coverage of mobile networks, the operators are able to offer end-to-end secure IP connectivity for M2M implementations. The security in the network is achieved by using MPLS-based VPN and private mobile networks based on SIM and global SIM technologies. Such a network appears as a completely closed network for the endpoints and is, therefore, an ideal infrastructure for M2M applications requiring global IP connectivity. [18]

Data Integration Point refers to the point where data provided by DEPs is integrated into an IT application. In the remote monitoring system, the remote monitoring server would assume the functions of the DIP. [18]

3.2.1 IP in mobile telecommunication networks

Because RMP is to be used over mobile wireless packet-switch networks such as General Packet Radio Service (GPRS), Enhanced Data GSM Environment, and Third-Generation Cell-Phone Technology (3G), this section will discuss how IP traffic is carried in those networks. However, this section will not give a penetrating analysis of the topic but will concentrate on the matters that should be taken into account in the RMP design. Furthermore, the scope will be on

GPRS technology, which is the ancestor for Enhanced Data GSM Environment and 3G technologies and thus sets the requirements for the RMP design.

Figure 4 illustrates the key components of the GSM/GPRS network. The components used in GPRS transmission are: Mobile Station (MS), Base Station Subsystem (BSS), and Network Switching Subsystem. MS refers to the end device such as a mobile phone or other network device with GSM radio interface. The BSS contains the Base Station Controller, Base Stations, and the Packet Control Unit. Packet Control Unit works as an interface for packet-switched traffic transferred from/to GSM radio network.

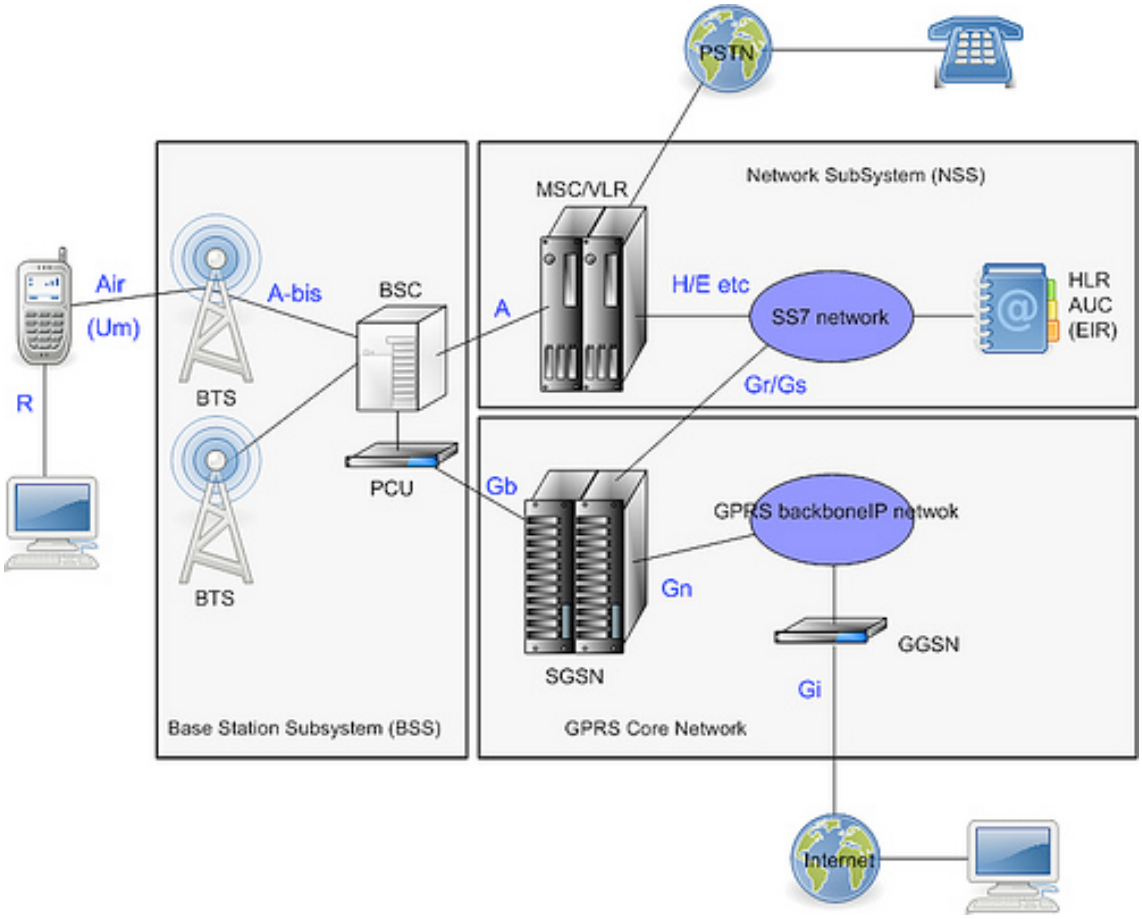

Figure 4. GSM/GPRS network architecture and key components [19]

GPRS consists of an IP backbone network that connects the BSS, the Internet, and the Public Land Mobile Network through the dedicated gateways (Figure 4). First, the GPRS core network is connected through Service GPRS Support Node into BSS. The Gateway GPRS Support Node (GGNS) functions as an

entry point from the GPRS core to the public Internet. Finally, the Public Land Mobile Network is connected via the border gateway (BG).

The packet-switched data traffic between the Mobile Station and the GGNS basis on a layered protocol model where the lower layer offers services to the one above (Figure 5). Furthermore, IP traffic in the GPRS core network consists of two different planes. First, the transport IP plane functions as a network layer protocol for the GPRS core network. Secondly, the user plane is used to carry the user IP traffic between the MS and the GGNS. This is, for example, where the RMP traffic will be carried. The user plane traffic is actually tunnelled between the MS and GGNS and can be performed in acknowledged or unacknowledged mode by the underlying protocols. The user plane link is considered to be reliable when the transport plane is driven in acknowledge mode, which is usually the case in mobile networks. In addition, the cost for the data transfer over a GPRS connection is based on the amount of data transfer on the user plane IP. The accounting includes the user plane IP packets and the upper layer data encapsulated into those packets. Therefore, the more data is transferred, the more money is paid. That is the very reason why RMP was required to be connectionless protocol so that the control traffic below the application layer could be minimized. [20], [21]

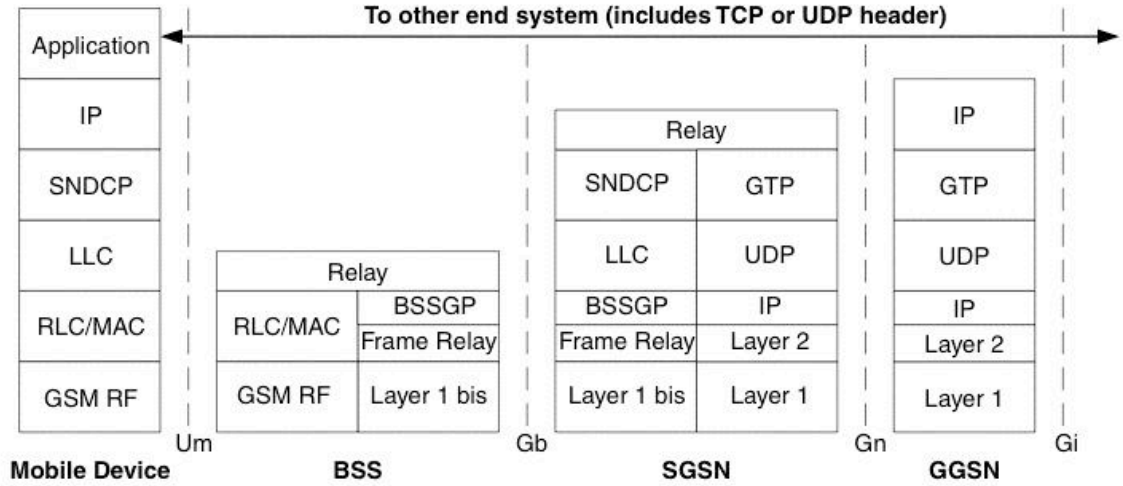

Figure 5. Layered protocol model between network components of the GPRS/GSM system. [20]

The PMTU size in the GPRS network is defined by the lowest interface MTU of the layer two protocol. Furthermore, the GPRS network configuration affects the size of the PMTU. How the GPRS network is configured depends on the providing mobile operator. [23]

3.2.2 Mobile Station Modes

Mobile stations are classified as A, B, and C by ETSI depending on how they are able to connect into GSM and GPRS networks. Class A devices can utilize the both networks simultaneously. A class A MS can receive GSM calls and GPRS calls at the same time. A class B MS functions as class A MS except that it cannot transfer data simultaneously. Finally, class C refers to devices that are able to connect both networks but not at the same time. A true class A mobile station requires two GSM radio interfaces in order to function as specified by ETSI. Although such a class A devices exists, they are hard to find and, in practice, all GPRS capable mobile stations belong to class B. This means that a GPRS data connection on class B MS is dropped if a higher priority GSM voice call occurs. [24]

3.2.3 GPRS Resource Sharing

The air interface of GSM uses a combination of Frequency Division Multiplexing, and Time Division Multiplexing (TDM) multiple access schemes. Frequency Division Multiplexing handles the frequency allocations for the communication channels and the TDM further divides each channel into 8 time slots, which together comprise the TDM frame(Figure 6). The TDM frames again are a part of a larger multi-frame. The time slots are divided between the GSM users sharing the same interface. GSM users use the seven of the eight slots and the one resting carries the control traffic. [25]

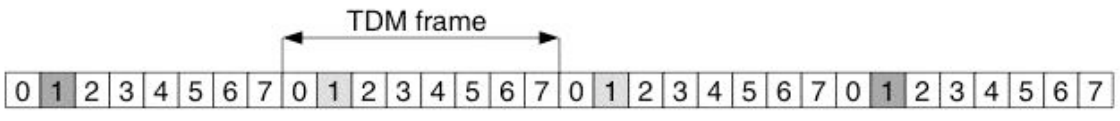

Figure 6. TDM time slots and frame [23]

GPRS traffic shares the same air interface as the GSM voice traffic and thus the time slots of TDM frames. When thinking of prioritization between GSM voice and GPRS data traffic, the GSM overrules the GPRS. Therefore, it may occur that GPRS connections may use only a single time slot, if many simultaneous GSM voice connections exist in the same air interface. Moreover, multiple GPRS connections may share a single time slot, which furthermore decreases the throughput of the connections. In addition, an existing GPRS connection is overruled and dropped from a time slot in case of a new GSM voice connection demanding a time slot and no other than GPRS is available. As a consequence, GPRS connections over the GSM air interface are considered unreliable resulting in bad throughput and connection loss. [25]

In order to transfer data over GPRS, the MS must attach and activate a data structure, called Packet Data Protocol context, which includes the user session information such as IP address, IMSI code, etc. After the activation, the GPRS Support Node and GGNS are ready to switch IP packets with MS. In order to send or receive IP packets, the MS must yet allocate the necessary timeslots from the GSM air-interface for the data transfer. If the data transfer between the MS and GGSN does not occur within a certain period, the timeslot allocation expires. The allocation causes delay into the system, which can be as long as couple of seconds for each transmission. [26]

3.3 Multi-protocol Label Switching

MPLS introduces the traditionally efficient layer two switching schema for layer three packets, combining many features such as security, routing, and QoS into a single service. Geometrically scattered private networks can be combined into a single global VPN using MPLS tunnelling over a public network(s) such as Internet Service Provider network or mobile operator’s network. This kind of global intranet offers great connectivity combined with security and QoS features.[27]

In such a schema, IP packets enter a public network from the Customer Edge (CE) router through a Provider Edge (PE) router. PE consults its routing table to resolve the path to the PE facing the CE of the destination private network. This path has been resolved beforehand using a routing protocol or configured statically on network layer. From that information, packets travelling to the same destination are included into a Forwarding Equivalent Class (FEC) and mapped to a label, this is called label binding. The PE enters the 32-bits long label between the IP header and the layer two header and forwards the packet to the next hop. The label is unique for each link and the MPLS routers keep track of the labels and corresponding links (incoming/outgoing interfaces) by exchanging the bindings and storing them in a Label Information Base (LIB). From now on, all the subsequent Label Switched Routers (LSR) perform the following. The label of incoming packet is examined and striped off, the LIB is consulted to find out the outgoing interface and the new label for the link, and finally the new label is inserted into the packet and the packet is forwarded to the next hop. The previous process can be performed by hardware and no layer three consulting is required, which makes the process fast. The process is repeated on each LSR until the packet arrives to the PE on the other end. PE stripes off the label and consults the LIB. The LIB does not contain any information for the outgoing interface, because the next hop for the packet will be the destination CE. Therefore, the PE initiates the regular layer three routing procedure and forwards the packet to the CE based on the IP header information of the packet. The CEs do not perform any MPLS-related tasks but are aware that the remote private network can be found behind the interface facing the PE and make the forwarding decisions based on that. [27]

The previously defined MPLS VPN includes the use of techniques such as Virtual Routing/Forwarding and Boarder Gateway Protocol signalling. In addition, when compared to VPNs using Internet Protocol Security, the MPLS VPN generates less overhead and, therefore, offers better throughput. However, the 32-bits overhead created by the MPLS label cannot be ignored when the PMTU is being calculated. [27]

3.4 Effects of the Network Infrastructure for RMP

Based on the topics discussed in this chapter, some conclusions can be made concerning the RMP design. The main issues that unfold from that discussion is how and by which layer the fragmentation/reassembling of IP packets exceeding the PMTU should be done. Among these, other issues arise such as the path MTU discovery problem, routing in NAT environment, and GPRS connection issues.

The fragmentation issue arises because of two reasons. First, the required maximum message size of RMP (64KB) cannot be transferred over any existing network link without being fragmented and reassembled at some point. This is because theoretically, the greatest MTU for a single link can only be around 9KB, and even that is rarely the case. The more realistic estimation for a link MTU would be somewhere between 1000 and 1500 bytes but even lower values may occur. The second issue mentioned earlier is that the RMP was specified to be a connectionless protocol and is thus lacking the fragmentation on transport layer. Therefore, the fragmentation may occur on the network layer (IP) or on the upper layer carried out by RMP itself or some other existing upper layer protocol. Letting the IP handle the fragmentation/reassembling would be a risk. The earlier discussion about problems of IPv4 and IPv6 fragmentation showed that the service is not reliable. IP fragmentation may function in some networks but may be disabled in some others. As the IP fragmentation is not an option, the upper layers must offer the fragmentation service. The next chapter covers these options.

The size of message fragments plays important role in data transmission efficiency. Therefore, when the fragmentation occurs at the sending device, the device must know the size of the PMTU. The PMTU size can be a configurable value in the protocol code or it can be automatically discovered. It is recommended that whenever UDP is used as transport layer protocol, some kind of path MTU discovery procedure should be implemented on the upper layers.

Routing over NAT routers is not an issue if the RMP protocol functions so that all the communication between the client and the server originates from the client. This approach allows only the SAG router at the client side to be configured with a static route pointing toward the remote monitoring centre network. In contrast, the remote monitoring centre does not need to know the address of the SAGs. This is convenient because there are many SAGs in the network but only one monitoring centre. Once the client initiates the communication, the return path over the NAT tables from the server to the client exists for a certain period. The period can be as long as two minutes, which allows the server to reply to the client. If the server acts as initiator for the communication, it would require more complex implementation of a routing schema. Even though the monitoring centre gateway and the numerous SAGs connect directly to the same network, the monitoring centre should still keep account of which SAG corresponds to which client equipment. This could be done by implementing a domain name server in the monitoring centre network.

The presence of the mobile networks in the system results in a certain unreliability in the network connections. Generally, wireless networks are more prone to interference than cable networks and connection loss can be expected to occur during the data transmission. Furthermore, the delay caused by timeslot allocation of GPRS may result in problems in the data transmission. The remote monitoring system must be able to recover if connection is lost during the transmission and some packets are lost. Therefore, it is vital to implement a flow control mechanism with message acknowledgement, retransmission, and message-duplication-detection capabilities into the system. Additionally, data transfer over the mobile networks is charged based on the amount of data transferred, which shall be taken into account in the RMP design.

4 Protocol Engineering

This chapter discusses protocol engineering, connectionless protocols on the transport layer and above, and authentication mechanisms. In protocol

engineering part, the basic protocol concepts, necessary for RMP, are explained. In addition, as RMP is to be a connectionless protocol, therefore the UDP is the only choice to be used on the transport layer and is discussed in this chapter. Furthermore, a UDPCP protocol is introduced as an example of an upper layer protocol, which functions on UDP and offers sequence control and fragmentations for application messages. Finally, some basic mutual authentication mechanisms are introduced and their suitability for RMP is discussed.

4.1 Protocol Mechanisms

In order to be a connectionless protocol functioning above UDP/IP, RMP shall address the following issues: sequence control, error detection, and re-transmission. In addition, as already mentioned that the use of IP fragmentation should be avoided, the issue of packets greater than MTU is left to be handled by RMP. This chapter will discuss sequence control mechanisms and a basic fragmentation procedure. In addition, some binary encoding rules are discussed.

4.1.1 Sequence Control and Error Control

The purpose of sequence control is to verify that the data transferred between two hosts can be successfully performed even when an error in transmission occurs. This requires that the protocol, responsible for sequence control, is able to detect errors when they occur and to perform the required correction actions. [28]

In any packet-based data transmission, packets may become lost during the data transfer. The result is that the application at the receiver receives no data or some data, and the system may fail. To avoid the latter situation, the protocol used for the data transmission should implement an acknowledgement mechanism with re-transmission, where the receiver acknowledges each data packet successfully received. The acknowledgement is done by sending an

Acknowledgement Packet (ACK) back to the sender. Each data packet has a unique identifier, which matches the data packet and the ACK packet. The sender must also store the packet into a buffer until the ACK is successfully received. In case that the sender does not receive the ACK within a certain period, the sender considers the transfer unsuccessful and re-transmits the packet. [28]

Sending the ACK packet is not a sufficient method if errors detection occurs on bit level at the receiver. For that, the data packet needs to include a checksum field and the ACK packet must be able to indicate to the sender whether the received packet was valid or corrupted. A simple checksum system performs a checksum calculation over the whole data portion of the packet. The idea is that the sender calculates the checksum value before the transmission and includes the result to the packet header. The receiver performs the same calculation and compares its result to the value found from the header. If the values match, the data in the packet are valid and if not, the packet is corrupted and dropped. In case of a corrupted packet, the receiver may just silently ignore the message and wait for the sender to re-send the packet when the timeout occurs. Alternatively, the receiver can indicate to the sender the packet re-transmission by sending a so-called negative ACK (NACK) packet back to sender. NACK is an ACK packet having a flag bit set to indicate the negative acknowledgement. Sending the NACK decreases the system delay because the sender does not need to wait for the timeout before the re-transmission. [28]

Duplicated packet reception is another error situation, which may result in halting the system , if not dealt properly. Packet duplication usually occurs when the receiver has successfully received a data packet and sends the ACK packet to the receiver but for some reason the ACK never gets back to the sender. The result is that the sender waits for the re-transmission timeout and re-sends the data packet to the receiver. To avoid errors, the receiver must be able to detect that the packet is duplication and silently ignore the packet. Message ID, when properly implemented, can be used to detect packet duplications. For example, the receiver can consult the message ID of each new received packet and

compare it to the one received before; if the values match the packet is duplication. The previous system requires that the message ID or sequence ID is a running counter that is incremented after each new packet creation and re-set in a pre-specified manner when the maximum value is reached. [28]

The previously described ACK/NACK system is only capable of detecting error but not correcting them at the receiver end. This approach is well-suited for systems that are not susceptible todelay. If the delay in transmission is a problem, the system must implement yet another mechanism. However, the error correction capabilities at the receiver end are not in the scope of this thesis and shall be left out of the discussion. [28]

4.1.2 Segmentation and Reassembly

The fragmentation was already discussed in Chapter 3 as a part of the IP protocol. As the IP fragmentation was discovered to be unreliable, this section introduces the necessary factors that an upper layer protocol must implement in order to perform fragmentation services.

A functional fragmentation / re-assembly system must be able to automatically segment the messages to confront the size of the currently valid path MTU. This is done by fragmenting the message into pieces and sending the pieces to the receiver where the message is reconstructed by a re-assembling service. The pieces, fragments, shall be labelled so that the re-assembling is possible at the receiver. A fragment consists of a header and partial amount of data from the original message. The header shall include at least two fields for fragmentation information, the total number of fragments completing the message and the order number that indicates from which part of the message the data carried in the fragment originates. This information is needed when the message is reconstructed at the receiver. Additionally, the system must know the valid PMTU value. PMTU can be a parameter value, which can be configured if changes in the network architecture occur or it can be automatically detected by the system. The manual configuration method is valid if the network architecture can be pre-defined and is not expected to alter. The automatic PMTU discovery

algorithm is necessary in cases when the underlying network topology cannot be determined and is susceptible to changes. [29]

4.2 Protocol Encoding

Encoding defines the rules for the representation of data within a protocol data unit in a way that all the parties in communication are able to interpret the received data. Each protocol has different demands for the encoding rules and they shall be carefully determined. Some of the main guidelines for determination of protocol encoding rules are: efficiency, delimiting, ease of encoding, and data transparency [30]. Efficient encoding refers to bandwidth saving. The data in PDUs is encoded so that it uses the minimal amount of bandwidth. The end and the beginning of data fields as well as PDUs should be easy to detect. The receiver should easily be able to determine the received data from the PDU. The encoding should allow the data fields in PDU to carry arbitrary sequences of bits without causing problems while decoding the fields.

Practically, the encoding rules contain three main categories: binary, type-length-value (TLV), and matched tag encodings. Because RMP is to transfer simple binary values or parameters that, in many cases, can be predefined, the sufficient encoding schema will delimit to binary and TLV encodings. The following sections shortly introduce these systems.

4.2.1 Binary Encoding

In simple binary encoding, data field within the PDU have a pre-defined position and length that are known by all the parties in communication. Binary encoding facilitates the decoding process when individual data fields can be masked out from the PDU. The data in PDU headers is usually encoded using fixed binary encoding because it allows quick identification of PDUs through the masking operations. [30]

4.2.2 Type-Length-Value Encoding

In Type-Length-Value (TLV) encoding, the representation of data fields includes a type field, a length field, and a value field. The type field is a unique identifier for the data carried in the value field. The types, or tags as they are often called, are pre-defined and known to all parties. The length field indicates the length of the data string in the value field. The length is usually represented in bytes. The value field contains the actual data of the data field.

TLV encoding offers flexibility, and forward and backward compatibility. The length information provides flexibility allowing fields of varying length. The receiver is able to detect the length of the each data field from the length portion and use the information while parsing the field. In addition, the system allows receiving unknown data types without errors. When the receiver detects an unknown data type from the type field, it can simply consult the length field and jump over the value portion to the next possible data type while parsing the PDU. Similarly, adding new data fields into the protocol is easy and flexible. [30]

4.3 User Datagram Protocol

UDP offers connectionless data transmission in IP-networks. UDP functions on top of the IP protocol on the transport layer. UDP offers multiplexing between applications by the use of port numbers. In UDP data transfer, the UDP packets are sent to the destination without any connection negotiation between the sender and the receiver. The applications using UDP only define the destination UDP port number and the destination IP address before the data transmission. The packets are routed to the destination by IP based on the IP address and the correct UDP process at the receiver is mapped based on the UDP port number. UDP transmissions are unreliable, which means that UDP does not guarantee the success of the delivery. An upper layer protocol or applications using UDP shall implement the necessary transmission control mechanisms, if required. However, UDP offers an option for minimum error detection by allowing the application to use the UDP checksum field in the UDP packet. [31]

Applications discuss with UDP through a socket, which is created whenever the UDP service is needed. The application creates the socket during its initialization procedure, binds it to a UDP port, and can use it for data transmissions with many different destination hosts. The socket concept is part of the Berkeley Standard Socket Application Programming Interface (BSD API) that defines a C-library for network programming. [32]

UDP offers an efficient way for data transmission without any connection or other control messaging between the sender and the receiver. UDP just pushes the packets towards the receiver and hopes that no error during the transmission occurs. The connectionless nature of UDP makes it ideal for the systems that need to send variable amounts of data inconsistently and where the application or some upper layer protocol offers the reliability services for the data transfer. This approach reduces the usage of the network bandwidth. [31]

4.3.1 UDP header

The Figure 7 illustrates the UDP header format. The header has four 16-bit fields, the source port, the destination port, the message length and the UDP checksum. The complete UDP packet size is limited to a maximum IP payload, which is 65,507 bytes [33].

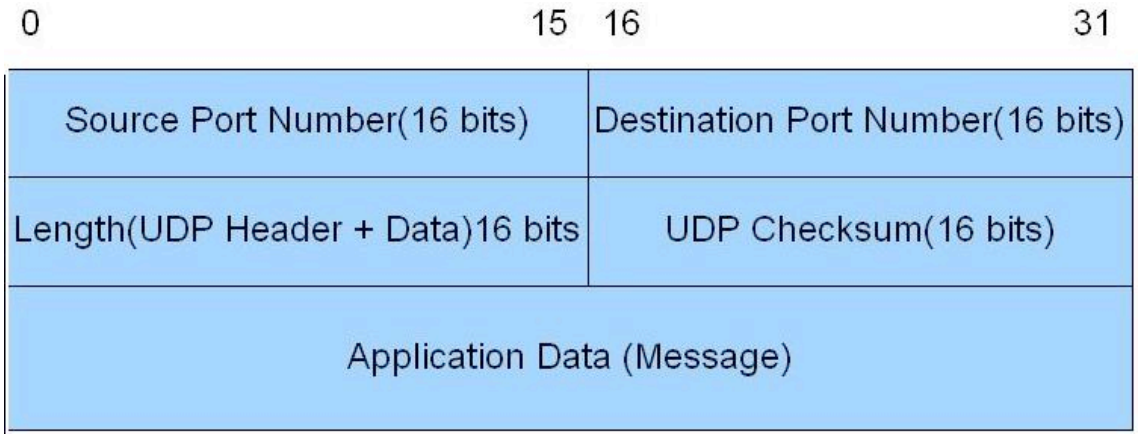

Figure 7. Description, length, and order of fields in UDP packet header [34]

A source port field indicates the port number of the UDP process in the sending host. The destination port field determines the port number of the UDP process

in the receiving host. The length field exposes the length of the complete UDP packet including the UDP header and the application data.

The receiving host uses the UDP checksum field for error detection purposes. UDP calculates the checksum over the whole UDP packet and a so-called Pseudo header (Figure 8). The Pseudo header is not part of the UDP packet but contains the IP address, protocol version and UDP packet length information from the IP header. The checksum field is set to zero during the calculation. The checksum calculation is performed as an addition of the 16-bit data block so that the result of two first blocks is added to the third block and so on. The process is continued until the final 16-bit block. The result of that final adding is taken as one's compliment and added to the UDP checksum field. One's compliment refers to a process where all the ones in a binary word convert to zeros and vice versa.

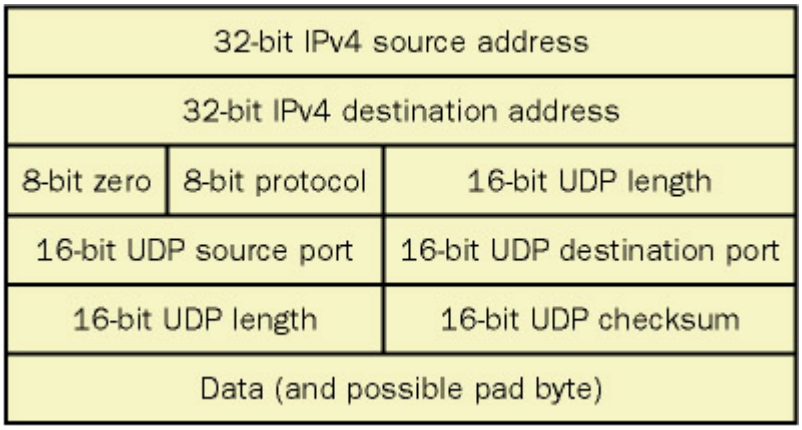

Figure 8. UDP checksum calculation is performed over the IPv4 Pseudo header and the complete UDP packet [35]

The receiver queries the Pseudo header from the IP layer and does an identical calculation for the received UDP packet and the Pseudo header. The result is added to the value from the received checksum field and if the addition gives 11111111 11111111 as a result the packet is correctly received. If one of the bits is zero, the packet contains an error and the UDP process silently discharges the packet. [36]

The UDP checksum mainly verifies that the packet has reached the correct destination but, at the same time, it offers a simple error detection mechanism

for the data transmission. The UDP checksum is an optional feature and the checksum field is set to all zeros when the feature is not in use. [36]

The data provided in the IPv6 Pseudo header differ from the one of IPv4. Figure 9 defines the fields used in the UDP checksum calculation with IPv6. [35]

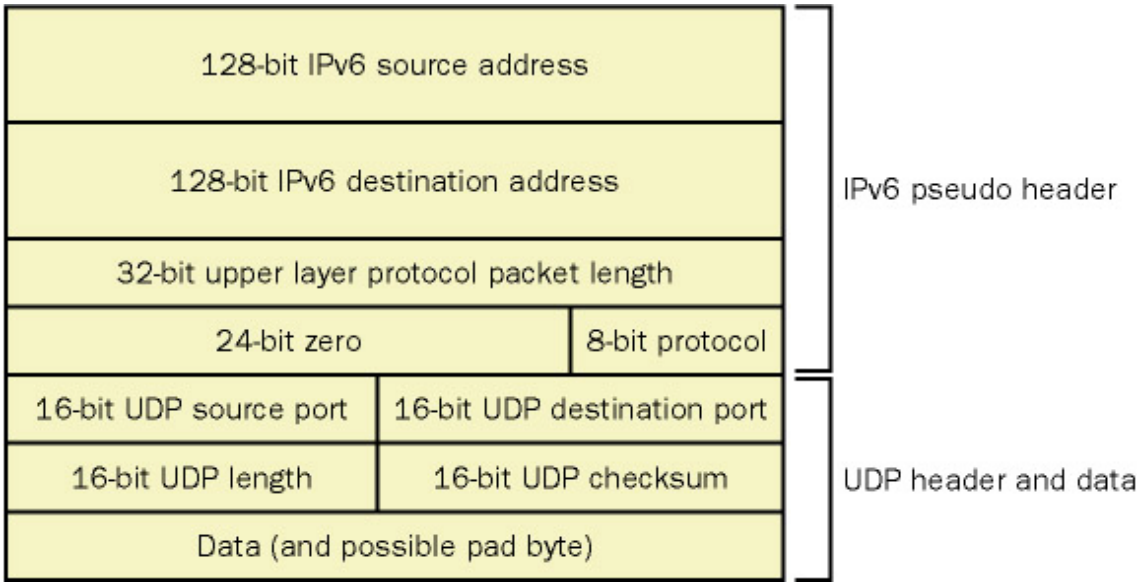

Figure 9. UDP checksum calculation is performed over the IPv6 pseudo header and the complete UDP packet [35]

4.4 UDP-based Communication Protocol

UDPCP is an upper layer protocol functioning over UDP. Open Base Station Architecture Initiative specifies the UDPCP protocol in its reference point 1 specifications [37]. The main purpose of UDPCP protocol was to offer reliable and unreliable data transfer over UDP/IP in the base station of a GSM network but it functions as a general solution in any UDP/IP network. The concept of UDPCP defines two types of packets: acknowledge packet and data packet. The data packet contains the header and data portions. The header is 12 bytes long and the payload can be up to 64 KB. The UDPCP process fragments the packets bypassing the MTU size. The acknowledge packet contains only the UDPCP header and acknowledges the data packet at the UDPCP level. The Msg type field in the header distinguishes the packet types. UDPCP can function in either acknowledged mode or unacknowledged mode when

transporting its data packets. Additionally, packet fragments can be sent so that either each fragment is acknowledged or just the one that completes the message is acknowledged. Flag setting in the packet header indicates the different acknowledgement modes. Moreover, UDPCP offers re-transmission for corrupted or unacknowledged packets, and is able to detect duplicated data packets at the receiver end. In addition, UDPCP specifies an API for upper layer.

Over all, UDPCP offers a flexible way for data transfer over the UDP/IP network. It specifies all the key functions for error detection and re-transmission and provides reliability for UDP-based data transfer yet being connectionless protocol. As a drawback, UDPCP does not specify any path MTU discovery procedure.

4.5 Authentication

RMP shall implement a mutual authentication procedure as a security mechanism. This section describes the basic concept of an entity authentication with a centralized management. In addition, the section introduces the Challenge-Handshake Protocol (CHAP) as an example of a shared secret authentication and the Extended Authentication Protocol (EAP) framework. The GSM authentication procedure is also shortly discussed.

4.5.1 Entity authentication requirements

The concept of entity authentication includes four parts: evidence, non-transferability, no third-party impersonation, and no leakage. The term evidence refers to procedure where both of the entities, the sender and the receiver, shall be successfully identified before the transmission between two parties can occur. Non-transferability guarantees that none of the two sides can impersonate each other whereas no third-party impersonation excludes the possibility of an unauthorized outsider to act as one of the authorized entities. Finally, the authentication system is said to have no leakage if the previously

defined three factors can be fulfilled despite the number of executions of the authentication protocol between the two sides. [38]

Authentication includes three main categories depending on their security level provided by the used identification method. First, weak authentication uses a simple password or a Personal Identification Number as credential for entity authentication. Second, strong authentication is based on a challenge request and a response generated from the value in the request message using a shared secret key. Finally, Zero knowledge authentication refers to the system where the knowledge of the secret is demonstrated without revealing the secret at all. From the three categories above, the strong authentication method can provide sufficiently enough protection for RMP. Therefore, the method is discussed in more detail in the following paragraph. [38]

4.5.2 Authentication with Shared Secret

The shared secret authentication is based on the secret key, which is known only to the parties participating into authentication procedure. The secret key is pre-shared for the devices and never sent over the network connection. Another key component of the system is a challenge value call nonce. The nonce is a randomly generated value that is used only once during the authentication procedure. The entity that initiates the authentication, let it be called A, generates the nonce and sends it to its counterpart, B, as a challenge message. B receives the nonce and performs a pre-defined function, known to both of the entities, that includes the nonce and the secret key. The function returns a response value, which is sent back to A by B. A performs the same function to its own secret key ? and the nonce it generated, and compares the result to the value received from B. If the values match, A can be sure that B possesses the same secret and thus B has proven its identity to A. In mutual authentication A must also prove its identity to B before the authentication is through to be successful. Therefore, B must generate its own nonce to be included into the response message to A. Once A has proven the identity of B, it respectively

calculates a response message to B using the same function with its own shared secret and the nonce generated by B. Finally, A sends the response to B where B performs the same identity verification as A did for B. When A is proven its identity to B the mutual authentication is said to be successful. [38]

4.5.3 Authentication protocols

CHAP is an example of an authentication method using a shared secret. Point-to-point links uses CHAP to authenticate the peer entity. CHAP can revalidate the authentication at random intervals during the link connection. [39], [40]

EAP is an authentication framework providing a very flexible way to implement an authentication procedure into a system. EAP can encapsulate many different authentication protocol messages inside the EAP frame and carry them over various different media such as Ethernet, Wireless, point-to-point, etc, links. [41]

In the basic concept, EAP authentication occurs between the authenticator and a client. The authenticator is usually a gateway device of the network which the external client is willing to access. The basic concept provides only the client authentication procedure, but mutual authentication can be performed, if after the client authentication the counterparts change roles and repeat the authentication procedure vice versa. EAP can also be used in conjunction with centralized authentication, authorization, and accounting protocols such as RADIUS. In case of RADIUS providing authentication, the authenticator acts only as an intermediary between the client and the authentication server. The RADIUS packets encapsulate the EAP messages between the authenticator and the RADIUS server, which allows only the RADIUS server to know the EAP message types and the authenticator can act as transparent middlemen [42]. [41]

The EAP frame has three fixed fields: code, identifier, and length (Table 1). The code identifies the EAP message code, which can be either code 1: request, code 2: response, code 3: success, or code 4: failure. EAP uses the identifier

field to match the request response message. The length field indicates the total length of the message. Additionally, the EAP frame consists of type and the type data fields. The type field defines the authentication method used for the authentication and the type date field carries the data needed in the authentication process. Internet Engineering Task Force has defined many authentication methods to be used with EAP [42]. Perhaps the most common and relatively secure, when used in private networks, authentication method used with EAP is Message Digest 5 Algorithm (MD5). The type for EAP-MD5 is four. The type data field carries the random nonce in the request message and the response value calculated from the nonce and the shared secret in the response message. Additionally, it is possible to specify a vendor-specific authentication method using the expanded type identifier 254 in the type field and vendor ID in the type data field. The use of type 254 includes two additional fields in the EAP frame vendor type and vendor data fields. The details for the use of expanded authentication type can be found in the following table [42].

Table 1. EAP frame consists of three fixed fields as a header and the type and type data fields for identifying and carrying the authentication method data

8	16	32
Code	Identifier	Length
Type	Type data	

Figure 10 illustrates the message exchange patterns for mutual authentication procedure with EAP. EAP features include retransmission and message duplication detection. The authenticator is responsible for all the re-transmissions. If the client does not respond within the timeout period, the authenticator shall re-transmit the request. EAP handles duplicated messages by silently discharging all unexpected messages. Additionally, EAP is a lock-step protocol sending only one packet at the time.

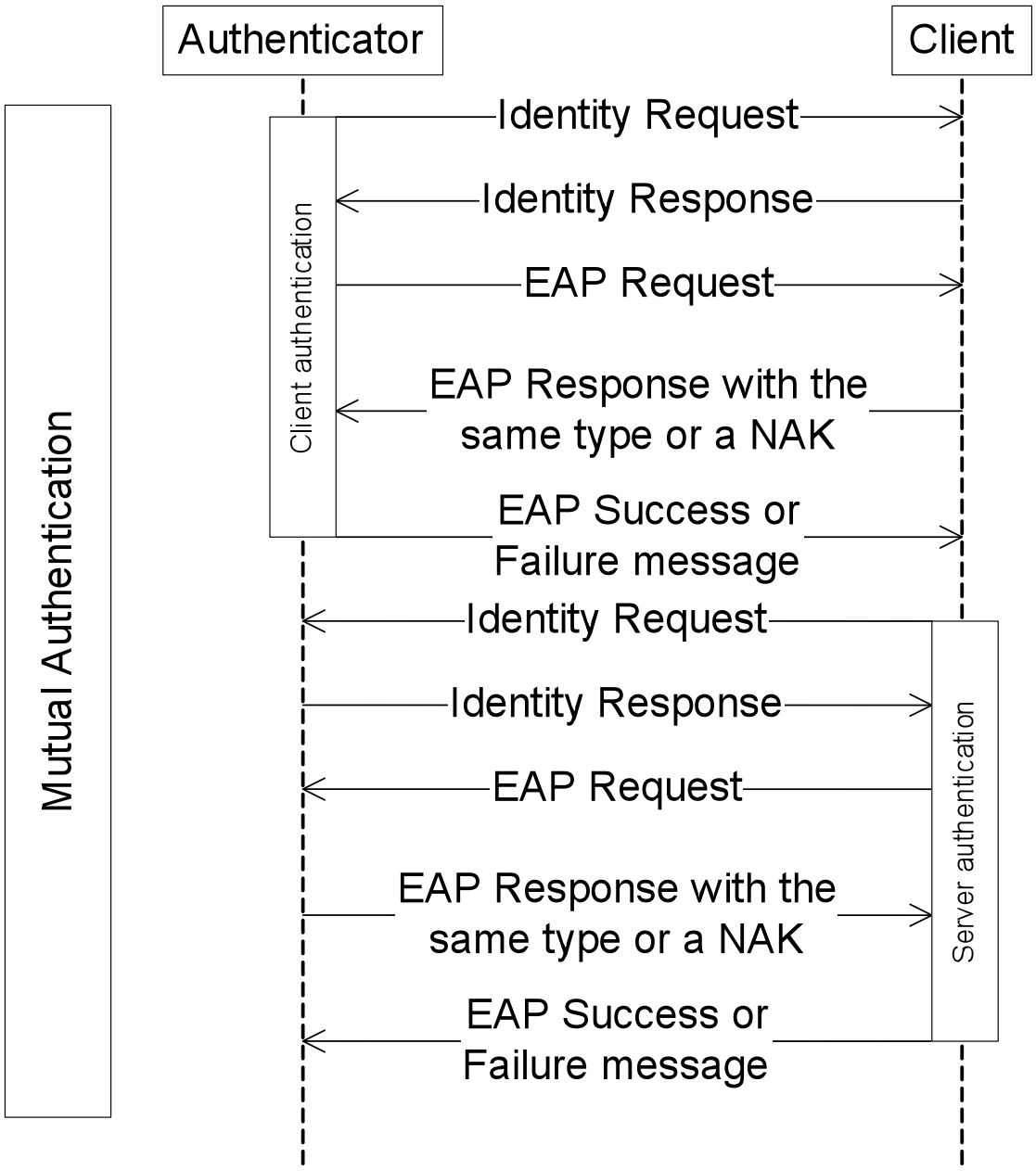

Figure 10. The message exchange sequences in mutual authentication of EAP

EAP is susceptible to network attacks, such as man-in-the-middle and the session hijacking attacks. However, when EAP is used over a secure network link, the risk of being attacked decreases. The more secure version of EAP is Protected EAP. Protected EAP performs the authentication in two steps. First, it negotiates a secure TLS tunnel between the entities and, then it performs the authentication process thought the secure tunnel [43].

GSM authentication uses the International Mobile Subscriber Identity code stored in the SIM-card of the MS as an identifier for authentication. MS authenticates with the home local register through the mobile switching centre, which functions as an intermediary in the process. In GSM authentication only the MS is authenticated which makes it susceptible to the type of network attacks where HLR is impersonated. [44]

The Remote Authentication Dial In User Service (RADIUS) protocol provides centralized authentication services for clients requesting access to a network. RADIUS topology includes the RADIUS server, Network Access Server (NAS), and the authentication clients. NAS works as client for the RADIUS server and is able to encapsulate the authentication messages from the authenticating client into the RADIUS packet and pass them to the RADIUS server. Controversially, NAS removes the RADIUS encapsulation of message from the RADIUS server to the client while forwarding them. The RADIUS server authenticates the client and authorizes the NAS to grant network access for the client if the authentication is successful. RADIUS uses UDP on the transport layer listening the ports 1812 or 1645.

RADIUS supports many authentication protocols such as Point-to-Point Password Authentication Protocol or CHAP, UNIX login, IEEE 802.1X, etc. Additionally, RADIUS functions with the EAP framework as mentioned before. The Internet Engineering Task Force document of RFC 2865 specifies the RADIUS. [45]

4.6 Discussion of RMP design

This section discusses the necessary protocol mechanisms to be implemented in RMP in order to fulfill the specified requirements. These include sequence control, fragmentation, error detection, message encoding, and authentication procedure.

The sequence control is needed when UDP is used as the transport layer protocol. One solution is to use UDPCP, which offers this service. In addition, UDPCP would be a convenient solution, because of its fragmentation feature.

However, implementing UDPCP under RMP would cause an extra data encapsulation and thus more overhead into the system. This again results in increased data transfer costs in the system, which can include hundreds of thousands reporting client devices. On the other hand, RMP can be specified to offer both the sequence control and the fragmentation internally. In that case, overhead is reduced and the sequence control as well as the fragmentation can be tailored to better meet the requirements of RMP. The tailoring is needed, for example, in case of using GPRS connection. This approach also clarifies the protocol design and thus eases the implementation process.

The UDP checksum feature can provide sufficient error detection capabilities for RMP. The sender can re-transmit the packet after a timeout, if the UDP process at the receiver finds an error. This causes some delay in the system, which is suitable for RMP not having real-time requirements. The use of UDP also reduces overhead when the RMP header can be left without the checksum field.

The RMP header PDU shall include two encoding schemas. First, the most efficient encoding schema for RMP header is the simple binary encoding with fixed field lengths. Second, the payload portion shall use the TLV encoding schema, which allows application messages to be flexibly specified.

In order to be more generic and flexible, RMP shall use EAP for authentication purposes. This allows RMP to be used with several different authentication methods, which can be decided while implementing the protocol. In addition, the use of EAP makes it easier to adapt RMP to an existing authentication schema.

5 Proposed Solution

5.1 Introduction

This chapter specifies Remote Monitoring Protocol for data communication between equipment analyzers and the Remote Monitoring Centre. The specification contains the complete RMP layer and parts of the application layer accordingly (Figure 11). RMP is an upper layer protocol, which is based on the connectionless UDP protocol on the transport layer. RMP uses standard Berkley Software Distribution API for data exchange with the UDP process. In addition to the RMP is the analyzer application at the equipment end or the remote monitoring server applications at the RMC end. The applications are bound to RMP through the RMP API. RMP is a client-oriented protocol using the request/response message exchange model. RMP offers fragmentation/re-assembling for all application messages that overstep the configured PMTU. The analyzer application can automatically discover the PMTU using the discovery procedure of RMP. In addition, the application has an option to configure the acknowledgements and re-transmissions on the RMP level. Furthermore, RMP specifies the message exchange procedures between the analyzer and RMS applications. The following list summarizes the RMP features:

- Message exchange procedures between the analyzer and the RMS applications
- Data structure for application messages
- Message fragmentation and reassembling service
- Optional PDU acknowledgements on RMP level
- Client redirection
- Action list
- PMTU discovery procedure
- RMP API

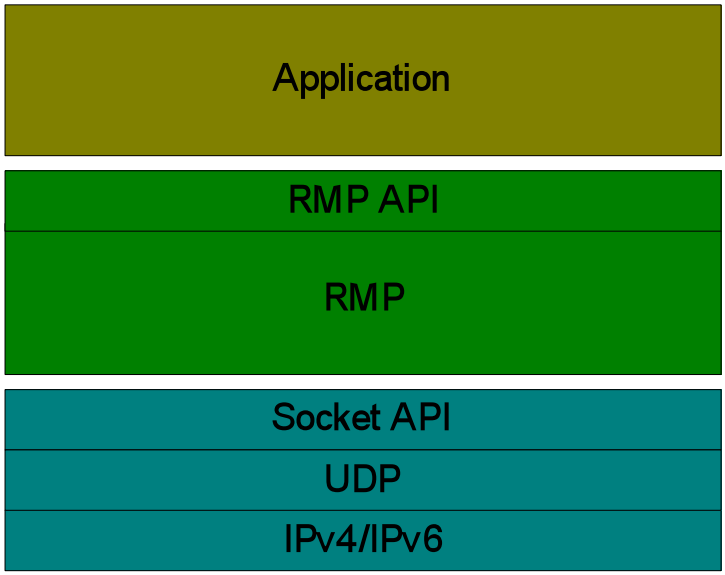

Figure 11. Layered model of RMP

5.2 Routing

RMP is a client-oriented protocol where all the conversations between the analyzer and RMS originate from the analyzer. This is due to the three different IP address spaces over which the message exchange shall occur. The routing in the system is performed as follows: Clients are pre-configured with a default RMS address. SAGs query the default gateway address dynamically from the access point network. The MPLS backbone switches the packets between the APN and the RMC gateways. The previously described routing scenario allows any client to send a request message RMS. RMS is able to reply to each request within the timeframe of a NAT address translation binding period. In case of RMS requesting the client, RMS can set an Action Flag (AF) bit in the RMP header to invoke an action request procedure from the client. RMS can use the AF function when it wants to push data to the analyzer. An example of the usage of AF is when the RMS has a new configuration for the analyzer or when RMC personnel want the client to send a BOM message from the client.

5.3 Action List

The equipment specific action list is part of the information database in RMS. Actions in the action list are pre-defined and known by both entities. That is, the client must be able to understand and perform the requested actions. This specification defines two actions that can be set to the action list. These are downloading a new analyzer configuration and sending the BOM message. RMS invokes the action request procedure for the particular analyzer if the actions list contains some actions. The indication is done by setting the AF bit to one in the RMP header of any response message to the analyzer. The RMS continues setting the AF until the client has successfully performed all the actions in the action.

5.4 Path MTU Discovery

As default, the PMTU size is configured into RMP through the RMP API. However, RMP also specifies a PMTU discovery procedure that allows RMP to automatically detect the size of PMTU. The procedure is initiated from the analyzer application. The discovery procedure is mainly considered to be used in a situation where the network is not responding. In other words, response messages are not received despite various re-transmissions.

5.5 Behavioural State Diagrams

The following figures illustrate the behavioural states and the transition between the states. Both entities, the analyzer and the server, are described. In the diagrams, a rounded square illustrates a program state and an arrow defines how a transition from a state to another may occur. An event or a fulfilment of a condition triggers the transition. A transition may also require some actions to be performed before the next state can be achieved. The notation of the events, conditions and actions in the diagrams follows the ensuing model: event name [condition] / actions. The statement inside the square brackets shall be true in order to fulfilment of the condition.

5.5.1 Analyzer

Figure 12 illustrates the behavioral states of the equipment analyzer. The initialization steps include the disable, authenticating, and registering states. The idle state is where the analyzer performs the reporting routines to the active RMS.

The RMP client transitions directly into disabled state once it has been powered on. Only authorized users can enable the analyzer from the disabled state. Once enabled, the RMP client performs the mutual authentication procedure with the default RMC. RMP uses the EAP authentication procedure with the corresponding messages. The client re-launches the authentication procedure until the EAP success message from the gateway is received or the authentication retry period expires. If the re-try period expires, the RMP client must transition back to the disabled state. The EAP procedure is specified in Section 5.11.1. The next client state is the registration. RMP sends the registration request to the default RMS. The registration continues until success or the authentication timeout. If the RMS accepts the registration, it has an option to invoke the action procedure at the analyzer by setting the action flag of the registration response message. This is done only, if the action list for the specific analyzer has some actions to be performed. The RMP client transitions to the idle state after the registration procedure is successfully performed. The client now verifies whether it has a valid configuration. If the client is missing the configuration, the client queries the configuration from RMS. Once the configuration procedure is successfully terminated, the RMP client starts to perform the routine reporting activities to RMS. The following events results in the analyzer reporting to the RMS: fault occurrence in the equipment, a state change in an existing fault, time for the statistics report, or action flag detection. Furthermore, the client transitions back to the authenticating state when the authentication period expires to renew the authentication with the active AS. Finally, the client transitions to the disabled state when the equipment monitoring is turned off by an authorized user.

The RMP client performs the reporting procedures sequentially, so that only one procedure is executed at the time. Therefore, a new message exchange procedure may only be initiated if the client is in the idle state. When an event or fulfillment of a condition triggers one of the message exchange procedures, the client composes the corresponding message and transitions to the sending state. Once the RMP has successfully sent the message and no response from RMS is expected, the client transitions back to the idle state. Alternatively, the RMP transitions to wait for the response state, if a response from RMS is expected. A message having an even message is a request message and must always receive a response. Odd message types are response messages and do not receive response. A timeout transitions the client from waiting-for-response state to idle state, if the response is not received. In case of the reception of the response message, RMP verifies whether the message is a fragment or a complete message. In case of a complete message, the client performs the necessary actions. When a fragment is received, RMP transitions to wait for the next fragment state. When all the fragments are received, RMP completes the message and continues to process as with the complete message. The message under the re-assembling process is discharged if a new message (different message type) arrives before the message is completed. The fragments are also discharged if timeout occurs before the reception of all the fragments.

A message received while in idle state, is processed as it was received in wait for response state.

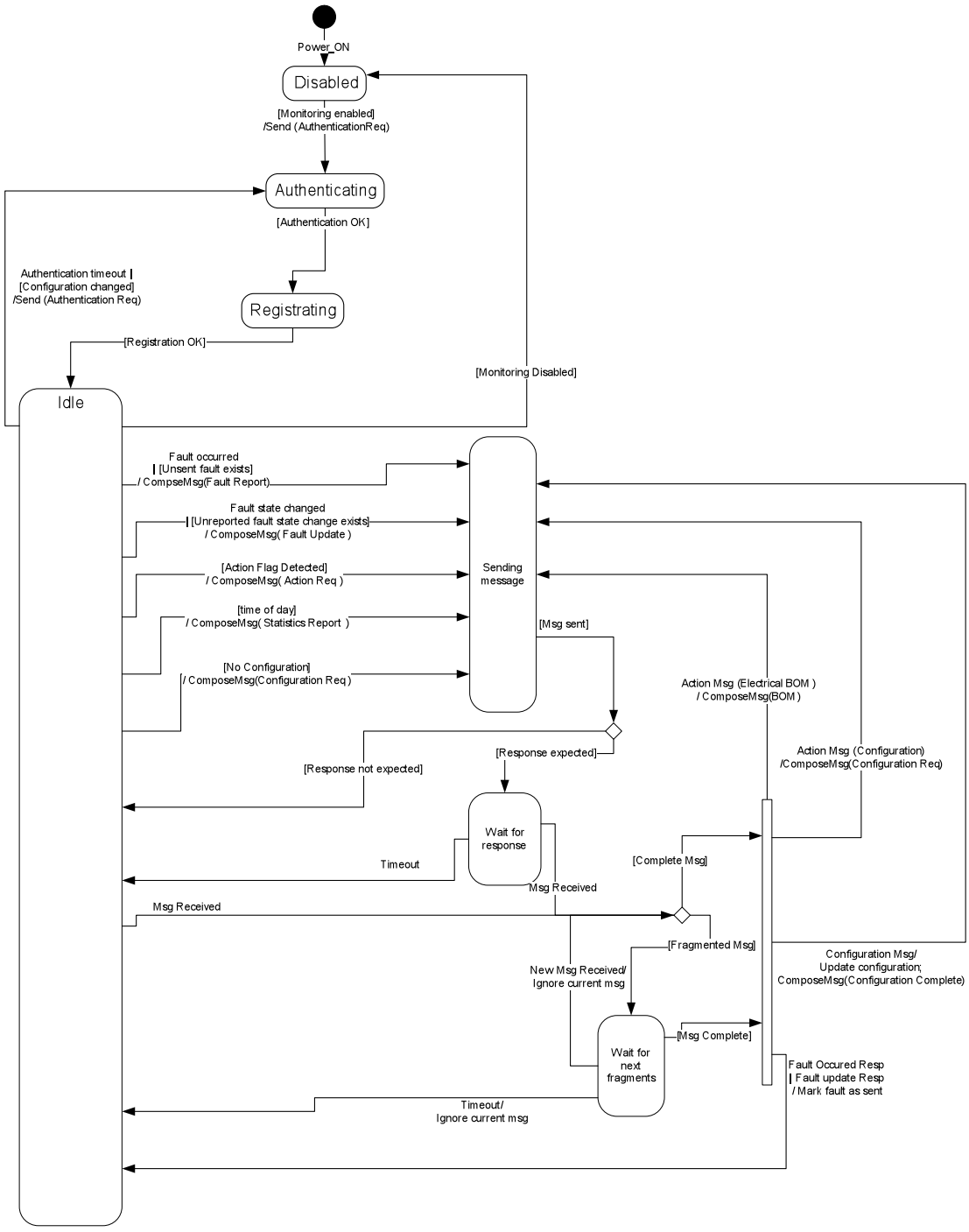

Figure 12. Behavioural State Diagram of Equipment Analyzer

5.5.2 Remote Monitoring Server

The remote monitoring server must operate with high performance, in multi-threaded environment, in order to serve many clients simultaneously. RMS can receive messages only in the active state in where it should stay as much as possible (Figure 13). After a message reception, the server verifies the registration of the sending client. Messages from the unregistered clients are discharged silently. Next, the need for re-assembling is verified. In case of a fragmented message, the buffer size is allocated when the first fragment is received. The buffer size can be determined from the RMP header. Additional fragments are added into the buffer when they are received. All the fragments must arrive within a configurable time, otherwise the buffer is freed. The received message may require additional actions such as sending a response message, or updating the action list. Otherwise, RMS transitions back to the active state. The action flag is set for the messages addressed to the clients having unperformed actions in their action list.

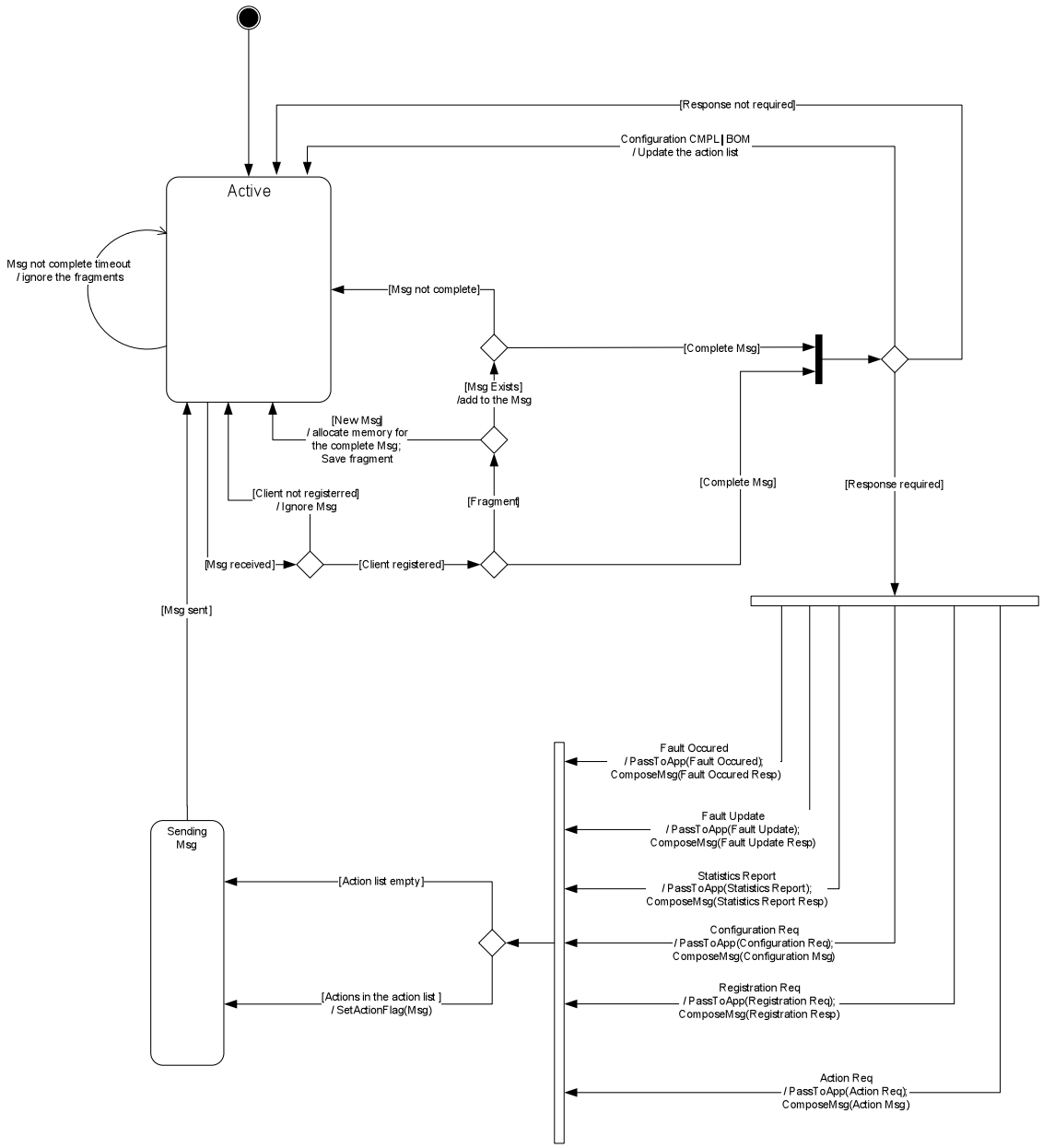

Figure 13. Behavioural state diagram of RMS

5.6 RMP PDU

RMP consists of two different types of PDUs, dataPDU and ackPDU. The data PDU contains the message header and the additional data fields to carry application data. The following diagram illustrates the RMP data PDU.

Figure 14. Structure of the RMP PDU: Fixed length header and TLV-fields as payload

The ackPDU contains only the message header. The purpose of ackPDU is to acknowledge the dataPDUs. RMP sends the ackPDU only if the received data PDU so indicates. The Flag-octet of the data PDU header defines the need for sending the ackPDU.

5.7 RMP Header

The RMP header has eight fixed length binary encoded fields (Table 2). The complete length of the header is 96 bits. The following section describes each of the fields in detail.

Table 2. RMP header

bit	8	16	24	32
0	Device category	Flag	Message type	Message ID
32	Client Device ID			
64	Message data length		Fragment amount	Fragment number
96	Message data			

The first field in the RMP header is the device category field, which is an 8-bit long field. The device category field identifies the group of similar type devices to which the reporting client belongs. The field size allows 256 different categories to be defined. The next field in the header is the flag-octet. The flag-octet is divided in two portions, flags and version (Table 3). The flags portion consists of the three LSB bits of the octet. The flag-bits are used to signal the recipient about the nature of the message. The bits 0 (PT) defines whether the

PDU type is data or ack. The following bit 1 (RACK) indicates the receiver whether the PDU is expected to be acknowledged or not. The bit 3 invokes the analyzer to send the action request message to the RMS. Only RMS can set the bit 3. RACK and AF bits can be accessed via RMP API. The version portion (four most significant bits) of the octet is reserved to indicate the future versions of the protocol inside the device category.

Table 3. Flag-octet

Name	Bit	Value	Description	Applies
PDU type (PT)	0	0	data PDU	Only for RMP
		1	ack PDU	
ACK Required (RACK)	1	0	No ACK required for this packet, for ack PDU always zero	Only for RMP
		1	Protocol level ACK required for this packet, for data PDU only	
Action Flag (AF)	2	0	No further action required from client	Only for the analyzer application
		1	Action request required from client	
Reserved	3		Reserved to indicate incremental protocol version inside device category	
	4			
	5			
	6			
	7			

After the flag-octet, the 8-bit long message type field identifies the application messages (

Table 4). The numbering follows the pattern where all the request messages use even numbers and the response messages use odd numbers. The system allows 128 different request/response pairs to be defined. The following table specifies the message types used between the analyzer and RMS applications.

Table 4. Application message types

Message Type	Description
0	Reserved
1	Reserved
2	Reserved
3	Reserved
4	PMTU request
5	PMTU response
6	Registration request
7	Registration response
8	Configuration request
9	Reserved
10	Configuration
11	Configuration response
12	Fault occurred
13	Fault occurred response
14	Fault update
15	Fault update response
16	Statistics report
17	Statistics report response
18	Action request
19	Action response
20	BOM request
21	BOM response

The fourth field of the RMP header is the message ID. The value of the message ID field is an 8-bit running counter that identifies the message. The counter is incremented by one after each usage. Once the counter reaches the maximum value, it is reset to zero. The same counter can be used for all clients at the RMS, no separated counters are required. The message ID value of a ackPDU is a copy of the dataPDU being acknowledged. Whereas the device category field identifies a larger group of devices, the 32-bit long device ID field uniquely identifies the client device. Both the request and response messages must always contain the device ID of the reporting equipment. RMS uses the

device ID to identify and register the equipments. All messages with unknown equipment ID shall be silently discharged. The last three fields of RMP carry the information needed in fragmentation and re-assembling processes. The message-data-length field indicates the length of the complete RMP payload. This is also indicated in the message-data-length field of each fragment of the fragmented message. Thus, the receiver can directly allocate the correct amount of memory for the complete message after the first fragment is received. The message length is 16-bits long and indicates the message length in bytes. The following fragment amount field has two functions. First, it tells the total number of the fragments that comprise the complete message. Second, the identification between a complete PDU and a fragmented PDU is based on the fragment amount value at the receiver end. If the value is greater than zero the receiver handles the PDU as a fragment. In case that the fragment value equals zero, the PDU is considered to be complete. Therefore, it is crucial that the fragment amount value is without exception set to zero for all messages sent as complete. The final field of the RMP header is the fragment number field. The fragment number field is 8-bit long and functions as order number for the fragments.

5.8 Encoding

The RMP header uses the fixed length binary encoded fields. The payload portion of the RMP packet is encoded as fields using TLV encoding. In TLV, the first two types and length fields identify the type and the length of the data carried in the following value field. The length of the first two fields are fixed at one byte for the type field and two bytes for the length. The length of the value field varies depending on the data being carried. The TLV encoding allows the decoder to ignore all unknown data types without causing the system to halt. The RMP encoding is based on full bytes and uses the little-endian style. In other words, the least significant bit of the least significant byte is sent first.

5.9 Application Messages

Application messages contain the EAP authentication messages and the remote monitoring data messages. EAP authentication messages follow the format specified in RFC 3748 [40]. This section specifies only the remote monitoring data message sent between the analyzer application and the RMS application. The application messages are encapsulated into the RMP packet, which also contains some information relevant for the application. Table 5 describes the RMP header values that have significance for the application. An application message may contain one, many, or no mandatory TLV data field and optional fields depending on the configuration of the analyzer. The message type value indicates whether a message is a request or a response. Even type values refer to a request and odd type values to a response. A request message having an even message type always receives a response from the counterpart. On the other hand, a response message shall not get any reply and it usually ends the message exchange procedure. Table 5 specifies the RMP header fields that RMP must either pass to the application or provide a mean to set their value. In addition, the application must be able to get and set the IP-address and UDP port values from/to RMP. Relevance of the RMP header fields for the applications

Table 5. Relevance of the RMP header fields for applications

Data field name	Relevance for the application
Device category	RMP must provide a method to set and get this value for both applications
Flags	RMP must pass the action flag value to the client applications and provide a method for the server application to set the action flag. RMP must provide a method to set the RMP mode flag for both applications
Client device ID	RMP must provide a method to set and get this value for both applications
Message type	RMP must provide a method to set and get this value for both applications

The data fields of each application message type are illustrated in the following figures. Each figure illustrates the request response message pair of the

specified procedure with the corresponding message type information. The field description contains the field name and a definition of the data carried in the value field. The definitions include the length of the value field, if known, and the relevance-information whether the field is mandatory or optional for the message. If the relevance-information is not mentioned, the field is thought to be mandatory. A mandatory field refers to the field that the message must always contain, whereas the optional field refers to the field that may be omitted from the message.

5.9.1 Authentication

The authentication message formats and procedure shall follow the EAP specification [40]. Any of the EAP-compliant authentication method can be used. The authentication procedure shall occur between the requesting client device and the Authentication Authorization Accounting (AAA) capable authentication server. The RMC gateway shall function as a network access server in the process. The authentication process is discussed in the Section 5.11.1.

5.9.2 PMTU Discovery

The RMP client sends the PMTU request message to RMS after the analyzer application has initiated the PMTU discovery procedure. The procedure adjusts the first request message size to 70% of the currently configured PMTU. If the request goes through, RMS sends the response message back to the client. If the response is not received within the retransmission period, the client shall reduce the PMTU request message size to 70% of the previously sent message size. The reduction shall continue until the reception of the first PMTU response message or until the PMTU request message data length is less than 568 bytes. The 568 derives from the 576 bytes, which is the recommendation for IPv4 host as a minimum MTU size. The 8 bytes difference is the UDP header size. The PMTU response message contains only the RMP header (Figure 15). The message length field in the header indicates the size of the payload carried

in the successfully received PMTU request message. The client shall update the PMTU parameter value based on the message-data-length value of the first received PMTU response message. Once the PMTU is successfully discovered, RMP shall discontinue the sending of PMTU requests and indicate to the application the termination of the PMTU discovery procedure.

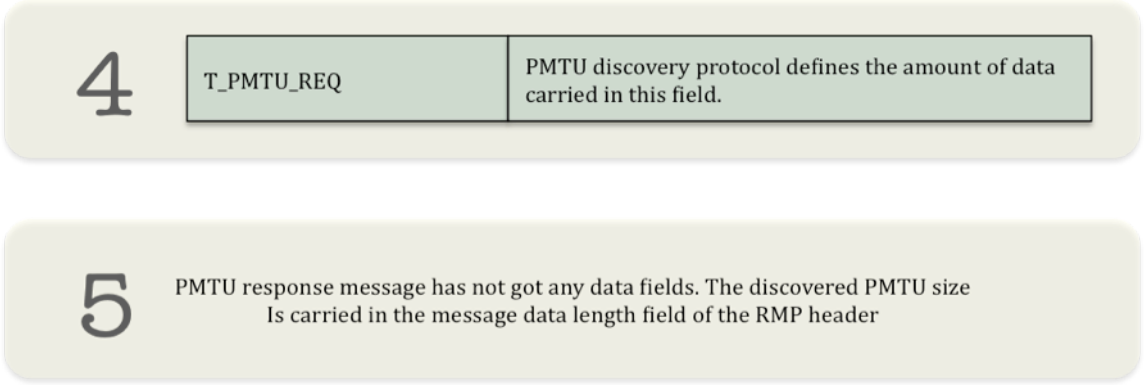

Figure 15. Only the PMTU request message has one data field and the response message contains only the RMP header

5.9.3 Registration

The analyzer sends the registration request (type 6) message to the remote monitoring server after a successful authentication procedure (Figure 16). The client sends the device ID as registration key to the RMS. RMS acknowledges the registration request, if the equipment ID turns out to be valid. On the other hand, RMS shall reply with a negative acknowledgement, if it does not accept the registration. The analyzer shall continue the registration procedure until a positive acknowledgement from the RMS is received. Once registered successfully, the analyzer may transition to the configuration state.

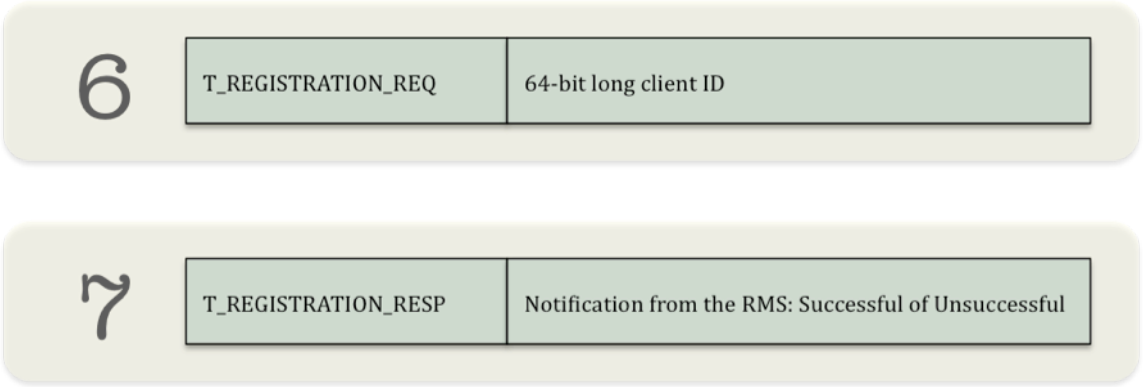

Figure 16. Both registration request and response messages both have one mandatory data field

5.9.4 Configuration

The configuration procedure includes three messages: configuration request (type 8), configuration (type 10), and the configuration response (type 11). The configuration request message is sent from the analyzer to the RMS if the analyzer is lacking the configuration or a reception of a received action response message so indicates. The server replies with the configuration message including the necessary data fields for the configuration parameters. Figure 17 shows the possible parameters of the messages. After the analyzer has received the configuration response and has successfully performed the configuration, it indicates the success of the configuration by sending the configuration response message back to the RMS. The Configuration request message does not include any mandatory data fields, only the RMP header is sent. The device ID and the message type shall invoke the configuration procedure at the RMS application.

8 Configuration request message does not contain a data field.

10

T_AUTH_G	The address of the authentication gateway with whom the analyzer shall authenticate.
T_AUTH_PERIOD	The length of the valid authentication period. The analyzer shall re authenticate when this period expires.
T_RMS_ADD	The address of the active RMS with whom the analyzer shall register and to whom the analyzer shall report
T_PMTU	The size of the current path MTU
T_FRAG_ACK	Defines weather the fragments shall be acknowledge or not by the RMP
T_DATE_TIME	Date and time information
T_STAT_REP_TIME	The time of day when the analyzer shall send the statistics report
T_FAULT_FILT	Defines which faults the analyzer shall or shall not send to RMS
T_STATUS_LOG_FILT	A list of fault classes that shall be send with status before and after information
T_RESEND_INTVAL	Defines the length of the period after which the analyzer shall resend the message if no response from the RMS have received

11

T_CONFIG_RESULT	Notification from the RMS: Successful of Unsuccessful

Figure 17. Configuration procedure includes three messages: The request from the client, configuration from the RMS, and again response from the client

5.9.5 Fault Occurred

The fault occurred (type 12) message is sent from the equipment analyzer to the remote monitoring server after each occurred fault event at the equipment. The request message contains three mandatory fields: fault code, time stamp,

and the status fields. The T_FAULT_CODE is equipment type-dependent 64-bit long value that identifies, localizes, categorizes, and specifies the state of the fault event. The T_TIME_STAMP is as well 64-bit long value indicating the exact time of the fault event (Figure 18). The T_STAUTUS field is a 20-byte long value, which indicates the status of the equipment at the moment of the fault occurrence. In addition, the fault occurred message may include optional status fields, which can be further categorized in two groups: T_STATUS_AFTER fields, and T_STATUS_BEFORE fields. The T_STATUS_AFTER fields indicate the status of the equipment after the fault occurrence. The time interval of the T_STATUS_AFTER fields as well as the number of those fields may vary depending on the analyzer configuration. The T_STATUS_BEFORE fields have exactly the same functionality as the status after fields but it gives the status of the equipment before the fault occurrence. There may be as many before and after fields in the message as configured in the analyzer. RMS shall send the response (type 13) message to the analyzer if the request message was successfully received. In case of an unsuccessful reception of the message, RMS shall silently ignore the received message and not send any response to the analyzer.

12

T_FAULT_CODE	A code that identifies the fault event
T_TIME_STAMP	Time when the fault event occurred
T_STATUS	The status of the equipment at T_TIME_STAMP
T_STATUS_B	Status field indicating the status of the equipment before the occurrence of the fault event. An optional field
T_STATUS_A	Status field indicating the status of the equipment after the occurrence of the fault event. An optional field

13

T_FAULT_CODE	A copy of the T_FAULT_CODE field of the request message
T_TIME_STAMP	A copy of the T_TIME_STAMP field of the request message

Figure 18. The fault occurred request message carries identification information of the fault. The response indicates the client that the fault was successfully received

5.9.6 Fault update

The fault update message is sent from the equipment analyzer to the remote monitoring server when a change in a state of the fault occurs.

The fault update message has two mandatory data fields the T_FAULT_CODE and the T_TIME_STAMP (Figure 19). T_FAULT_CODE maps to the existing fault and indicates the change in it. T_TIME_STAMP defines the moment when the change occurred.

RMS shall send the response (type 15) message to the analyzer if the request message was successfully received. In case of an unsuccessful reception of the message, RMS shall silently ignore the received message and not send any response to the analyzer.

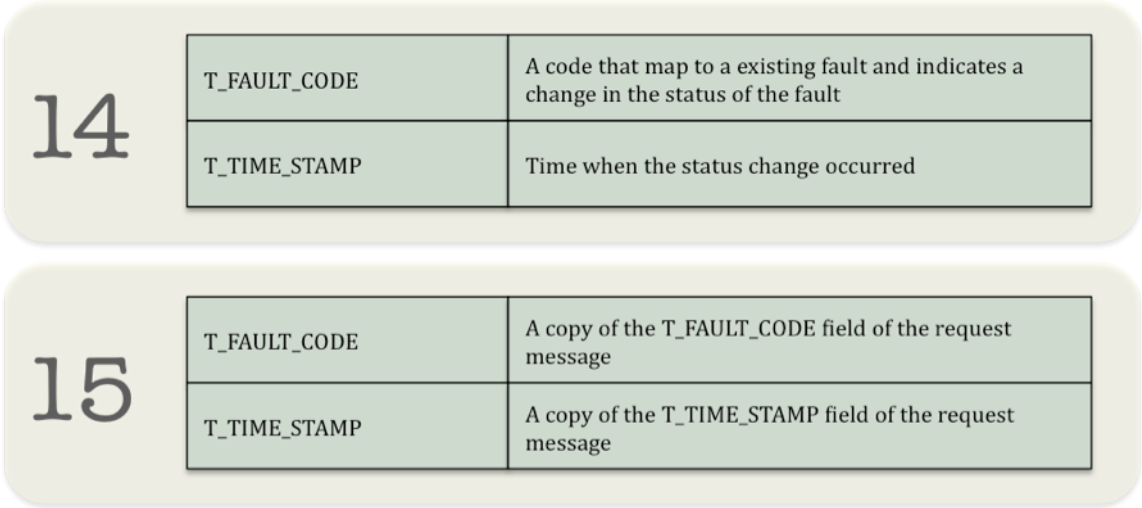

Figure 19. Fault update messages have two data fields

5.9.7 Statistics report

The statistics report (type 16) message is sent from the equipment analyzer to the remote monitoring server at a certain time of the day, which is defined in the configuration of the analyzer. The statistics report contains the T_STATUS data field and as many additional data fields (T_COUNTER and T_MEASURE) for the counter- and measurement data as defined in the analyzer configuration (Figure 20). RMS shall send the response (type 17) message to the analyzer if the request message was successfully received. In case of an unsuccessful reception of the message, RMS shall silently ignore the received message and not send any response to the analyzer.

16

T_STATUS	The status of the equipment at T_TIME_STAMP
T_COUNTER	Counter values
T_MEASURE	Measurement values

17

T_STAT_SUCCESS	Indication of successful receptions

Figure 20. The statistics request message carries information of the client status and statistics. The response only confirms a successful reception

5.9.8 Action messages

The action request (type 18) message is sent to the RMS when the analyzer receives an AF indication from the RMP. The action request message is sent only if no existing message exchange procedure occurs (Figure 21). The RMS replies with the action response (type 19) message. RMS may only demand one action at the time form the analyzer. RMS keeps setting the AF bit to 1 until no more actions from the analyzer are required. The analyzer shall perform the demanded procedures defined in the action message. After this procedure, the analyzer shall re-transmit the action request message if the AF bit of the last message from RMS is still set to 1. The action request message does not include any mandatory data fields, only the RMP header is sent. The device ID and the message type shall initiate the action response procedure in the RMS application.

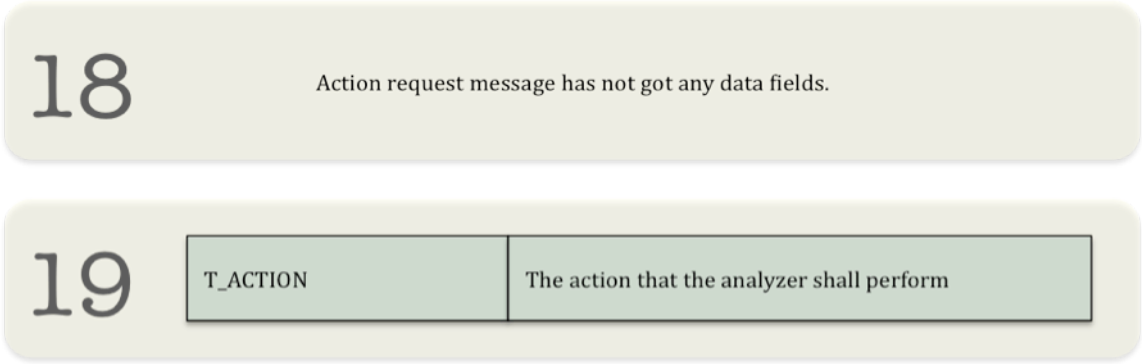

Figure 21. The action response carries the procedure information that the client shall perform. The request only invokes the server

5.9.9 BOM

The analyzer sends the BOM (type 21) message to RMS when requested in the action message (Figure 22). The BOM message contains the necessary data fields required for the BOM data. The BOM response message confirms a successfully reception at the server end for the client.

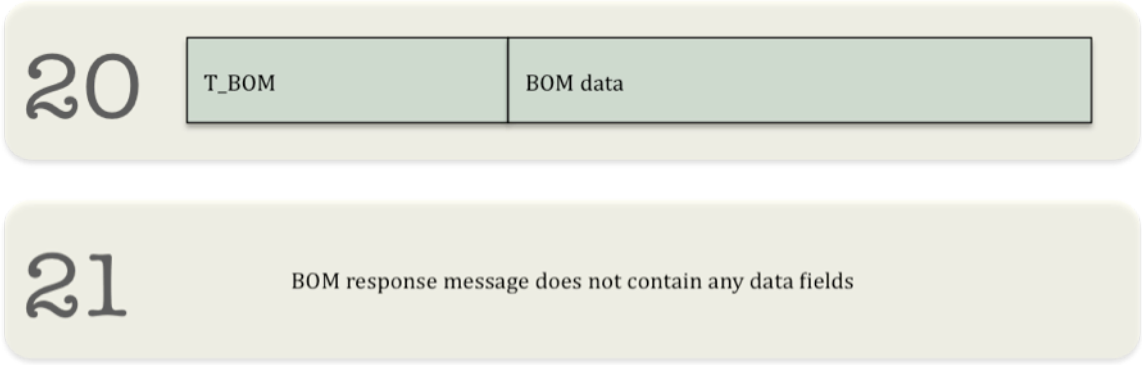

Figure 22. The BOM procedure is used in conjunction with the action procedure to query equipment data from the client

5.10 Functionality on RMP level

RMP offers fragmentation services as default feature for all the application messages. Additionally, RMP can be configured to function in unacknowledged or acknowledged mode. In the unacknowledged mode, RMP does not offer any kind of sequence controlling or retransmission services for the PDUs.

5.10.1 Acknowledged Mode and Retransmissions

RMP can be configured to acknowledge the data PDUs. The acknowledgement is done for each data PDU having the RACK bit set (section 5.7.2). This also applies for message fragments if the complete message is set to be acknowledged by the application. The acknowledgement is done by sending the ackPDU back to the sender. RMP can only send positive acknowledgements. In ackPDU all the header fields are copied from the header of the data PDU being acknowledged and only the PT flag is set from the flag-octet. Message ID is used to map the data PDU and the corresponding ackPDU at the sender. The sender can remove the data PDU from its buffer only if a positive acknowledgement from the destination is received. That is, the sender shall re-transmit the data PDU in pre-defined intervals until the reception of the positive acknowledgement.

The following figures illustrate the RMP PDU exchange procedures in both RMP modes.

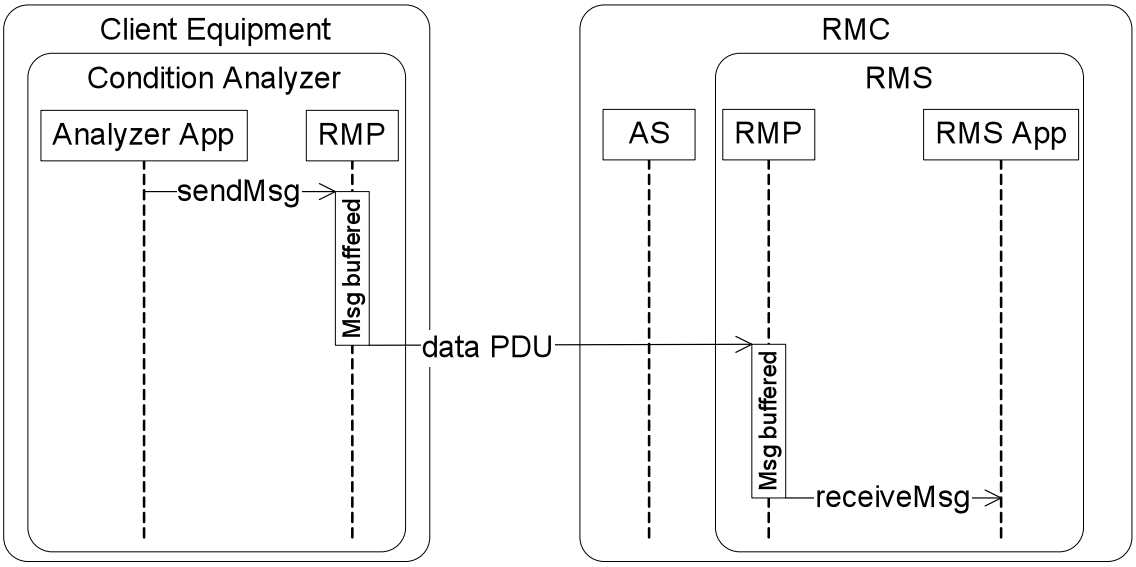

Figure 23. RMP unacknowledged mode - successful PDU transmission

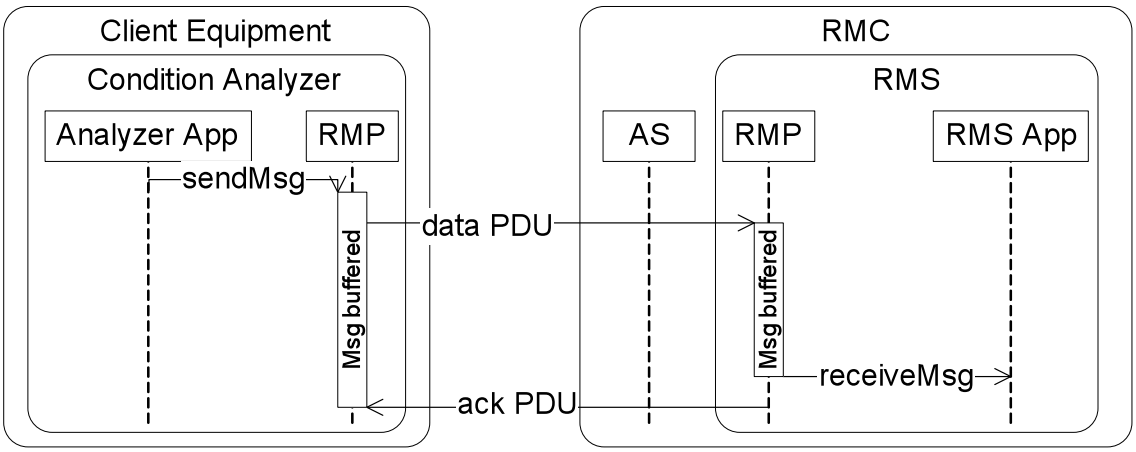

Figure 24. RMP acknowledged mode - successful PDU transmission

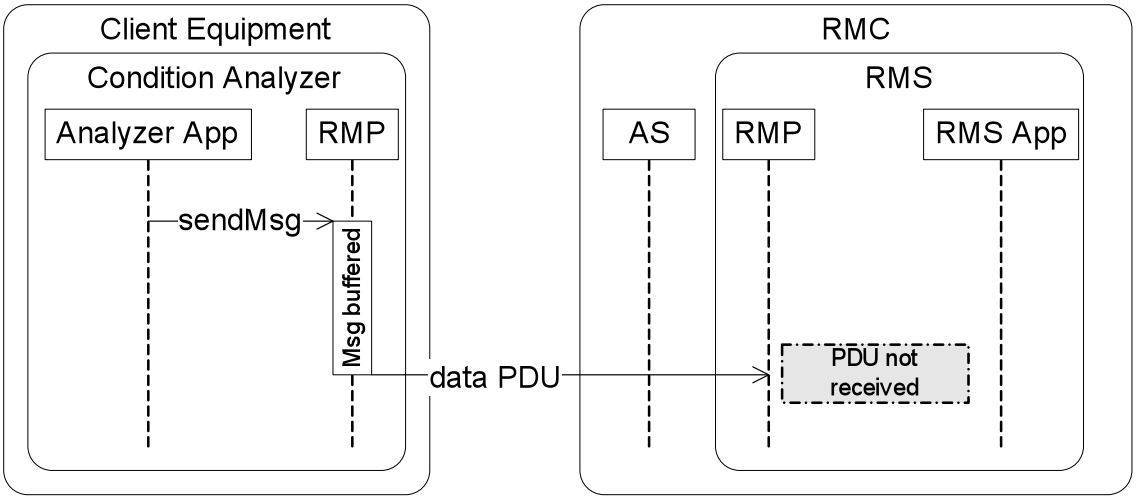

Figure 25. RMP unacknowledged mode - error in PDU transmission

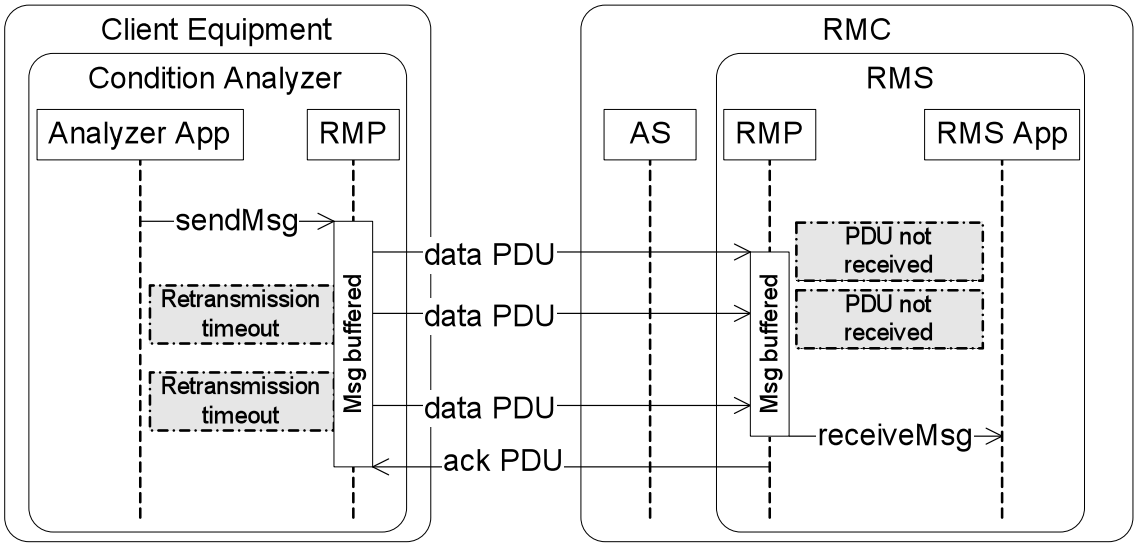

Figure 26. RMP acknowledged mode - error in PDU transmission

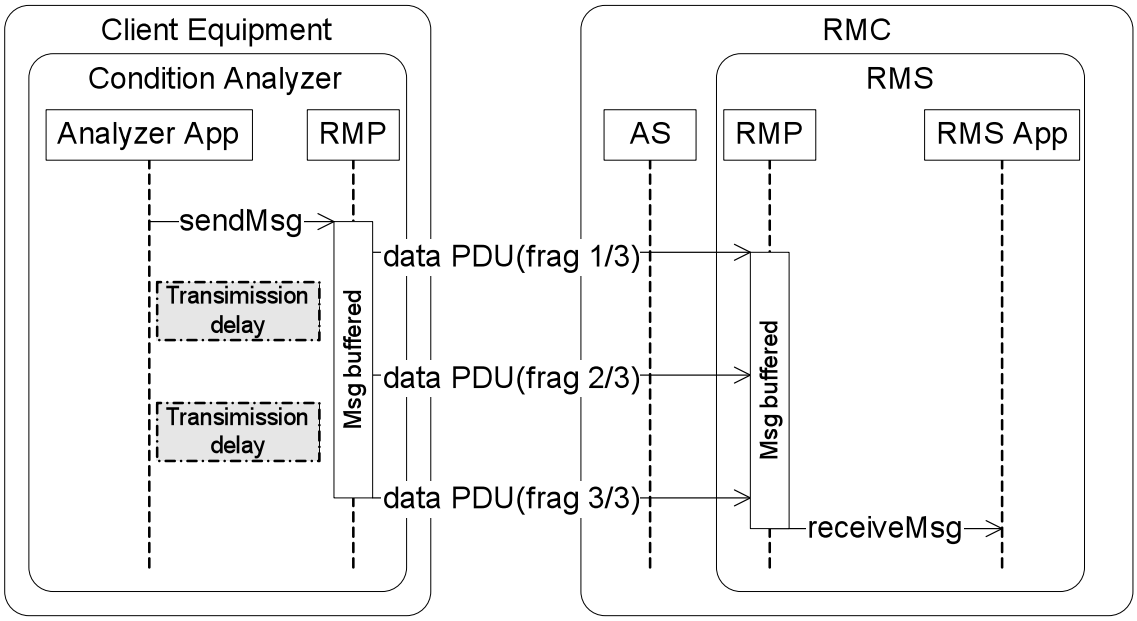

Figure 27. RMP unacknowledged mode - fragmented message

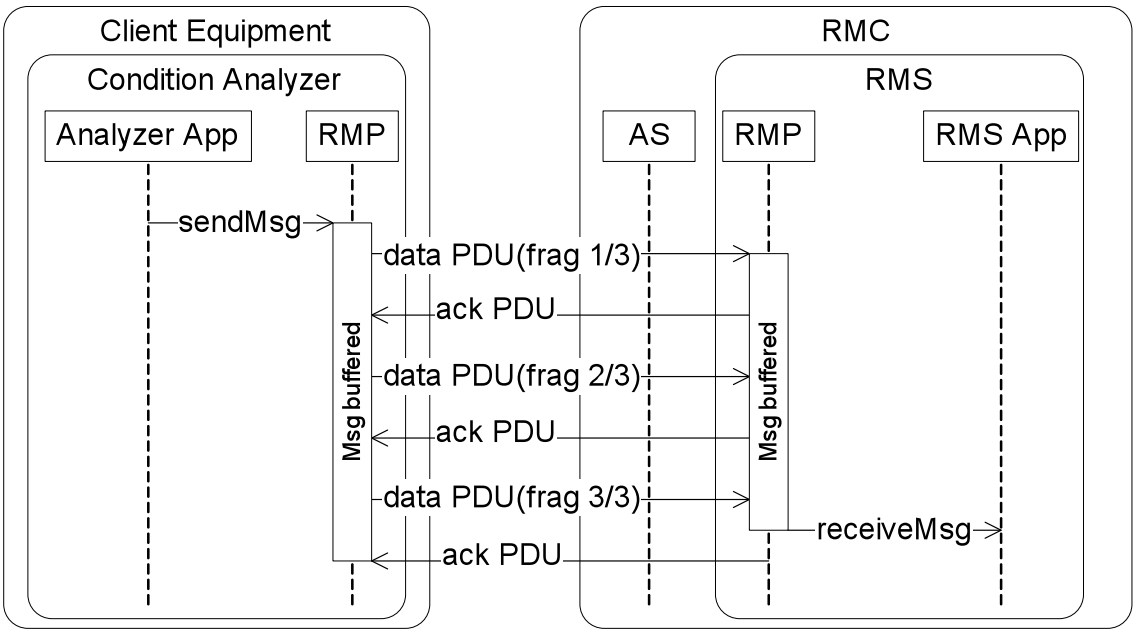

Figure 28. RMP acknowledged mode - fragmented message

5.10.2 Fragmentation / Re-assembly

RMP shall deliver the application messages even if their size oversteps the configured path MTU size. Therefore, RMP fragments the messages at the transmitter end and reassembles the fragments at the receiver end respectively. The transmission of the fragments can be configured to be reliable or unreliable.

Once a message is composed and a send command received from the analyzer application, RMP shall calculate the complete length, including header, of the message. The length is then set into message length field in the header. Fragmentation is needed if the length oversteps the PMTU size. The message is split into number of fragments each containing the RMP header and data fields from the original message. The fragmentation can occur only at the junction of two data fields so that each fragment message only contain complete data fields. The total amount of fragments is calculated and each fragment is numbered (Figure 29). The fragmentation information is added into the header of each fragment respectively. The message length field in the fragmented message header indicates the length of the complete message, not the length of a message fragment.

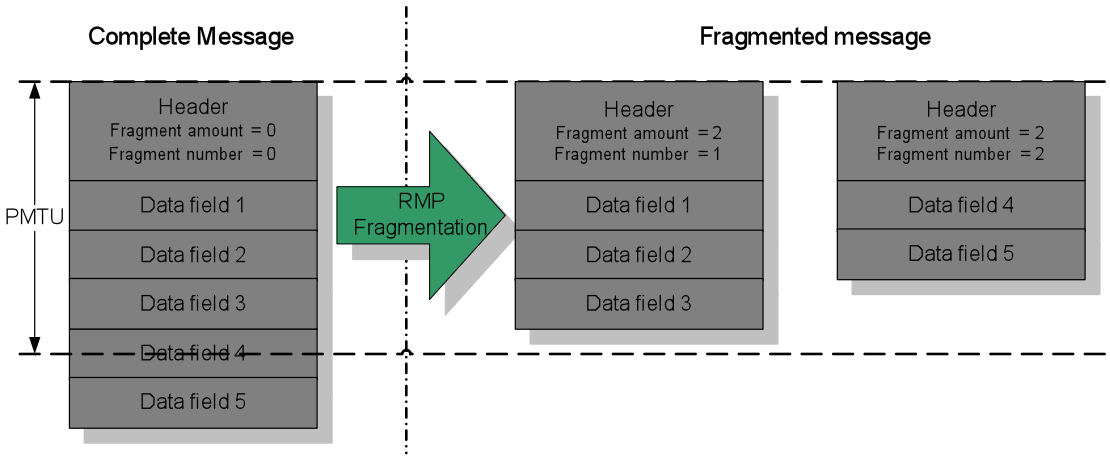

Figure 29. RMP fragmentation and fragment numbering

The re-assembling process at the receiver initiates only if the fragment amount field in the message header is greater than zero. First, the receiver verifies whether there already exists a buffer for the message or not. If not, the receiver consults the message length field from the header and allocates the needed memory for the complete message. Second, the fragment is saved into the buffer and a timer is started. The following fragments are added into the buffer according to their fragment number. Once the final fragment is received (fragment number = fragment amount), the message is completed and passed to the analyzer application. After this, the buffer is freed and the timer stopped. The buffer shall also be freed and the message discharged, if the message is not completed within a pre-defined period of time. That is, the timer timeout occurs.

In acknowledged transmission mode, each fragment is expected to be acknowledged by the receiver. The sender can set the RACK bit (section 5.7.2) to 1 in order to indicate the usage of the acknowledged transmission mode. The acknowledged transmission starts from the first fragment and proceeds to the next fragment only if a positive acknowledgement from the receiver is received. Once all the fragments are sent, the buffer is freed.

The use of the unacknowledged transmission mode is indicated by clearing the RACK bit (section 5.7.2) at the sender. In this mode RMP sends all the fragments, starting from the first one, without waiting for acknowledgement from

the receiver. The transmission in this mode shall be done with moderate bitrates due to the slow GPRS timeslot allocation. Once the fragments are sent, the buffer can be freed.

5.11 Message Exchange Sequences

This paragraph illustrates the messages exchanges procedures between the equipment analyzer and the remote monitoring server. The illustrations include five different cases. First, the authentication, registration, configuration and re-direction of the client are shown in a single sequence diagram. Second, a successful and unsuccessful client-oriented message exchange is illustrated in two separated diagrams. These two sequence diagrams generally describes the message exchange functionality for all the request/response procedures. The duplicated message detection procedure is described subsequently and, finally, the action flag procedure is shown in a dedicated diagram.

5.11.1 Authentication, Registration, Configuration and Re-Direction of the Client

The authentication process in RMP must successfully be able to authenticate the entities, the RMP client, and the remote monitoring centre before the RMP client can access the main RMC network. Therefore, authentication shall occur already at the RMC network boarder with the gateway router. The client and the RMC gateway shall mutually authenticate according to the EAP procedure. First, the client takes the role of the authenticator and authenticates the gateway (Figure 30). After that, the roles are changed and the gateway authenticates the client. The authentication procedure may include an AAA server if centralized authentication management is required. All the equipment analyzers shall be preconfigured with the IP information of the default RMC.

The re-transmission and the message duplication control of the authentication process shall follow the EAP specification [40]. However, the re-transmission shall continue only a configurable time after which the RMP shall transition to the disabled state.

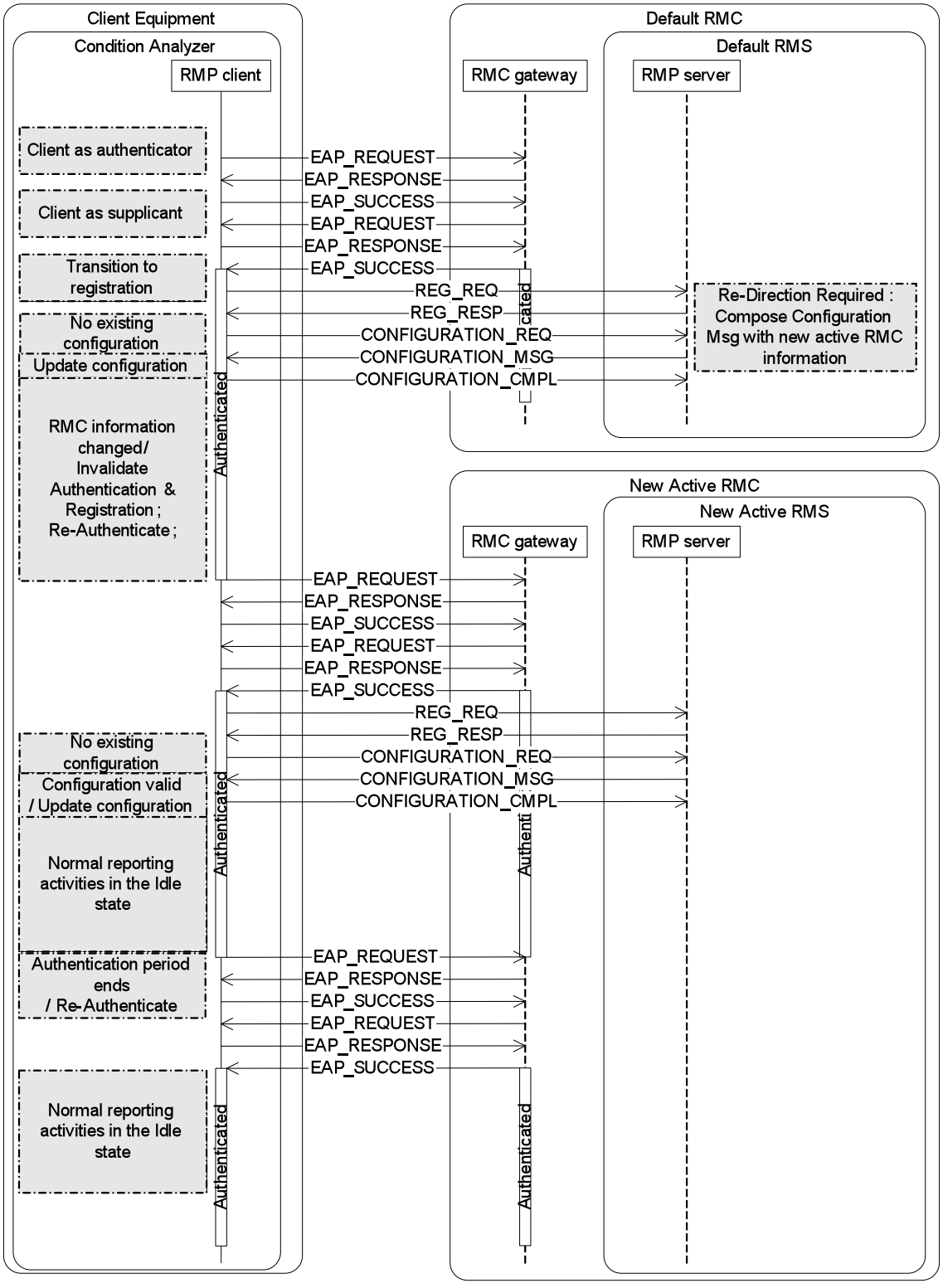

Figure 30. Authentication, registration, configuration, and re-direction procedure

Once the client has been granted access to the default RMC network, the client shall request registration from the default RMS. RMS may either accept or

reject the request of the analyzer. The equipment ID is used to determine whether the requesting analyzer is valid or not. RMS replies to the valid client with a registration accepted message. From now on, the client verifies whether it has a valid configuration or not and continues with the configuration procedure or starts the reporting activity accordingly.

Optionally, the RMS may indicate the client by setting the AF bit in the registration response message, to send the action request to RMS. This may occur if RMS wants to re-direct the client to another RMC. After RMS receives the action request, it sends the action response with an action field of configuration request back to the client. This results in the analyzer sending a configuration request to the RMS. RMS may now send the configuration response with new IP information of the active RMC to whom the client should register with. The client shall update its configuration with this information and re-authenticate with the new RMC.

The re-direction feature dynamically allows the RMSs to perform load balancing in case of too many requests from the clients. The re-direction feature also allows having either one centralized RMC or many localized RMCs in the remote monitoring system.

5.11.2 Downloading Configuration of the Condition Analyzer

The RMP client shall send a configuration request to the RMS after a successful authentication and registration procedures with RMC (Figure 30). Additionally, RMS may launch the configuration request from the client by defining it in the action message. RMS shall reply to the configuration request of the client with the configuration message. The configuration message may contain either the equipment specific configuration parameters or a default configuration parameter for the analyzer. When the client receives the configuration message, it shall acknowledge the message with a configuration completed message, and change its configuration according to the received data. After successful configuration download, the client may transition to the idle state. In the idle state, the client has five different use cases: sending a fault, updating

the fault status, sending statistics, sending an action request, and uploading the equipment data.

5.11.3 Sending a Fault

The client shall log each fault occurrence detected in the equipment into a local fault log. The fault log shall contain the following information for each fault occurrence: 64-bit long fault code, 64-bit long timestamp, the equipment's status information at the time of the fault occurrence, and the message exchange state. The log may also contain additional information concerning the state of the equipment. This is, a configurable number of the equipment's status information fields before and after the fault occurrence. The timer interval between the status fields can also be configured. For example, the analyzer can be configured to record 100 status fields after, and 20 fields before the fault occurrence with the interval of 100 ms. This would show the equipment's status 10 seconds after and 2 seconds before the fault occurrence.

The RMP client sends the fault occurred message to the active RMS after each new fault occurrence (Figure 31, p. 73). The message contains the fault code, timestamp and the status fields. Optionally, if configured so, the client may include the additional status fields, described above, into the message. The message exchange state in the log shall be set to pending state after the message is successfully sent. Once the client receives the response message from the RMS, the state field in the fault log shall be updated accordingly.

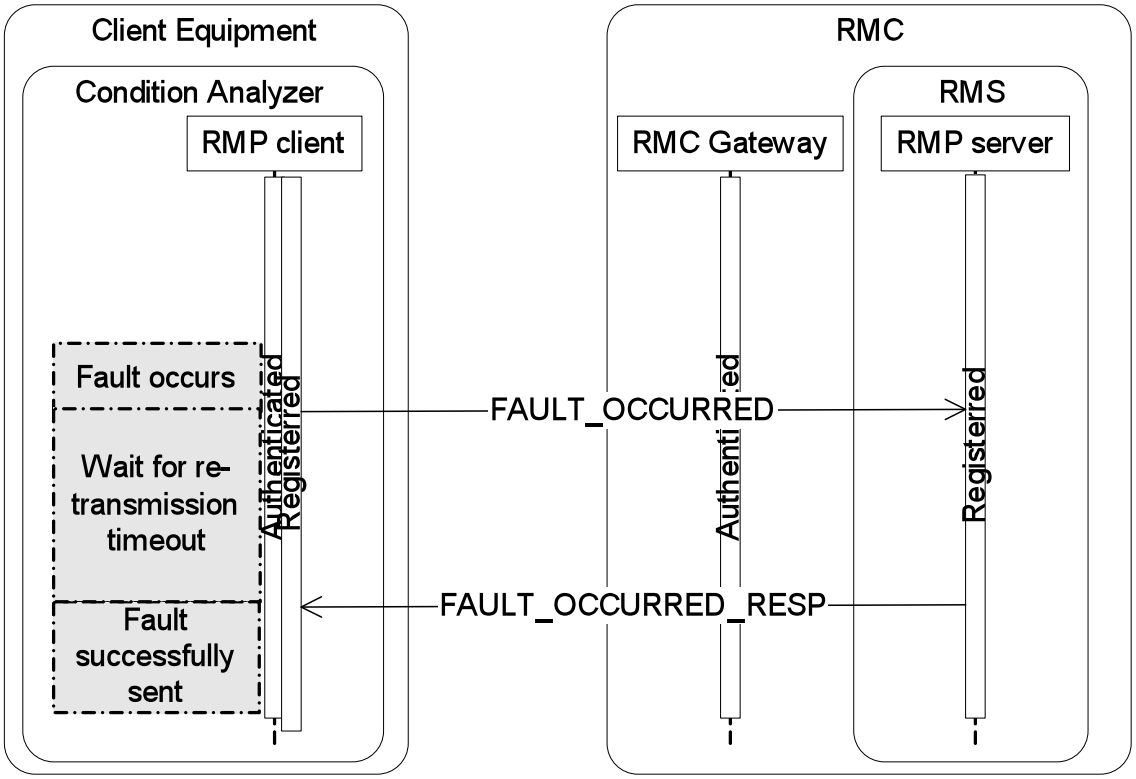

Figure 31. Sending the fault occurred message

The RMP client shall re-transmit the fault occurred message after a configurable period if the message exchange state is pending and no ACK is received from the RMS (Figure 32, p. 74). The re-transmission shall continue until the RMS has successfully acknowledged the message.

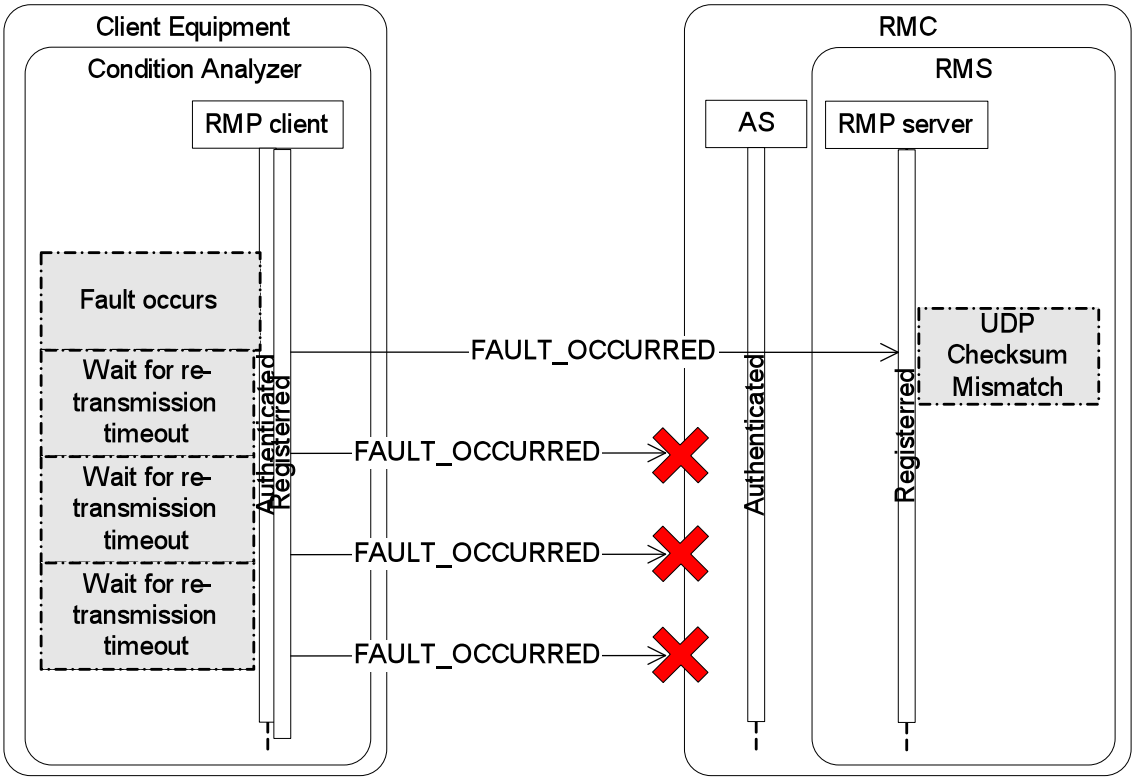

Figure 32. Error in transmission while sending the fault occurred message

Duplicated messages shall be silently discharged at RMS, if received (Figure 33).

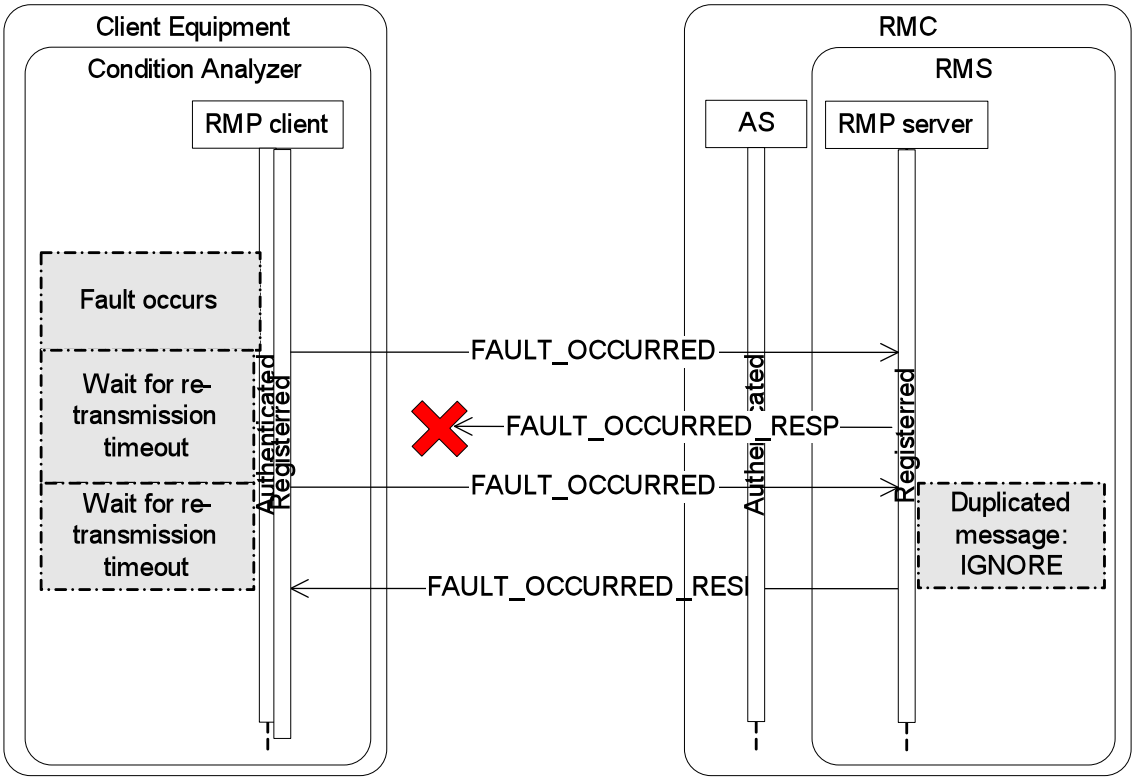

Figure 33. Message duplication

5.11.4 Updating a Fault Status

A fault update message is sent to the RMS when a change in the status of an existing fault occurs. This kind of change may occur, i.e., when the equipment recovers resulting that the fault disappears. The fault update message contains only the fault code indicating the change and a timestamp. The fault update procedure includes the re-transmission scenarios described in Section 5.10.3.

5.11.5 Sending Statistics

The RMP client shall send the counter or measurement data of the equipment to the RMS. This transmission shall occur at regular intervals at a configurable time of day. A recommended interval is once a day. The client shall perform, if required, the retransmission scenarios described in Section 5.10.3.

5.11.6 Sending the Action Request

RMS sets the AF bit for all response messages to the clients that have unperformed action in their action list. A client that receives a message having the AF set to one shall send the action request message back to the RMS (Figure 34, p. 76). However, the action message shall not be sent if another message exchange procedure is going on. That is, if, for example, the client receives a configuration message with AF set to one, it shall finish the configuration procedure by sending the configuration completed message before the action request message is sent to the RMS. RMS may only request one action to be performed at the time.

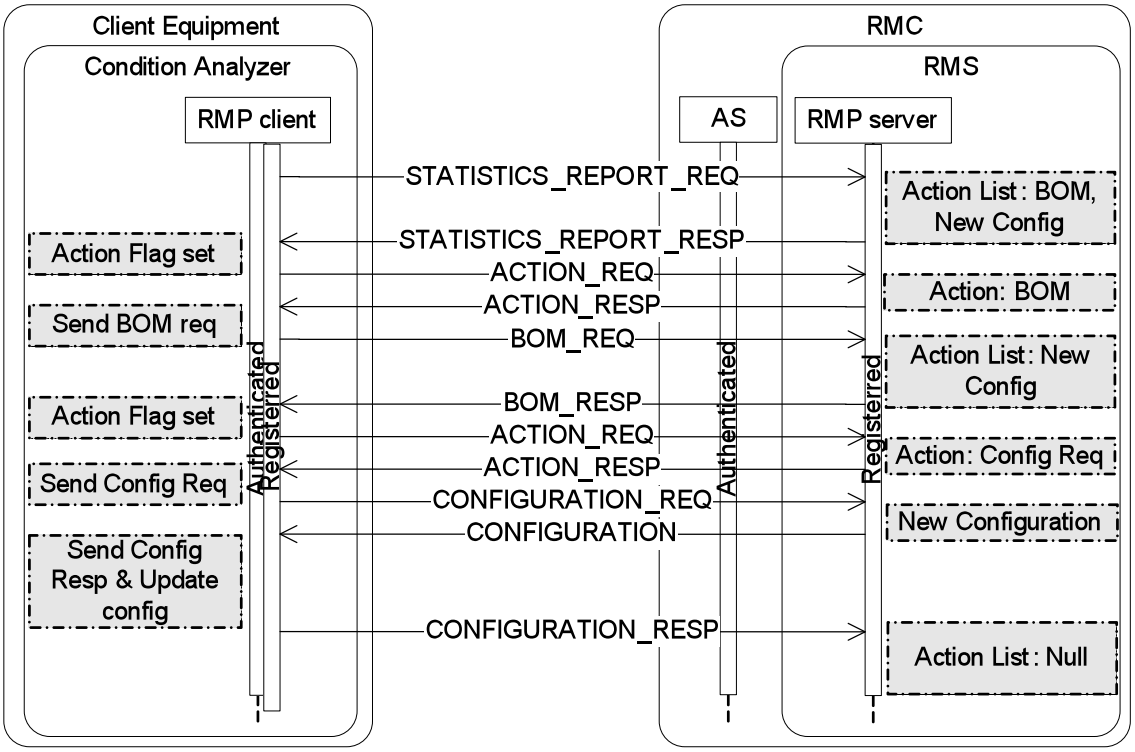

Figure 34. Action message procedure

5.11.7 Upload equipment data

The RMP client shall send the BOM message to the RMS when requested in the action response message (Figure 34).

5.12 RMP API

The RMP Application Programming Interface offers three different interfaces for applications to use RMP services. The RMPClient and RMPServer interfaces specify the method for sending and receiving of data, configuring the RMP, and invoking the PMTU discovery procedure. The client side includes both blocking and non-blocking methods whereas the server side has only the non-blocking methods. The RMPMessage interface defines various methods to compose and read application messages. After composing an application message through the message interface, the application passes the reference of the created RMPmessage-object to the RMP layer using the methods of the RMPclient or server APIs. The details information of RMP API in javadoc-format is included with this thesis as Appendix I.

6 Reference implementation

6.1 Introduction

The purpose of the RMP reference implementation was to demonstrate some of the key functions of RMP specification. This included the complete RMP layer with fragmentation and re-assembling methods, re-transmission and RMP acknowledgments as well as the API implementations. The application layer was partly demonstrated with registration, configuration and fault occurred procedures implementations. The implementations were performed as a single thread solution for both the server and client applications. In addition, the action procedure was also displayed. The testing was performed over a local area network and a public mobile Internet-link.

6.2 Materials and Methods

The programming of the RMP-protocol was implemented in the Eclipse development environment using JAVA. The network setup comprised of two laptop computers where one of the computers was connected to the Internet

through a GPRS modem and the other was in a private LAN. The LAN had a connection to the Internet through an ADSL modem performing NAPT translation.

6.3 JAVA Implementation

The RMP implementation of JAVA comprises of six classes for the RMP layer and simple application demo classes for the client and the server. The RMP, RMPClient, and RMPServer classes include all the protocol functionalities of RMP. The RMP class is a parent class that contains the rules and functions that are equal for the client and the server. These are: sending and receiving to/from UDP, fragmentation protocol, retransmission rules, and sending and receiving acknowledgment packets. In addition, the RMPclass declares all the shared fields such as delays and re-transmission parameters, UDP socket and message objects. The RMPClient and RMPServer classes extend the RMP-class inheriting all its protected methods and fields. The child class contains the endpoint specific fields and receiving protocol rules.

The RMPPacket class contains all the RMP header fields and the application message payload field as a byte array. The class also provides methods to access all the fields. The ApplicationMessage class contains the fields and access methods for the application message data. However, the proper TLV-encoded was not implemented and the application message data contained only String-data driven into byte[]-array.

RMP services were divided in two classes where one is handling the encapsulation service for RMPPackets converting them to/from DatagramPacket. The re-assembling and fragmentation services are combined into one class. The FragmentationRe-assembling class takes RMPPackets as constructor parameters and performs fragmentation and re-assembling services for them. The class also has a RMPPacket array field where the fragments can be stored for sending or receiving purposes.

The ClientAppDemo class contains main method for the client application and is able to send and receive application messages to/from RMP client object using

the methods specified in the RMPClientAPI document. In addition, the client application initiates an object of the ApplicationMessageAPI-interface, which is used to compose, send and receive application messages. The server side has a similar Demo class with receiving and sending function, and some message handling protocol rules.

The whole implementation was documented during the programming phase using javadoc-format. The javadoc-documentation as well as the complete source code of the implementation is attached with this thesis as Appendix II: javadoc, and Appendix III: source code. The detailed information of the implementation project can be found in those documents.

6.4 Results

All the functions of the RMP layer were successfully demonstrated. The following lists these functions:

- Fragmentation
- Re-assembling
- RMP Acknowledged mode
- RMP Unacknowledged mode
- Re-transmission
- RMP PDU structure

Some of the Application level specifications were demonstrated. The following lists the RMP application level specification that were shown in the demonstration:

- Registration procedure
- Configuration procedure
- Fault reporting
- Action procedure
- Application message structure (partly)

6.5 Summary

Even though the reference implementation did not display all the features of the RMP specification, it successfully demonstrated the functionality of the RMP design. It also demonstrated the feasibility of the protocol in the required monitoring system. The next phase of this implementation would be a properly planned testing process for a C implementation of a real embedded environment.

7 Conclusion

This thesis proposed a solution for a generic remote monitoring protocol in condition monitoring of mechatronics devices in service business. The specification of the RMP protocol determines the telecommunication procedures between client devices and the remote monitoring centre. The RMP design combines some existing and new specifications. In addition, the RMP specification was implemented in a JAVA environment to demonstrate some of the functionality of the protocol. The following sections summarise the main requirements of RMP and discuss how the requirements were met. Furthermore, this chapter presents some future steps concerning the RMP protocol.

7.1.1 Meeting the Requirements

The hosts in RMP communication were required to mutually authenticate with the active RMC gateway. The RMP specification includes the EAP authentication procedure performed in two directions to meet the mutual authentication requirement. The authentication procedure initiates automatically when the RMP client is enabled. In addition, the client performs a re-authentication procedure in configurable intervals. For the disabling requirement, RMP specifies a configurable retry period after which the client disables itself, if the authentication continues failing.

RMP specifies a registration procedure, which allows the client to register with the active RMS. The client launches the registration procedure automatically after the mutual EAP authentication has been successfully performed. RMS discharges all the messages, except registration messages, from unregistered clients.

RMP uses client-originated transactions, because of the network architecture consisting of three address spaces and NAT translations. Therefore, RMP specifies an action procedure to fully meet the configuration download and the BOM request requirements. RMC personnel can use the action procedure to indicate the client to send either the BOM or the configuration request. In addition, the action procedure specification allows easy implementations of new actions for future purposes.

The RMP client is able to download a configuration from the RMS using the configuration request. RMS sends a configuration to the client in the configuration response message. The client requests the configuration whenever it is needed. RMS is able to indicate the client about a new configuration through the action procedure.

RMP specifies three message exchange procedures for fault and statistics reporting. For each new fault occurrence in the device, the client uses the fault occurred request message to report the event. The fault occurred request includes the fault code, timestamp, the status of the device. Additionally, the message can include more detailed status information. Fault update request procedures allows the client to inform the RMS when a change in fault status occurs. The client can performs daily statistics reporting through the statistics procedure. Finally, using the action procedure, RMS can request the client to send the equipment data message (BOM) to RMS.

In addition, RMP was specified to meet some non-functional requirements caused by the network architecture or some other factors. RMP offers completely connectionless communication using the UDP/IP stack for data transmission. However, RMP contains the necessary mechanisms for reliable data transfer between the client and the server. Furthermore, RMP is able to

reliably send messages confronting the maximum UDP message size by offering the fragmentation and re-assembling services. The communication between the entities includes the minimum amount of control messaging to avoid purposeless bandwidth utilization. RMP can be used with both IP versions over several IP address spaces and dynamic NAT. The configuration of the client device is automated allowing the client to download configuration parameters from the Remote Monitoring Server. Finally, RMP specifies an optional path MTU discovery procedure to guarantee the throughput in case of the network topology change.

7.1.2 Future Perspectives

The Remote Management Protocol is generic in a sense that it is applicable for different monitoring systems. However, the protocol was designed to be implemented on a secure network platform provided by a single network provider. As a result, great care needs to be taken while implementing RMP over a public unsecured network. RMP messages are transmitted as plain text without any encryption mechanism through the network. This includes the authentication messages, as well. The implementation of RMP over a unsecured network without applying a proper encryption mechanism would result in a security risk. The development of an efficient and yet secure enough encryption method for RMP would also expand the use of RMP over public unsecured networks. Despite the JAVA reference implementation, the RMP design should undergo an extensive testing process where all of its functions and use cases would be tested. As RMP is mostly to be used in embedded environment, the testing should be carefully planned by setting the testing requirement accordingly. The C implementation of RMP would be one of the first steps in that process.

7.1.3 Summary

All in all, the outcome of this study, the RMP specification, managed to fulfil the specified requirements for the remote monitoring protocol. The RMP design

offers an efficient, reliable, simple, and scalable solution for remote monitoring systems. Furthermore, the RMP specification defines the generic means for data communication in many fields of remote monitoring.

8 Acknowledgements

This study was carried out for the Research and Development organization of KONE Corporation during the spring 2010. I gratefully acknowledge Aki Karvonen for providing me the opportunity to work in this stimulating project. I also want to thank Aki for his support concerning my future carrier. In addition, a big thanks goes to Tony Wigren, Markku Saarinen, and Reidar Österblom for helping me to get involved with this project.

I owe my deepest gratitude to my supervisor Hannu Rytilä (KONE) for his valuable technical advices and continuous support throughout the whole project. The knowledge his has passed to me is priceless and will, most certainly, be utilized in my future projects. I also want to thank my other supervisor Patric Granholm (TUAS) for his constructive guidance during the thesis work. Poppy Skarli (TUAS) is acknowledged for the excellent language corrections.

My sincere thanks goes to my beloved wife Noora for putting me back on the track when I felt insecure about the thesis. I also want to thank Noora, my sons Pessi and Pieti, and our dog Rontti, for tolerating and understanding me during the times when I was present but not there. Their endless love and support has been the key factor on the way to my graduation. I also give my warm thanks to my father Jouko and my mother Merja for everithing they have provided into my life. Thanks to my father and my brother Pertti for showing me that it is never too late to start studying. My mother-in-law Kaisa also gets my sincere thanks for looking after the children.

9 References

[1] [www-document]. FIMECC OY. Official web site of Finnish Metals and Engineering Competence Cluster. Retrieved March 31, 2010. Available at: http://www.fimecc.com/en/index.php/Main_Page

[2] [www-document]. FIMECC OY. Competitiveness through research. January 22, 2010. Retrieved March 31, 2010. Available at: http://www.fimecc.com/fi/images/b/b9/FIMECC_strateginen_tutkimusagenda2010.pdf

[3] FIMECC OY. Energy and life cycle cost efficient machines – EFFIMA, Program proposal for FIMECC Intelligent Solutions theme. May 25, 2009.

[4] Hällström, Markus A. 2009. *Remote Monitoring Data Utilization in Maintenance for Improved Product Performance.* Tampere University of Technology

[5] Hill, David J. Minsker, Barbara S. Eyal, Amir. 2009. *Real-time Bayesian Anomaly Detection for Environmental Sensor Data*. Published by Elsevier Ltd.

[6] [www-document]. RFC 791: Internet Protocol. 1981, September. Retrieved January 20, 2010. Available at: http://www.ietf.org/rfc/rfc791.txt.

[7] [www-document]. RFC 1883: Internet Protocol, Version 6 (IPv6) Specification. 1995, December. Retrieved January 21, 2010. Available at: http://www.ietf.org/rfc/rfc1883.txt.

[8] Sharp, Robin. 2008. *Principles of Protocol Design*. Berlin, Heidelberg: Springer-Verlag. p. 210-214

[9] [www-document]. Cisco Systems. IPv6 Extension Headers and Consideratons. October, 2006. Retrieved May 5, 2010. Available at: http://www.cisco.com/en/US/technologies/tk648/tk872/technologies_white_paper0900aecd8054d37d_ps6553_Products_White_Paper.html

[10] [www-document]. IANA: Number Resources. Retrieved February 25, 2010. Available at: http://www.iana.org/numbers/

[11] [www-document]. RFC 950: Internet Standard Subnetting Procedure. 1985, August. Retrieved January 21, 2010. Available at: http://www.ietf.org/rfc/rfc950.txt.

[12] [www-document]. RFC 4632: Classless Inter-domain Routing (CIDR) - The Internet Address Assignment and Aggregation Plan. 2006, August. Retrieved January 21, 2010. Available at: http://www.ietf.org/rfc/rfc4632.txt.

[13] [www-document]. RFC 2373: IP Version 6 Addressing Architecture. 1998, July. Retrieved February 24, 2010. Available at: http://www.ietf.org/rfc/rfc2373.txt.

[14] Sharp, Robin. 2008. *Principles of Protocol Design*. Berlin, Heidelberg: Springer-Verlag. p. 215-240

[15] [www-document]. RFC 3022: Traditional IP Network Address Translator (Traditional NAT). 2001, January. Retrieved January 21, 2010. Available at: http://www.ietf.org/rfc/rfc3022.txt.

[16] [www-document]. RFC 1191: IP Path MTU Discovery. 1990, November. Retrieved February 24, 2010. Available at: http://tools.ietf.org/html/rfc1191

[17] [www-document]. RFC 4787: NAT UDP Unicast Requirements. 2007, January. Retrieved March 3, 2010. Available at: http://tools.ietf.org/rfc/rfc4787.txt

[18] [www-document]. Walter, K-D. Implementing M2M applications via GPRS, EDGE and UMTS. 2009, December. Retrieved Wednesday 27, 2010. M2M Alliance e.V., Aachen, Germany. Available at: http://m2m.com/docs/DOC-1003

[19] [www-document]. Flickr from Yahoo. GSM-GPRS Architecture. January 2, 2010. Retrieved Tuesday 13, 2010. Available at: http://farm3.static.flickr.com/2694/4239683146_f2c4a35b3f.jpg

[20] Bannister, J, Mather, P, Coope, S. 2004. *Convergence Technologies for 3G Networks: IP, UMTS; EGPRS and ATM*. Hoboken, NJ, USA: John Wiley & Sons, Incorporated. p 94.

[21] Bannister, J, Mather, P, Coope, S. 2004. *Convergence Technologies for 3G Networks: IP, UMTS; EGPRS and ATM*. Hoboken, NJ, USA: John Wiley & Sons, Incorporated. p 98.

[22] Bannister, J, Mather, P, Coope, S. 2004. *Convergence Technologies for 3G Networks: IP, UMTS; EGPRS and ATM*. Hoboken, NJ, USA: John Wiley & Sons, Incorporated. p 556.

[23] Bannister, J, Mather, P, Coope, S. 2004. *Convergence Technologies for 3G Networks: IP, UMTS; EGPRS and ATM*. Hoboken, NJ, USA: John Wiley & Sons, Incorporated. p 201.

[24] European Telecommunication Standards Institute / 3GPP. Specification TS 23.060 version 5.9.0 Release 5. 2004.

[25] Bannister, J, Mather, P, Coope, S. 2004. *Convergence Technologies for 3G Networks: IP, UMTS; EGPRS and ATM*. Hoboken, NJ, USA: John Wiley & Sons, Incorporated. p 88.

[26] Personal discussion with Senior Project Manager Hannu Rytilä from Kone Oy. Functionality of GPRS. March 18, 2010

[27] Bannister, J, Mather, P, Coope, S. 2004. *Convergence Technologies for 3G Networks: IP, UMTS; EGPRS and ATM*. Hoboken, NJ, USA: John Wiley & Sons, Incorporated. p 620-623

[28] Sharp, Robin. 2008. *Principles of Protocol Design*. Berlin, Heidelberg: Springer-Verlag. p. 76-82

[29] Sharp, Robin. 2008. *Principles of Protocol Design*. Berlin, Heidelberg: Springer-Verlag. p. 112-116

[30] Sharp, Robin. 2008. Principles of Protocol Design. Berlin, Heidelberg: Springer-Verlag. p. 241

[31] [www-document]. RFC 768: User Datagram Protocol. August 28, 1980. Retrieved March 29, 2010. Available at: http://www.ietf.org/rfc/rfc768.txt.

[32] [www-document]. OpenSourceProject.org.cn. BSD Sockets API Primer. Retrieved March 29, 2010 Available at: http://book.opensourceproject.org.cn/embedded/tcpipembedded/opensource/0108.html.

[33] [www-document]. Fairhurst, G. The User Datagram Protocol (UDP). November 19, 2008. Retrieved February 11, 2010. Available at: http://www.erg.abdn.ac.uk/users/gorry/eg3567/inet-pages/udp.html

[34] [www-document]. Wikibooks. Computer Networks/UDP. Retreived March 29, 2010. Available at: http://en.wikibooks.org/wiki/Computer_Networks/UDP

[35] [www-document]. Winsocketdotnetworkprogramming.com. IPv4 and IPv6 pseudo header format. 2009. Retrieved March 29, 2010. Available at: http://www.winsocketdotnetworkprogramming.com/clientserversocketnetworkcommunication8n.html

[36] [www-document]. Network Sorcery Inc. UDP, user datagram protocol. 1998-2009. Retrieved March 29, 2010. Available at: http://www.networksorcery.com/enp/protocol/udp.htm

[37] [www-document]. OBSAI: Reference Point 1 Specification, Appendix A. 2008. Retrieved February 11, 2010. Available at: http://www.obsai.com/obsai/documents/public_documents/download_specifications/rp_specifications

[38] Sharp, Robin. 2008. *Principles of Protocol Design*. Berlin, Heidelberg: Springer-Verlag. p. 173-175

[39] [www-document]. RFC 1334: IEEE PPP Authentication Protocols. 1992, October. Retrieved February 5, 2010. Available at: http://www.ietf.org/rfc/rfc1334.txt.

[40] [www-document]. RFC 3748: Extensible Authentication Protocol (EAP). 2004, June. Retrieved February 5, 2010. Available at: http://www.ietf.org/rfc/rfc3748.txt.

[41] [www-document]. EAP authentication protocols for WLANs. February 18, 2005. Retrieved April 4, 2010. Available at: http://www.ciscopress.com/articles/article.asp?p=369223

[42] [www-document]. Microsoft TechNet. EAP. January 21, 2005. Retrieved April 7, 2010. Available at: http://technet.microsoft.com/en-us/library/cc782851(WS.10).aspx

[43] [www-document]. Microsoft TechNet. PEAP. January 21, 2005. Retrieved April 7, 2010. Available at: http://technet.microsoft.com/en-us/library/cc757996(WS.10).aspx

[44] Bannister, J, Mather, P, Coope, S. 2004. *Convergence Technologies for 3G Networks: IP, UMTS; EGPRS and ATM*. Hoboken, NJ, USA: John Wiley & Sons, Incorporated. p 65.

[45] [www-document]. RFC 2865: Remote Authentication Dial In User Service (RADIUS). 2000, June. Retrieved February 4, 2010. Available at: http://www.ietf.org/rfc/rfc2865.txt.

10 Appendices

I Javadoc-documentation of the RMP application programming interface

II Javadoc-documentation of the reference implementation of RMP

III JAVA-source code of the reference implementation of RMP

APPENDIX I

Javadoc-documentation of the RMP Application Programming Interface

Antti Siirilä

Turku University of Applied Sciences

Package rmpAPI

Interface Summary	
ApplicationMessageAPI	
RMPClientAPI	
RMPServerAPI	

rmpAPI

Interface ApplicationMessageAPI

All Known Implementing Classes:

ApplicationMessage

`public interface` **`ApplicationMessageAPI`**

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This is the application message interface for the RMP client and server applications

---

Method Summary

void	**composeMsg**(boolean RACK, int msgType, byte[] appMsg) Compose an application message to be sent from the client
void	**composeMsg**(int deviceCategory, boolean RACK, boolean AF, int msgType, int deviceID, byte[] appMsg, java.net.InetAddress clientIP, int clientPort) Compose an application message to be sent from the server
boolean	**getAF**() Return the state of the action flag
byte[]	**getAppMsg**() Returns the encoded application message to be send for RMP
java.net.InetAddress	**getClientIP**() Return the IP-address of the client device from where the packet was sent
int	**getClientPort**() Return the UDP port of the client device from where the packet was sent
int	**getDeviceCategory**() Return the value of device category of the client device
int	**getDeviceID**() Return the value of device ID of the client device
byte[]	**getField**(int tag)

	Returns the value of the field matching with the given tag from the received application message
int	**getMsgType**() Return the value of message type
int	**getNextTag**() Return the next tag value in the application message
boolean	**getPMTUDiscoveryState**() returns the state of the PMTU Discovery
int	**getPMTUSize**() return the size of the PMTU
boolean	**getRMPMode**() Return the RMP mode information
void	**initMsg**(boolean RACK, int msgType) Initializes an application message for the client
void	**initMsg**(int deviceCategory, boolean RACK, boolean AF, int msgType, int deviceID, java.net.InetAddress clientIP, int clientPort) Initializes an application message to be sent from the server
boolean	**isRead**() Indicates the status of the received application message.
void	**msgReceived**(boolean AF, int msgType, int deviceID, byte[] appMsg) Copies the received message data into the client application
void	**msgReceived**(int deviceCategory, int msgType, int deviceID, byte[] appMsg, java.net.InetAddress clientIP, int clientPort) Copies the received message data into the server application
void	**pathMTUDiscoveryResult**(boolean status, int PMTUSize) Indicates the application about the status of the PMTU discovery process.
void	**setAF**(boolean AF) sets the action flag (only server)
boolean	**setField**(int tag, byte[] value) Set the data provided in the value-array with the given tag into the application message
void	**setOffSet**(int offSet) This the offset of the TLV tags in the application message.
void	**setPMTUSize**(int PMTUSize) sets the PMTU size
void	**setRMPMode**(boolean mode) sets the RMP mode

Method Detail

msgReceived

```
void msgReceived(boolean AF,
                 int msgType,
                 int deviceID,
                 byte[] appMsg)
```

Copies the received message data into the client application

Parameters:

`AF` - True: Action Flag set, False: Action Flag cleared
`msgType` - Application message type
`deviceID` - ID of the client device
`appMsg` - application data from RMP PDU

---

msgReceived

```
void msgReceived(int deviceCategory,
                 int msgType,
                 int deviceID,
                 byte[] appMsg,
                 java.net.InetAddress clientIP,
                 int clientPort)
```

Copies the received message data into the server application

Parameters:

`deviceCategory` - Category of the client device
`AF` - Action Flag
`msgType` - Application message type
`deviceID` - ID of the client device
`appMsg` - Application message data
`clientIP` - IP address of the client
`clientPort` - UDP port of client's RMP process.

---

pathMTUDiscoveryResult

```
void pathMTUDiscoveryResult(boolean status,
                            int PMTUSize)
```

Indicates the application about the status of the PMTU discovery process.

Parameters:

`status` - True: PMTU discovery successfully performed
`PMTUSize` -

---

getDeviceCategory

```
int getDeviceCategory()
```

Return the value of device category of the client device

Returns:
device category

---

getAF

```
boolean getAF()
```

Return the state of the action flag

Returns:
True: action flag set, False: action flag cleared

---

setAF

```
void setAF(boolean AF)
```

sets the action flag (only server)

Parameters:
AF -

---

getMsgType

```
int getMsgType()
```

Return the value of message type

Returns:
message type

---

getDeviceID

```
int getDeviceID()
```

Return the value of device ID of the client device

Returns:
device ID

---

getField

```
byte[] getField(int tag)
```

Returns the value of the field matching with the given tag from the received application message

Parameters:

tag -

Returns:

the field data in byte array

---

setField

```
boolean setField(int tag,
                 byte[] value)
```

Set the data provided in the value-array with the given tag into the application message

Parameters:

tag -

Returns:

True: success False: error

---

getNextTag

```
int getNextTag()
```

Return the next tag value in the application message

Returns:

next tag value || -1 if error

---

getAppMsg

```
byte[] getAppMsg()
```

Returns the encoded application message to be send for RMP

Returns:

Application message data (RMP payload)

---

getClientIP

```
java.net.InetAddress getClientIP()
```

Return the IP-address of the client device from where the packet was sent

Returns:

Client IP

---

getClientPort

```
int getClientPort()
```

Return the UDP port of the client device from where the packet was sent

Returns:
UDP port value

---

setRMPMode

```
void setRMPMode(boolean mode)
```

sets the RMP mode

Parameters:
`mode` -

---

getPMTUSize

```
int getPMTUSize()
```

return the size of the PMTU

Returns:
PMTU value

---

setPMTUSize

```
void setPMTUSize(int PMTUSize)
```

sets the PMTU size

Parameters:
`PMTUSize` -

---

composeMsg

```
void composeMsg(int deviceCategory,
                boolean RACK,
                boolean AF,
                int msgType,
                int deviceID,
                byte[] appMsg,
                java.net.InetAddress clientIP,
                int clientPort)
```

Compose an application message to be sent from the server

Parameters:
`deviceCategory` -

AF -
msgType -
deviceID -
appMsg -
clientIP -
clientPort -

---

initMsg

```
void initMsg(int deviceCategory,
             boolean RACK,
             boolean AF,
             int msgType,
             int deviceID,
             java.net.InetAddress clientIP,
             int clientPort)
```

Initializes an application message to be sent from the server

Parameters:

deviceCategory -
AF -
msgType -
deviceID -
clientIP -
clientPort -

---

composeMsg

```
void composeMsg(boolean RACK,
                int msgType,
                byte[] appMsg)
```

Compose an application message to be sent from the client

Parameters:

RACK -
msgType -
appMsg -

---

initMsg

```
void initMsg(boolean RACK,
             int msgType)
```

Initializes an application message for the client

Parameters:

RACK - RMP mode
msgType - Message type

getPMTUDiscoveryState

`boolean **getPMTUDiscoveryState**()`

returns the state of the PMTU Discovery

Returns:
True: Inactive False: Active

getRMPMode

`boolean **getRMPMode**()`

Return the RMP mode information

Returns:
True: RMP acknowledged. False: RMP unacknowledged

isRead

`boolean **isRead**()`

Indicates the status of the received application message.

Returns:
True: message is read and can be over written (application sets). False: unread message (RMP sets)

setOffSet

`void **setOffSet**(int offSet)`

This the offset of the TLV tags in the application message. When the getNextTag has returned -1, which indicates that the last tag was read, this method can be used to reset the offset to zero.

Parameters:
`offSet` -

rmpAPI

Interface RMPClientAPI

All Known Implementing Classes:

RMPClient

`public interface` **`RMPClientAPI`**

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 The RMP client implements the RMPClientAPI-interface. The client application can configure RMP parameters, and send and receive messages.

Method Summary

int	**configure**(int deviceCategory, int deviceID, int PMTU, java.net.InetAddress RMS_IP, java.net.InetAddress AS_IP, int RMS_PORT, int AS_PORT, int LOCAL_PORT, int TX_delay, int reTX_T, int udpBufferLength, int reTransmisssionRetryCount, int reassemblingTimeout) Sets the RMP client system parameters
int	**receiveMsg**(ApplicationMessageAPI callback) Non-blocking method for receiving an application message.
int	**receiveMsg**(ApplicationMessageAPI callback, int timeout) Blocking method for receiving an application message.
int	**sendMsg**(ApplicationMessageAPI callback) Non-blocking method for sending an application message.
int	**sendMsg**(ApplicationMessageAPI callback, int timeout) Blocking method for sending an application message.
int	**startPMTU**(ApplicationMessageAPI callback) Non-blocking type: start the PMTU discovery procedure.
int	**startPMTU**(ApplicationMessageAPI callback, int timeout) Blocking type: start the PMTU discovery procedure.

Method Detail

configure

```
int configure(int deviceCategory,
              int deviceID,
              int PMTU,
              java.net.InetAddress RMS_IP,
              java.net.InetAddress AS_IP,
              int RMS_PORT,
              int AS_PORT,
              int LOCAL_PORT,
              int TX_delay,
              int reTX_T,
              int udpBufferLength,
              int reTransmisssionRetryCount,
              int reassemblingTimeout)
```

Sets the RMP client system parameters

Parameters:

deviceCategory -
deviceID -
PMTU -
RMS_IP -
AS_IP -
RMS_PORT -
AS_PORT -
LOCAL_PORT -
TX_delay -
reTX_T -
udpBufferLength -
reTransmisssionRetryCount -
reassemblingTimeout -

Returns:

---

sendMsg

```
int sendMsg(ApplicationMessageAPI callback,
            int timeout)
```

Blocking method for sending an application message. Returns after the message is passed to UDP. Application shall perform the memory management for msg.

Parameters:

msg - - Reference to the RMP_Msg object

Returns:

- -1: Error

---

sendMsg

```
int sendMsg(ApplicationMessageAPI callback)
```

Non-blocking method for sending an application message. RMP creates a new thread for the sending process, copies the message object, and returns.

Parameters:

`msg` - - Reference to the message object. Application shall allocate the memory for the message Object. RMP shall remove the message object from the memory.

`RACK` - - True: RMP acknowledged mode. False: RMP unacknowledged mode

Returns:

- -1: Error

---

receiveMsg

```
int receiveMsg(ApplicationMessageAPI callback,
               int timeout)
```

Blocking method for receiving an application message. Returns if a message is received or after the timeout. RMP shall allocate the memory for the message object. Application shall remove the message object from the memory.

Parameters:

`timeout` - - Defines the period after when the method shall return if no message has been received.

Returns:

- the reference to the received message object

---

receiveMsg

```
int receiveMsg(ApplicationMessageAPI callback)
```

Non-blocking method for receiving an application message. RMP creates a new thread for the receiving process and returns. Once a message is received, RMP calls the msgReceived -callback method to pass the reference of the message object to the application. (callback.msgReceived(msg))

Parameters:

`callback` - - Reference to the application interface.

Returns:

- -1: Error

---

startPMTU

```
int startPMTU(ApplicationMessageAPI callback,
              int timeout)
```

Blocking type: start the PMTU discovery procedure. RMP indicates the application about the procedure status using callback.pathMTUDiscoveryResult(status) -callback method.

Parameters:

`callback` - - Reference to the application interface.

startPMTU

`int` **`startPMTU`**`(`ApplicationMessageAPI `callback)`

Non-blocking type: start the PMTU discovery procedure. RMP indicates the application about the procedure status using callback.pathMTUDiscoveryResult(status) -callback method.

Parameters:

`callback` - - Reference to the application interface.

rmpAPI

Interface RMPServerAPI

All Known Implementing Classes:

RMPServer

```
public interface RMPServerAPI
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013

Method Summary

int	**receiveMsg**(ApplicationMessageAPI receiveAppMsg) Non-blocking method for receiving an application message.
int	**sendMsg**(ApplicationMessageAPI sendAppMsg) Non-blocking method for sending an application message.
int	**setParameters**(int PMTU, int RMS_PORT, int TX_delay, int reTX_T, int udpBufferLength, int reTransmisssionRetryCount, int reassemblingTimeout) Sets the RMP server parameters.

Method Detail

setParameters

```
int setParameters(int PMTU,
                  int RMS_PORT,
                  int TX_delay,
                  int reTX_T,
                  int udpBufferLength,
                  int reTransmisssionRetryCount,
                  int reassemblingTimeout)
```

Sets the RMP server parameters.

Parameters:

PMTU -
RMS_PORT -
TX_delay -
reTX_T -
udpBufferLength -
reTransmisssionRetryCount -
reassemblingTimeout -

Returns:

-1 if error occurs and 0: for successful sending

---

sendMsg

```
int sendMsg(ApplicationMessageAPI sendAppMsg)
```

Non-blocking method for sending an application message. RMP creates a new thread for the sending process, copies the message object, and returns.

Parameters:

sendAppMsg - object that holds the application message data to be sent

Returns:

-1 if error occurs and 0: for successful sending

---

receiveMsg

```
int receiveMsg(ApplicationMessageAPI receiveAppMsg)
```

Non-blocking method for receiving an application message. RMP creates a new thread for the receiving process and returns. Once a message is received, RMP calls the msgReceived -callback method to pass the reference of the message object to the application. (callback.msgReceived(msg))

Parameters:

callback - object that will hold the received application message data

Returns:

-1 if error occurs and 0: for successful sending

APPENDIX II

Javadoc-documentation of the reference implementation of RMP

Antti Siirilä

Turku University of Applied Sciences

Package rmp

Class Summary	
ApplicationMessage	
ApplicationMessageTags	
ClientAppGUI	
ClientApplicationDemo	
InputOutput	
RMP	
RMPClient	
RMPFragmentationReassembling	
RMPPacket	
RMPPacketEncapsulationToDatagramPacket	
RMPServer	
ServerAppDemo	
ServerAppGUI	

rmp

Class ApplicationMessage

```
java.lang.Object
  └ rmp.ApplicationMessage
```

All Implemented Interfaces:

ApplicationMessageAPI

```
public class ApplicationMessage
extends java.lang.Object
implements ApplicationMessageAPI
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This class holds the methods and data of application level messages

---

Field Summary

private boolean	**AF**
private byte[]	**appMsg**
private java.net.InetAddress	**clientIP**
private int	**clientPort**
private int	**deviceCategory**
private int	**deviceID**
private byte[]	**field**
private boolean	**messageState**
private int	**msgType**

private int	**offSet**
private int	**PMTUSize**
private boolean	**RACK**
private boolean	**status**
private static int	**T_ACTION**
private static int	**T_ACTION_REQUEST**
private static int	**T_AUHTENTICATION_PERIOD**
private static int	**T_AUHTENTICATON_SERVER**
private static int	**T_BOM**
private static int	**T_BOM_RESPONSE**
private static int	**T_CONFIGURATION_REQUEST**
private static int	**T_CONFIGURATION_RESPONSE**
private static int	**T_COUNTER**
private static int	**T_DATE_TIME**
private static int	**T_FAULT_CODE**
private static int	**T_FAULT_FILTER**
private static int	**T_FRAGMENT_ACKNOWLEDGEMENT**
private static int	**T_MEASURE**
private static int	**T_PMTU**
private static int	**T_REGISTRATION_REQUEST**

private static int	**T_REGISTRATION_RESPONSE**
private static int	**T_RESEND_INTERVAL**
private static int	**T_RMS_ADDRESS**
private static int	**T_STATISTICS_REPORT_TIME**
private static int	**T_STATISTICS_RESPONSE**
private static int	**T_STATUS**
private static int	**T_STATUS_AFTER**
private static int	**T_STATUS_BEFORE**
private static int	**T_STATUS_LOG_FILTER**
private static int	**T_TIME_STAMP**

Constructor Summary

ApplicationMessage()
ApplicationMessage(byte[] appMsgReceived, boolean AF, int msgType) Client receive
ApplicationMessage(byte[] appMsgReceived, boolean AF, int msgType, int deviceID, java.net.InetAddress sourceIP, int sourcePort) Server receive

Method Summary

void	**composeMsg**(boolean RACK, int msgType, byte[] appMsg) Composes an application message for the client
void	**composeMsg**(int deviceCategory, boolean RACK, boolean AF, int msgType, int deviceID, byte[] appMsg, java.net.InetAddress clientIP, int clientPort) Composes an application message for the server
boolean	**getAF**() Return the state of the action flag
byte[]	**getAppMsg**() Returns the encoded application message to be send for RMP

java.net.InetAddress	**getClientIP**() Return the IP-address of the client device from where the packet was sent
int	**getClientPort**() Return the UDP port of the client device from where the packet was sent
int	**getDeviceCategory**() Return the value of device category of the client device
int	**getDeviceID**() Return the value of device ID of the client device
byte[]	**getField**(int tag) Returns the value of the field matching with the given tag from the received application message
int	**getMsgType**() Return the value of message type
int	**getNextTag**() Return the next tag value in the application message
boolean	**getPMTUDiscoveryState**() returns the state of the PMTU Discovery
int	**getPMTUSize**() return the size of the PMTU
boolean	**getRMPMode**() Return the RMP mode information
void	**initMsg**(boolean RACK, int msgType) Initializes an application message for the client
void	**initMsg**(int deviceCategory, boolean RACK, boolean AF, int msgType, int deviceID, java.net.InetAddress clientIP, int clientPort) Initializes an application message to be sent from the server
boolean	**isRead**() return the state of the message
private boolean	**isSyncRead**() Synchronized message state
void	**msgReceived**(boolean AF, int msgType, int deviceID, byte[] appMsg) Copies the received message data into the client application
void	**msgReceived**(int deviceCategory, int msgType, int deviceID, byte[] appMsg, java.net.InetAddress clientIP, int clientPort) Copies the received message data into the server application
void	**pathMTUDiscoveryResult**(boolean status, int PMTUSize) Indicates the application about the status of the PMTU discovery process.
void	**setAF**(boolean AF) sets the action flag (only server)

boolean	**setField**(int tag, byte[] value) Set the data provided in the value-array with the given tag into the application message
void	**setMessageState**(boolean state) sets the message state value
void	**setOffSet**(int offSet) This the offset of the TLV tags in the application message.
void	**setPMTUSize**(int PMTUSize) sets the PMTU size
void	**setRMPMode**(boolean RACK) sets the RMP mode
private void	**syncMessageState**(boolean state)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

deviceCategory

```
private int deviceCategory
```

AF

```
private boolean AF
```

RACK

```
private boolean RACK
```

---

msgType

```
private int msgType
```

---

deviceID

```
private int deviceID
```

---

appMsg

```
private byte[] appMsg
```

---

field

```
private byte[] field
```

---

clientIP

```
private java.net.InetAddress clientIP
```

---

clientPort

```
private int clientPort
```

---

status

```
private boolean status
```

PMTUSize

```
private int PMTUSize
```

---

messageState

```
private boolean messageState
```

---

offSet

```
private int offSet
```

T_REGISTRATION_REQUEST

```
private static final int T_REGISTRATION_REQUEST
```

See Also:

Constant Field Values

T_REGISTRATION_RESPONSE

```
private static final int T_REGISTRATION_RESPONSE
```

See Also:
Constant Field Values

T_CONFIGURATION_REQUEST

```
private static final int T_CONFIGURATION_REQUEST
```

See Also:
Constant Field Values

T_AUHTENTICATON_SERVER

```
private static final int T_AUHTENTICATON_SERVER
```

See Also:
Constant Field Values

T_AUHTENTICATION_PERIOD

```
private static final int T_AUHTENTICATION_PERIOD
```

See Also:
Constant Field Values

T_RMS_ADDRESS

```
private static final int T_RMS_ADDRESS
```

See Also:
Constant Field Values

T_PMTU

```
private static final int T_PMTU
```

See Also:
Constant Field Values

T_FRAGMENT_ACKNOWLEDGEMENT

```
private static final int T_FRAGMENT_ACKNOWLEDGEMENT
```

See Also:

Constant Field Values

T_DATE_TIME

```
private static final int T_DATE_TIME
```

See Also:
Constant Field Values

T_STATISTICS_REPORT_TIME

```
private static final int T_STATISTICS_REPORT_TIME
```

See Also:
Constant Field Values

T_FAULT_FILTER

```
private static final int T_FAULT_FILTER
```

See Also:
Constant Field Values

T_STATUS_LOG_FILTER

```
private static final int T_STATUS_LOG_FILTER
```

See Also:
Constant Field Values

T_RESEND_INTERVAL

```
private static final int T_RESEND_INTERVAL
```

See Also:
Constant Field Values

T_CONFIGURATION_RESPONSE

```
private static final int T_CONFIGURATION_RESPONSE
```

See Also:
Constant Field Values

T_FAULT_CODE

```
private static final int T_FAULT_CODE
```

See Also:
Constant Field Values

T_TIME_STAMP

```
private static final int T_TIME_STAMP
```

See Also:
Constant Field Values

T_STATUS

```
private static final int T_STATUS
```

See Also:
Constant Field Values

T_STATUS_BEFORE

```
private static final int T_STATUS_BEFORE
```

See Also:
Constant Field Values

T_STATUS_AFTER

```
private static final int T_STATUS_AFTER
```

See Also:
Constant Field Values

T_COUNTER

```
private static final int T_COUNTER
```

See Also:
Constant Field Values

T_MEASURE

```
private static final int T_MEASURE
```

See Also:
Constant Field Values

T_STATISTICS_RESPONSE

```
private static final int T_STATISTICS_RESPONSE
```

See Also:
Constant Field Values

T_ACTION_REQUEST

```
private static final int T_ACTION_REQUEST
```

See Also:
Constant Field Values

T_ACTION

```
private static final int T_ACTION
```

See Also:
Constant Field Values

T_BOM

```
private static final int T_BOM
```

See Also:
Constant Field Values

T_BOM_RESPONSE

```
private static final int T_BOM_RESPONSE
```

See Also:
Constant Field Values

Constructor Detail

ApplicationMessage

```
public ApplicationMessage()
```

ApplicationMessage

```
public ApplicationMessage(byte[] appMsgReceived,
                          boolean AF,
                          int msgType)
```

Client receive

Parameters:

appMsgReceived -
AF -
msgType -

ApplicationMessage

```
public ApplicationMessage(byte[] appMsgReceived,
                          boolean AF,
                          int msgType,
                          int deviceID,
                          java.net.InetAddress sourceIP,
                          int sourcePort)
```

Server receive

Parameters:

appMsgReceived -
AF -
msgType -
deviceID -
sourceIP -
sourcePort -

Method Detail

getAF

```
public boolean getAF()
```

Description copied from interface: **ApplicationMessageAPI**
Return the state of the action flag

Specified by:
getAF in interface ApplicationMessageAPI

Returns:
True: action flag set, False: action flag cleared

---

setAF

```
public void setAF(boolean AF)
```

Description copied from interface: **ApplicationMessageAPI**
sets the action flag (only server)

Specified by:
setAF in interface ApplicationMessageAPI

---

setRMPMode

```
public void setRMPMode(boolean RACK)
```

Description copied from interface: **ApplicationMessageAPI**
sets the RMP mode

Specified by:
setRMPMode in interface ApplicationMessageAPI

---

getRMPMode

```
public boolean getRMPMode()
```

Description copied from interface: **ApplicationMessageAPI**
Return the RMP mode information

Specified by:
getRMPMode in interface ApplicationMessageAPI

Returns:
True: RMP acknowledged. False: RMP unacknowledged

---

getAppMsg

```
public byte[] getAppMsg()
```

Description copied from interface: **ApplicationMessageAPI**
Returns the encoded application message to be send for RMP

Specified by:
getAppMsg in interface ApplicationMessageAPI

Returns:
Application message data (RMP payload)

---

getClientIP

```
public java.net.InetAddress getClientIP()
```

Description copied from interface: **ApplicationMessageAPI**
Return the IP-address of the client device from where the packet was sent

Specified by:
getClientIP in interface ApplicationMessageAPI

Returns:
Client IP

---

getClientPort

```
public int getClientPort()
```

Description copied from interface: **ApplicationMessageAPI**
Return the UDP port of the client device from where the packet was sent

Specified by:
getClientPort in interface ApplicationMessageAPI

Returns:
UDP port value

---

getDeviceCategory

```
public int getDeviceCategory()
```

Description copied from interface: **ApplicationMessageAPI**
Return the value of device category of the client device

Specified by:
getDeviceCategory in interface ApplicationMessageAPI

Returns:
device category

---

getDeviceID

```
public int getDeviceID()
```

Description copied from interface: **ApplicationMessageAPI**
Return the value of device ID of the client device

Specified by:
getDeviceID in interface ApplicationMessageAPI

Returns:
device ID

---

getField

```
public byte[] getField(int tag)
```

Description copied from interface: ApplicationMessageAPI
Returns the value of the field matching with the given tag from the received application message

Specified by:
getField in interface ApplicationMessageAPI

Returns:
the field data in byte array

---

getMsgType

```
public int getMsgType()
```

Description copied from interface: ApplicationMessageAPI
Return the value of message type

Specified by:
getMsgType in interface ApplicationMessageAPI

Returns:
message type

---

msgReceived

```
public void msgReceived(boolean AF,
                        int msgType,
                        int deviceID,
                        byte[] appMsg)
```

Description copied from interface: ApplicationMessageAPI
Copies the received message data into the client application

Specified by:
msgReceived in interface ApplicationMessageAPI

Parameters:
`AF` - True: Action Flag set, False: Action Flag cleared
`msgType` - Application message type
`deviceID` - ID of the client device
`appMsg` - application data from RMP PDU

---

msgReceived

```
public void msgReceived(int deviceCategory,
                        int msgType,
```

```
                        int deviceID,
                        byte[] appMsg,
                        java.net.InetAddress clientIP,
                        int clientPort)
```

Description copied from interface: **ApplicationMessageAPI**
Copies the received message data into the server application

Specified by:
msgReceived in interface ApplicationMessageAPI

Parameters:
deviceCategory - Category of the client device
msgType - Application message type
deviceID - ID of the client device
appMsg - Application message data
clientIP - IP address of the client
clientPort - UDP port of client's RMP process.

---

pathMTUDiscoveryResult

```
public void pathMTUDiscoveryResult(boolean status,
                                   int PMTUSize)
```

Description copied from interface: **ApplicationMessageAPI**
Indicates the application about the status of the PMTU discovery process.

Specified by:
pathMTUDiscoveryResult in interface ApplicationMessageAPI

Parameters:
status - True: PMTU discovery successfully performed

---

setField

```
public boolean setField(int tag,
                        byte[] value)
```

Description copied from interface: **ApplicationMessageAPI**
Set the data provided in the value-array with the given tag into the application message

Specified by:
setField in interface ApplicationMessageAPI

Returns:
True: success False: error

---

getNextTag

```
public int getNextTag()
```

Description copied from interface: ApplicationMessageAPI
Return the next tag value in the application message

Specified by:
getNextTag in interface ApplicationMessageAPI

Returns:
next tag value || -1 if error

---

getPMTUSize

```
public int getPMTUSize()
```

Description copied from interface: ApplicationMessageAPI
return the size of the PMTU

Specified by:
getPMTUSize in interface ApplicationMessageAPI

Returns:
PMTU value

---

setPMTUSize

```
public void setPMTUSize(int PMTUSize)
```

sets the PMTU size

Specified by:
setPMTUSize in interface ApplicationMessageAPI

Parameters:
PMTUSize -

---

getPMTUDiscoveryState

```
public boolean getPMTUDiscoveryState()
```

returns the state of the PMTU Discovery

Specified by:
getPMTUDiscoveryState in interface ApplicationMessageAPI

Returns:
True: Inactive False: Active

---

composeMsg

```
public void composeMsg(int deviceCategory,
                       boolean RACK,
                       boolean AF,
                       int msgType,
                       int deviceID,
                       byte[] appMsg,
                       java.net.InetAddress clientIP,
                       int clientPort)
```

Composes an application message for the server

Specified by:

composeMsg in interface ApplicationMessageAPI

---

composeMsg

```
public void composeMsg(boolean RACK,
                       int msgType,
                       byte[] appMsg)
```

Composes an application message for the client

Specified by:

composeMsg in interface ApplicationMessageAPI

---

isRead

```
public boolean isRead()
```

return the state of the message

Specified by:

isRead in interface ApplicationMessageAPI

Returns:

True: message is read and can be over written (application sets). False: unread message (RMP sets)

---

isSyncRead

```
private boolean isSyncRead()
```

Synchronized message state

Returns:

---

setMessageState

```
public void setMessageState(boolean state)
```

sets the message state value

Parameters:

`state` - True: read (application). False: not read (RMP)

---

syncMessageState

```
private void syncMessageState(boolean state)
```

---

initMsg

```
public void initMsg(int deviceCategory,
                    boolean RACK,
                    boolean AF,
                    int msgType,
                    int deviceID,
                    java.net.InetAddress clientIP,
                    int clientPort)
```

Description copied from interface: **ApplicationMessageAPI**
Initializes an application message to be sent from the server

Specified by:

initMsg in interface ApplicationMessageAPI

---

initMsg

```
public void initMsg(boolean RACK,
                    int msgType)
```

Description copied from interface: **ApplicationMessageAPI**
Initializes an application message for the client

Specified by:

initMsg in interface ApplicationMessageAPI

Parameters:

`RACK` - RMP mode
`msgType` - Message type

---

setOffSet

```
public void setOffSet(int offSet)
```

Description copied from interface: **ApplicationMessageAPI**
This the offset of the TLV tags in the application message. When the getNextTag has returned -1,

which indicates that the last tag was read, this method can be used to reset the offset to zero.

Specified by:

`setOffSet` in interface `ApplicationMessageAPI`

---

rmp

Class ApplicationMessageTags

```
java.lang.Object
  └ rmp.ApplicationMessageTags
```

```
public class ApplicationMessageTags
extends java.lang.Object
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This class holds the information of application message tags. The class can be used to convert a integer tag to a string name.

Field Summary

private static int	T_ACTION
private static int	T_ACTION_REQUEST
private static int	T_AUHTENTICATION_PERIOD
private static int	T_AUHTENTICATON_SERVER
private static int	T_BOM
private static int	T_BOM_RESPONSE
private static int	T_CONFIGURATION_REQUEST
private static int	T_CONFIGURATION_RESPONSE
private static int	T_COUNTER
private static int	T_DATE_TIME

private static int	**T_FAULT_CODE**
private static int	**T_FAULT_FILTER**
private static int	**T_FRAGMENT_ACKNOWLEDGEMENT**
private static int	**T_MEASURE**
private static int	**T_PMTU**
private static int	**T_REGISTRATION_REQUEST**
private static int	**T_REGISTRATION_RESPONSE**
private static int	**T_RESEND_INTERVAL**
private static int	**T_RMS_ADDRESS**
private static int	**T_STATISTICS_REPORT_TIME**
private static int	**T_STATISTICS_RESPONSE**
private static int	**T_STATUS**
private static int	**T_STATUS_AFTER**
private static int	**T_STATUS_BEFORE**
private static int	**T_STATUS_LOG_FILTER**
private static int	**T_TIME_STAMP**
private java.lang.String	**tagName**

Constructor Summary

ApplicationMessageTags(int tag)

Method Summary

java.lang.String	**getTagName**()
void	**setTagName**(java.lang.String tagName)
private void	**switchTagToName**(int tag)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

T_REGISTRATION_REQUEST

private static final int **T_REGISTRATION_REQUEST**

See Also:
Constant Field Values

T_REGISTRATION_RESPONSE

private static final int **T_REGISTRATION_RESPONSE**

See Also:
Constant Field Values

T_CONFIGURATION_REQUEST

private static final int **T_CONFIGURATION_REQUEST**

See Also:
Constant Field Values

T_AUHTENTICATON_SERVER

private static final int **T_AUHTENTICATON_SERVER**

See Also:
Constant Field Values

T_AUHTENTICATION_PERIOD

```
private static final int T_AUHTENTICATION_PERIOD
```

See Also:
Constant Field Values

T_RMS_ADDRESS

```
private static final int T_RMS_ADDRESS
```

See Also:
Constant Field Values

T_PMTU

```
private static final int T_PMTU
```

See Also:
Constant Field Values

T_FRAGMENT_ACKNOWLEDGEMENT

```
private static final int T_FRAGMENT_ACKNOWLEDGEMENT
```

See Also:
Constant Field Values

T_DATE_TIME

```
private static final int T_DATE_TIME
```

See Also:
Constant Field Values

T_STATISTICS_REPORT_TIME

```
private static final int T_STATISTICS_REPORT_TIME
```

See Also:
Constant Field Values

T_FAULT_FILTER

```
private static final int T_FAULT_FILTER
```

See Also:

Constant Field Values

T_STATUS_LOG_FILTER

```
private static final int T_STATUS_LOG_FILTER
```

See Also:
Constant Field Values

T_RESEND_INTERVAL

```
private static final int T_RESEND_INTERVAL
```

See Also:
Constant Field Values

T_CONFIGURATION_RESPONSE

```
private static final int T_CONFIGURATION_RESPONSE
```

See Also:
Constant Field Values

T_FAULT_CODE

```
private static final int T_FAULT_CODE
```

See Also:
Constant Field Values

T_TIME_STAMP

```
private static final int T_TIME_STAMP
```

See Also:
Constant Field Values

T_STATUS

```
private static final int T_STATUS
```

See Also:
Constant Field Values

T_STATUS_BEFORE

```
private static final int T_STATUS_BEFORE
```

See Also:
Constant Field Values

T_STATUS_AFTER

```
private static final int T_STATUS_AFTER
```

See Also:
Constant Field Values

T_COUNTER

```
private static final int T_COUNTER
```

See Also:
Constant Field Values

T_MEASURE

```
private static final int T_MEASURE
```

See Also:
Constant Field Values

T_STATISTICS_RESPONSE

```
private static final int T_STATISTICS_RESPONSE
```

See Also:
Constant Field Values

T_ACTION_REQUEST

```
private static final int T_ACTION_REQUEST
```

See Also:
Constant Field Values

T_ACTION

```
private static final int T_ACTION
```

See Also:
Constant Field Values

T_BOM

```
private static final int T_BOM
```

See Also:
Constant Field Values

T_BOM_RESPONSE

```
private static final int T_BOM_RESPONSE
```

See Also:
Constant Field Values

---

tagName

```
private java.lang.String tagName
```

Constructor Detail

ApplicationMessageTags

```
public ApplicationMessageTags(int tag)
```

Method Detail

switchTagToName

```
private void switchTagToName(int tag)
```

---

getTagName

```
public java.lang.String getTagName()
```

Returns:
the tagName

---

setTagName

```
public void setTagName(java.lang.String tagName)
```

Parameters:

`tagName` - the tagName to set

---

rmp

Class ClientAppGUI

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
                  └javax.swing.JFrame
                      └rmp.ClientAppGUI
```

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

```
public class ClientAppGUI
extends javax.swing.JFrame
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This is the client application GUI to control client activities.

See Also:

Serialized Form

Nested Class Summary

class	**ClientAppGUI.EventListener** Private inner class that handles the event when the user clicks the Calculate button.

Nested classes/interfaces inherited from class javax.swing.JFrame
javax.swing.JFrame.AccessibleJFrame

Nested classes/interfaces inherited from class java.awt.Frame
java.awt.Frame.AccessibleAWTFrame

Nested classes/interfaces inherited from class java.awt.Window
java.awt.Window.AccessibleAWTWindow

Nested classes/interfaces inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BaselineResizeBehavior, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

private static int	**ACTION**
private static int	**ACTION_REQUEST**
private static int	**ACTION_RESPONSE**
private java.net.InetAddress	**AS_IP**
private int	**AS_PORT**
private static int	**AUTHENTICATING**
private static int	**BOM_REQUEST**
private static int	**BOM_RESPONSE**
private javax.swing.JPanel	**buttonPanel**
private javax.swing.JButton[]	**buttonPanelB**
javax.swing.JTextArea	**cipherText1**
javax.swing.JTextArea	**cipherText2**
private ClientAppGUI.EventListener	**clientListener**
private static int	**CONFIGURATION**
private static int	**CONFIGURATION_REQUEST**
private static int	**CONFIGURATION_RESPONSE**

private javax.swing.JButton[]	**configurationButton**
private javax.swing.JPanel	**configurationPanel**
private static int	**CONFIGURING**
private int	**deviceCategory**
private int	**deviceID**
private static int	**DISABLED**
private static int	**FAULT_OCCURRED_REQUEST**
private static int	**FAULT_OCCURRED_RESPONSE**
private static int	**FAULT_UPDATE_REQUEST**
private static int	**FAULT_UPDATE_RESPONSE**
private static int	**GO_TO_IDLE**
private static int	**IDLE**
private int	**LOCAL_PORT**
private javax.swing.ButtonGroup	**messages**
private java.lang.String	**msg**
private int	**msgType**
private javax.swing.JRadioButton[]	**msgTypes**
private javax.swing.JTextField[]	**parameterInput**
private javax.swing.JLabel[]	**parameters**
private int	**PMTU**

private javax.swing.JCheckBox	**RACK**
private int	**reassemblingTimeout**
private static int	**RECEIVE**
private static int	**RECEIVED**
private static int	**RECEIVING**
private static int	**REGISTERING**
private static int	**REGISTRATION_REQUEST**
private static int	**REGISTRATION_RESPONSE**
private int	**reTransmisssionRetryCount**
private int	**reTX_T**
private boolean	**RMPMode**
private java.net.InetAddress	**RMS_IP**
private int	**RMS_PORT**
private javax.swing.JScrollPane	**scrollingArea1**
private javax.swing.JScrollPane	**scrollingArea2**
private javax.swing.JButton[]	**send**
private static int	**SEND**
private static int	**SENDING**
private javax.swing.JPanel	**sendingPanel**
private javax.swing.JTextField	**sendingPanelInput**
private int	**State**

private static int	**STATISTICS_REPORT_REQUEST**
private static int	**STATISTICS_REPORT_RESPONSE**
private javax.swing.JTabbedPane	**systems**
private javax.swing.JPanel	**TextArea**
private int	**TX_delay**
private int	**udpBufferLength**

Fields inherited from class javax.swing.JFrame
accessibleContext, EXIT_ON_CLOSE, rootPane, rootPaneCheckingEnabled

Fields inherited from class java.awt.Frame
CROSSHAIR_CURSOR, DEFAULT_CURSOR, E_RESIZE_CURSOR, HAND_CURSOR, ICONIFIED, MAXIMIZED_BOTH, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MOVE_CURSOR, N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR, SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR, W_RESIZE_CURSOR, WAIT_CURSOR

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface javax.swing.WindowConstants
DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

ClientAppGUI() Constructs the client GUI

Method Summary

private void	**buildButtonPanel**() The buildButtonPanel method builds the button panel.
private void	**buildConfigurationPanel**() Build the client configuration panel with required input fields

private void	**buildSendingPanel**() build the client sendigPanel
private void	**buildTabPane**() build the Tab pane and add the panel into it
private void	**buildTextArea**() Build the text areas
java.net.InetAddress	**getAS_IP**()
int	**getAS_PORT**()
int	**getDeviceCategory**()
int	**getDeviceID**()
int	**getLOCAL_PORT**()
java.lang.String	**getMsg**()
int	**getMsgType**()
int	**getPMTU**()
int	**getReassemblingTimeout**()
int	**getReTransmisssionRetryCount**()
int	**getReTX_T**()
java.net.InetAddress	**getRMS_IP**()
int	**getRMS_PORT**()
int	**getState**() Synchronized method to get the client state
int	**getTX_delay**()
int	**getUdpBufferLength**()
boolean	**isRMPMode**()
void	**setAS_IP**(java.net.InetAddress aSIP)

void	**setAS_PORT**(int aSPORT)
void	**setDeviceCategory**(int deviceCategory)
void	**setDeviceID**(int deviceID)
void	**setLOCAL_PORT**(int lOCALPORT)
void	**setMsg**(java.lang.String msg)
void	**setMsgType**(int msgType)
void	**setPMTU**(int pMTU)
void	**setReassemblingTimeout**(int reassemblingTimeout)
void	**setReTransmisssionRetryCount**(int reTransmisssionRetryCount)
void	**setReTX_T**(int reTXT)
void	**setRMPMode**(boolean rMPMode)
void	**setRMS_IP**(java.net.InetAddress rMSIP)
void	**setRMS_PORT**(int rMSPORT)
void	**setState**(int State) Synchronized method to set the client state
void	**setTX_delay**(int tXDelay)
void	**setUdpBufferLength**(int udpBufferLength)

Methods inherited from class javax.swing.JFrame
addImpl, createRootPane, frameInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getGraphics, getJMenuBar, getLayeredPane, getRootPane, getTransferHandler, isDefaultLookAndFeelDecorated, isRootPaneCheckingEnabled, paramString, processWindowEvent, remove, repaint, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setIconImage, setJMenuBar, setLayeredPane, setLayout, setRootPane, setRootPaneCheckingEnabled, setTransferHandler, update

Methods inherited from class java.awt.Frame

addNotify, getCursorType, getExtendedState, getFrames, getIconImage, getMaximizedBounds, getMenuBar, getTitle, isResizable, isUndecorated, remove, removeNotify, setCursor, setExtendedState, setMaximizedBounds, setMenuBar, setResizable, setTitle, setUndecorated

Methods inherited from class java.awt.Window

addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getGraphicsConfiguration, getIconImages, getInputContext, getListeners, getLocale, getModalExclusionType, getMostRecentFocusOwner, getOwnedWindows, getOwner, getOwnerlessWindows, getToolkit, getWarningString, getWindowFocusListeners, getWindowListeners, getWindows, getWindowStateListeners, hide, isActive, isAlwaysOnTop, isAlwaysOnTopSupported, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isShowing, pack, paint, postEvent, processEvent, processWindowFocusEvent, processWindowStateEvent, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, reshape, setAlwaysOnTop, setBounds, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setIconImages, setLocationByPlatform, setLocationRelativeTo, setMinimumSize, setModalExclusionType, setSize, setSize, setVisible, show, toBack, toFront

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBaseline, getBaselineResizeBehavior, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered,

isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isOpaque, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resize, resize, setBackground, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setLocation, setLocation, setMaximumSize, setName, setPreferredSize, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.awt.MenuContainer

getFont, postEvent

Field Detail

deviceCategory

private int **deviceCategory**

---

deviceID

private int **deviceID**

PMTU

private int **PMTU**

RMS_IP

private java.net.InetAddress **RMS_IP**

AS_IP

private java.net.InetAddress **AS_IP**

RMS_PORT

private int **RMS_PORT**

AS_PORT

private int **AS_PORT**

LOCAL_PORT

private int **LOCAL_PORT**

TX_delay

private int **TX_delay**

reTX_T

private int **reTX_T**

udpBufferLength

private int **udpBufferLength**

reTransmisssionRetryCount

private int **reTransmisssionRetryCount**

reassemblingTimeout

private int **reassemblingTimeout**

State

private int **State**

msgType

private int **msgType**

msg

```
private java.lang.String msg
```

RMPMode

```
private boolean RMPMode
```

DISABLED

```
private static final int DISABLED
```

See Also:
Constant Field Values

AUTHENTICATING

```
private static final int AUTHENTICATING
```

See Also:
Constant Field Values

REGISTERING

```
private static final int REGISTERING
```

See Also:
Constant Field Values

IDLE

```
private static final int IDLE
```

See Also:
Constant Field Values

SENDING

```
private static final int SENDING
```

See Also:
Constant Field Values

RECEIVING

```
private static final int RECEIVING
```

See Also:
Constant Field Values

CONFIGURING

```
private static final int CONFIGURING
```

See Also:
Constant Field Values

GO_TO_IDLE

```
private static final int GO_TO_IDLE
```

See Also:
Constant Field Values

RECEIVED

```
private static final int RECEIVED
```

See Also:
Constant Field Values

RECEIVE

```
private static final int RECEIVE
```

See Also:
Constant Field Values

ACTION

```
private static final int ACTION
```

See Also:
Constant Field Values

SEND

```
private static final int SEND
```

See Also:
Constant Field Values

REGISTRATION_REQUEST

```
private static final int REGISTRATION_REQUEST
```

See Also:
Constant Field Values

CONFIGURATION_REQUEST

```
private static final int CONFIGURATION_REQUEST
```

See Also:
Constant Field Values

FAULT_OCCURRED_REQUEST

```
private static final int FAULT_OCCURRED_REQUEST
```

See Also:
Constant Field Values

FAULT_UPDATE_REQUEST

```
private static final int FAULT_UPDATE_REQUEST
```

See Also:
Constant Field Values

STATISTICS_REPORT_REQUEST

```
private static final int STATISTICS_REPORT_REQUEST
```

See Also:
Constant Field Values

ACTION_REQUEST

```
private static final int ACTION_REQUEST
```

See Also:
Constant Field Values

BOM_REQUEST

```
private static final int BOM_REQUEST
```

See Also:
Constant Field Values

REGISTRATION_RESPONSE

```
private static final int REGISTRATION_RESPONSE
```

See Also:
Constant Field Values

CONFIGURATION

```
private static final int CONFIGURATION
```

See Also:
Constant Field Values

CONFIGURATION_RESPONSE

```
private static final int CONFIGURATION_RESPONSE
```

See Also:
Constant Field Values

FAULT_OCCURRED_RESPONSE

```
private static final int FAULT_OCCURRED_RESPONSE
```

See Also:
Constant Field Values

FAULT_UPDATE_RESPONSE

```
private static final int FAULT_UPDATE_RESPONSE
```

See Also:
Constant Field Values

STATISTICS_REPORT_RESPONSE

```
private static final int STATISTICS_REPORT_RESPONSE
```

See Also:
Constant Field Values

ACTION_RESPONSE

```
private static final int ACTION_RESPONSE
```

See Also:
Constant Field Values

BOM_RESPONSE

```
private static final int BOM_RESPONSE
```

See Also:
Constant Field Values

---

cipherText1

```
public javax.swing.JTextArea cipherText1
```

---

cipherText2

```
public javax.swing.JTextArea cipherText2
```

---

buttonPanel

```
private javax.swing.JPanel buttonPanel
```

TextArea

```
private javax.swing.JPanel TextArea
```

---

buttonPanelB

```
private javax.swing.JButton[] buttonPanelB
```

---

configurationButton

```
private javax.swing.JButton[] configurationButton
```

---

send

```
private javax.swing.JButton[] send
```

---

scrollingArea1

```
private javax.swing.JScrollPane scrollingArea1
```

---

scrollingArea2

```
private javax.swing.JScrollPane scrollingArea2
```

---

parameterInput

```
private javax.swing.JTextField[] parameterInput
```

---

sendingPanelInput

```
private javax.swing.JTextField sendingPanelInput
```

---

msgTypes

```
private javax.swing.JRadioButton[] msgTypes
```

---

messages

```
private javax.swing.ButtonGroup messages
```

RACK

```
private javax.swing.JCheckBox RACK
```

---

parameters

```
private javax.swing.JLabel[] parameters
```

---

configurationPanel

```
private javax.swing.JPanel configurationPanel
```

---

systems

```
private javax.swing.JTabbedPane systems
```

---

sendingPanel

```
private javax.swing.JPanel sendingPanel
```

---

clientListener

```
private ClientAppGUI.EventListener clientListener
```

Constructor Detail

ClientAppGUI

```
public ClientAppGUI()
```

Constructs the client GUI

Method Detail

buildTabPane

```
private void buildTabPane()
```

build the Tab pane and add the panel into it

---

buildConfigurationPanel

```
private void buildConfigurationPanel()
```

Build the client configuration panel with required input fields

---

buildSendingPanel

```
private void buildSendingPanel()
```

build the client sendigPanel

---

buildTextArea

```
private void buildTextArea()
```

Build the text areas

---

buildButtonPanel

```
private void buildButtonPanel()
```

The buildButtonPanel method builds the button panel.

---

setState

```
public void setState(int State)
```

Synchronized method to set the client state

Overrides:
`setState` in class `java.awt.Frame`

Parameters:
`State` - state value

---

getState

```
public int getState()
```

Synchronized method to get the client state

Overrides:
`getState` in class `java.awt.Frame`

Returns:

---

getDeviceCategory

```
public int getDeviceCategory()
```

Returns:
the deviceCategory

---

setDeviceCategory

```
public void setDeviceCategory(int deviceCategory)
```

Parameters:
`deviceCategory` - the deviceCategory to set

---

getDeviceID

```
public int getDeviceID()
```

Returns:
the deviceID

---

setDeviceID

```
public void setDeviceID(int deviceID)
```

Parameters:
`deviceID` - the deviceID to set

---

getPMTU

```
public int getPMTU()
```

Returns:
the pMTU

---

setPMTU

```
public void setPMTU(int pMTU)
```

Parameters:
`pMTU` - the pMTU to set

---

getRMS_IP

```
public java.net.InetAddress getRMS_IP()
```

Returns:
the rMS_IP

---

setRMS_IP

```
public void setRMS_IP(java.net.InetAddress rMSIP)
```

Parameters:
`rMSIP` - the rMS_IP to set

---

getAS_IP

```
public java.net.InetAddress getAS_IP()
```

Returns:
the aS_IP

---

setAS_IP

```
public void setAS_IP(java.net.InetAddress aSIP)
```

Parameters:
`aSIP` - the aS_IP to set

---

getRMS_PORT

```
public int getRMS_PORT()
```

Returns:
the rMS_PORT

---

setRMS_PORT

```
public void setRMS_PORT(int rMSPORT)
```

Parameters:
`rMSPORT` - the rMS_PORT to set

---

getAS_PORT

```
public int getAS_PORT()
```

Returns:
the aS_PORT

---

setAS_PORT

```
public void setAS_PORT(int aSPORT)
```

Parameters:
`aSPORT` - the aS_PORT to set

---

getLOCAL_PORT

```
public int getLOCAL_PORT()
```

Returns:
the lOCAL_PORT

---

setLOCAL_PORT

```
public void setLOCAL_PORT(int lOCALPORT)
```

Parameters:
`lOCALPORT` - the lOCAL_PORT to set

---

getTX_delay

```
public int getTX_delay()
```

Returns:
the tX_delay

---

setTX_delay

```
public void setTX_delay(int tXDelay)
```

Parameters:
`tXDelay` - the tX_delay to set

---

getReTX_T

```
public int getReTX_T()
```

Returns:
the reTX_T

---

setReTX_T

```
public void setReTX_T(int reTXT)
```

Parameters:
`reTXT` - the reTX_T to set

---

getUdpBufferLength

```
public int getUdpBufferLength()
```

Returns:
the udpBufferLength

---

setUdpBufferLength

```
public void setUdpBufferLength(int udpBufferLength)
```

Parameters:
`udpBufferLength` - the udpBufferLength to set

---

getReTransmisssionRetryCount

```
public int getReTransmisssionRetryCount()
```

Returns:
the reTransmisssionRetryCount

---

setReTransmisssionRetryCount

```
public void setReTransmisssionRetryCount(int reTransmisssionRetryCount)
```

Parameters:
`reTransmisssionRetryCount` - the reTransmisssionRetryCount to set

---

getReassemblingTimeout

```
public int getReassemblingTimeout()
```

Returns:
the reassemblingTimeout

---

setReassemblingTimeout

```
public void setReassemblingTimeout(int reassemblingTimeout)
```

Parameters:
`reassemblingTimeout` - the reassemblingTimeout to set

---

getMsgType

```
public int getMsgType()
```

Returns:
the msgType

---

setMsgType

```
public void setMsgType(int msgType)
```

Parameters:

`msgType` - the msgType to set

---

getMsg

```
public java.lang.String getMsg()
```

Returns:
the msg

---

setMsg

```
public void setMsg(java.lang.String msg)
```

Parameters:
`msg` - the msg to set

---

isRMPMode

```
public boolean isRMPMode()
```

Returns:
the rMPMode

---

setRMPMode

```
public void setRMPMode(boolean rMPMode)
```

Parameters:
`rMPMode` - the rMPMode to set

---

rmp

Class ClientApplicationDemo

```
java.lang.Object
  └ rmp.ClientApplicationDemo
```

```
public class ClientApplicationDemo
extends java.lang.Object
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This is the RMP client application demonstration. The client registers with the active server and sends fault, statistics, and configuration messages when requested from the client GUI. In addition, the client performs the action procedure if server so requires.

Constructor Summary

ClientApplicationDemo()

Method Summary

static void	**main**(java.lang.String[] args)
private static void	**printReceivedmsg**(ClientAppGUI gui, ApplicationMessageAPI receiving)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ClientApplicationDemo

```
public ClientApplicationDemo()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
                throws java.net.SocketTimeoutException
```

Parameters:

args -

Throws:

java.net.SocketTimeoutException

---

printReceivedmsg

```
private static void printReceivedmsg(ClientAppGUI gui,
                                     ApplicationMessageAPI receiving)
```

---

rmp

Class InputOutput

```
java.lang.Object
  └ rmp.InputOutput
```

```
public class InputOutput
extends java.lang.Object
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This class is used when ever RMP needs input or output stream services

Field Summary

java.io.ByteArrayInputStream	**bin**
java.io.ByteArrayOutputStream	**bout**
java.io.DataInputStream	**din**
java.io.DataOutputStream	**dout**

Constructor Summary

InputOutput() Constructs an output stream
InputOutput(byte[] input) Constructs an input stream from given array

Method Summary

void	**closeInputStream**() Closes the input stream
void	**closeOutputStream**()

	Closes the output stream
void	**startInputStream**(byte[] input) Start an input stream from the given array
void	**stratOutputStream**() Starts an output stream

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

bin

```
public java.io.ByteArrayInputStream bin
```

---

bout

```
public java.io.ByteArrayOutputStream bout
```

---

din

```
public java.io.DataInputStream din
```

---

dout

```
public java.io.DataOutputStream dout
```

Constructor Detail

InputOutput

```
public InputOutput()
```

Constructs an output stream

InputOutput

```
public InputOutput(byte[] input)
```

Constructs an input stream from given array

Parameters:

`input` - data

Method Detail

stratOutputStream

```
public void stratOutputStream()
```

Starts an output stream

---

closeOutputStream

```
public void closeOutputStream()
```

Closes the output stream

---

startInputStream

```
public void startInputStream(byte[] input)
```

Start an input stream from the given array

Parameters:

`input` - data

---

closeInputStream

```
public void closeInputStream()
```

Closes the input stream

---

rmp

Class RMP

```
java.lang.Object
  └ rmp.RMP
```

Direct Known Subclasses:

RMPClient, RMPServer

```
public class RMP
extends java.lang.Object
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This class holds the RMP level functions common for both the server and the client

---

Field Summary

(package private) int	**counter**
(package private) int	**counter2**
protected boolean	**encore**
protected int	**error**
protected int	**msqID**
protected int	**PMTU**
protected int	**reassemblingTimeoutPeriod**
protected RMPPacket	**receive**
protected	**receiving**

ApplicationMessageAPI	
protected byte[]	**receivingUDPBuffer**
protected int	**receivingUDPBufferLength**
protected int	**retransmissionCount**
protected int	**reTXT**
protected RMPPacket	**send**
protected ApplicationMessageAPI	**sending**
protected int	**sessionid**
protected java.net.DatagramPacket	**toReceive**
protected java.net.DatagramPacket	**toSend**
protected int	**TX_delay**
protected java.net.DatagramSocket	**udp**
protected int	**updPort**

Constructor Summary

RMP()

Method Summary

protected void	**generateMsgID**() Generates the message ID and store it into msgID field
protected boolean	**getFromUDP**(int timeout) Receive DatagramPacket from UDP
protected boolean	**passToUDP**(java.net.DatagramPacket toUDP, boolean rmpMode, boolean RMPType, int msgID, int reTransmissionTimout) Passes the packet to udp.
protected boolean	**receive**() Receives the DatagramPacket, decapsulates it to RMPPacket, and sends an ACK PDU if required

protected boolean	**receiveAcknowledgment**(int msgID, int timeout) Receives and checks the ackPDU
protected boolean	**send**(RMPPacket send) Protocol for sending PDUs
protected boolean	**sendAcknowledgment**(RMPPacket toBeAcknowledged) Generates and sends the ackPDU

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

toSend

protected java.net.DatagramPacket **toSend**

---

toReceive

protected java.net.DatagramPacket **toReceive**

---

udp

protected java.net.DatagramSocket **udp**

TX_delay

protected int **TX_delay**

---

reTXT

protected int **reTXT**

---

retransmissionCount

protected int **retransmissionCount**

---

reassemblingTimeoutPeriod

protected int **reassemblingTimeoutPeriod**

encore

protected boolean **encore**

receivingUDPBufferLength

protected int **receivingUDPBufferLength**

receivingUDPBuffer

protected byte[] **receivingUDPBuffer**

msgID

protected int **msgID**

PMTU

protected int **PMTU**

error

protected int **error**

sessionid

protected int **sessionid**

updPort

protected int **updPort**

sending

protected ApplicationMessageAPI **sending**

receiving

protected ApplicationMessageAPI **receiving**

send

```
protected RMPPacket send
```

---

receive

```
protected RMPPacket receive
```

---

counter

```
int counter
```

---

counter2

```
int counter2
```

Constructor Detail

RMP

```
public RMP()
```

Method Detail

send

```
protected boolean send(RMPPacket send)
```

Protocol for sending PDUs

Returns:

-1 if error occurs during the sending process

---

passToUDP

```
protected boolean passToUDP(java.net.DatagramPacket toUDP,
                            boolean rmpMode,
                            boolean RMPType,
                            int msgID,
                            int reTransmissionTimout)
```

Passes the packet to udp. Includes the retransmission loop for RMP level ACKs.

Parameters:

`toUDP` - packet to be passed to UDP

`rmpMode` - State of the RMP mode bit on the packet
`RMPType` - State of the RMP packet type bit
`sessionID` - session identification of the packet
`reTransmissionTimout` - reTransmission timeout

Returns:

True: packet successfully passed to UDP. False: packet was not passed

---

getFromUDP

```
protected boolean getFromUDP(int timeout)
```

Receive DatagramPacket from UDP

Parameters:

`timeout` - How long the UDP socket blocks for receiving

Returns:

True: something was received. False: timeout occurred

---

sendAcknowledgment

```
protected boolean sendAcknowledgment(RMPPacket toBeAcknowledged)
```

Generates and sends the ackPDU

Parameters:

`toBeAcknowledged` -

Returns:

see passToUDP

---

receiveAcknowledgment

```
protected boolean receiveAcknowledgment(int msgID,
                                        int timeout)
```

Receives and checks the ackPDU

Parameters:

`toBeAcknowledged` -
`timeout` -

Returns:

---

receive

```
protected boolean receive()
```

Receives the DatagramPacket, decapsulates it to RMPPacket, and sends an ACK PDU if required

Returns:

True: RMPPacket successfully received and stored in receive field. False: error while receiving or while sending the ACK PDU

---

generateMsgID

```
protected void generateMsgID()
```

Generates the message ID and store it into msgID field

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

rmp

Class RMPClient

```
java.lang.Object
  └ rmp.RMP
      └ rmp.RMPClient
```

All Implemented Interfaces:

RMPClientAPI

```
public class RMPClient
extends RMP
implements RMPClientAPI
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This class holds the RMP level functions specific for the client.

Field Summary

private java.net.InetAddress	**AS_IP**
private int	**AS_PORT**
private int	**deviceCategory**
private int	**deviceID**
private boolean	**isOpen**
private java.net.InetAddress	**RMS_IP**
private int	**RMS_PORT**

Fields inherited from class rmp.RMP

counter, counter2, encore, error, msgID, PMTU, reassemblingTimeoutPeriod, receive, receiving, receivingUDPBuffer, receivingUDPBufferLength, retransmissionCount, reTXT, send, sending, sessionid, toReceive, toSend, TX_delay, udp, updPort

Constructor Summary

RMPClient()

Method Summary

private boolean	**clientReceive**()
private boolean	**clientSend**()
int	**configure**(int deviceCategory, int deviceID, int PMTU, java.net.InetAddress RMS_IP, java.net.InetAddress AS_IP, int RMS_PORT, int AS_PORT, int LOCAL_PORT, int TX_delay, int reTXT, int udpBufferLength, int retransmissionRetryCount, int reassemblingTimeout) Sets the RMP client system parameters
int	**receiveMsg**(ApplicationMessageAPI callback) Non-blocking method for receiving an application message.
int	**receiveMsg**(ApplicationMessageAPI callback, int timeout) Blocking method for receiving an application message.
int	**sendMsg**(ApplicationMessageAPI callback) Non-blocking method for sending an application message.
int	**sendMsg**(ApplicationMessageAPI callback, int timeout) Blocking method for sending an application message.
int	**startPMTU**(ApplicationMessageAPI callback) Non-blocking type: start the PMTU discovery procedure.
int	**startPMTU**(ApplicationMessageAPI callback, int timeout) Blocking type: start the PMTU discovery procedure.

Methods inherited from class rmp.RMP

generateMsgID, getFromUDP, passToUDP, receive, receiveAcknowledgment, send, sendAcknowledgment

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

deviceCategory

```
private int deviceCategory
```

---

deviceID

```
private int deviceID
```

RMS_PORT

```
private int RMS_PORT
```

AS_PORT

```
private int AS_PORT
```

RMS_IP

```
private java.net.InetAddress RMS_IP
```

AS_IP

```
private java.net.InetAddress AS_IP
```

---

isOpen

```
private boolean isOpen
```

Constructor Detail

RMPClient

```
public RMPClient()
```

Method Detail

configure

```
public int configure(int deviceCategory,
                     int deviceID,
                     int PMTU,
                     java.net.InetAddress RMS_IP,
                     java.net.InetAddress AS_IP,
```

```
                    int RMS_PORT,
                    int AS_PORT,
                    int LOCAL_PORT,
                    int TX_delay,
                    int reTXT,
                    int udpBufferLength,
                    int retransmissionRetryCount,
                    int reassemblingTimeout)
```

Description copied from interface: RMPClientAPI
Sets the RMP client system parameters

Specified by:
configure in interface RMPClientAPI

Returns:

---

receiveMsg

```
public int receiveMsg(ApplicationMessageAPI callback,
                      int timeout)
```

Description copied from interface: RMPClientAPI
Blocking method for receiving an application message. Returns if a message is received or after the timeout. RMP shall allocate the memory for the message object. Application shall remove the message object from the memory.

Specified by:
receiveMsg in interface RMPClientAPI

`timeout` - - Defines the period after when the method shall return if no message has been received.

Returns:
- the reference to the received message object

---

receiveMsg

```
public int receiveMsg(ApplicationMessageAPI callback)
```

Description copied from interface: RMPClientAPI
Non-blocking method for receiving an application message. RMP creates a new thread for the receiving process and returns. Once a message is received, RMP calls the msgReceived -callback method to pass the reference of the message object to the application. (callback.msgReceived(msg))

Specified by:
receiveMsg in interface RMPClientAPI

Parameters:
`callback` - - Reference to the application interface.

Returns:

- -1: Error

---

sendMsg

```
public int sendMsg(ApplicationMessageAPI callback,
                   int timeout)
```

Description copied from interface: RMPClientAPI
Blocking method for sending an application message. Returns after the message is passed to UDP. Application shall perform the memory management for msg.

Specified by:
sendMsg in interface RMPClientAPI

Returns:
- -1: Error

---

sendMsg

```
public int sendMsg(ApplicationMessageAPI callback)
```

Description copied from interface: RMPClientAPI
Non-blocking method for sending an application message. RMP creates a new thread for the sending process, copies the message object, and returns.

Specified by:
sendMsg in interface RMPClientAPI

Returns:
- -1: Error

---

startPMTU

```
public int startPMTU(ApplicationMessageAPI callback,
                     int timeout)
```

Description copied from interface: RMPClientAPI
Blocking type: start the PMTU discovery procedure. RMP indicates the application about the procedure status using callback.pathMTUDiscoveryResult(status) -callback method.

Specified by:
startPMTU in interface RMPClientAPI

Parameters:
`callback` - - Reference to the application interface.

---

startPMTU

```
public int startPMTU(ApplicationMessageAPI callback)
```

Description copied from interface: RMPClientAPI
Non-blocking type: start the PMTU discovery procedure. RMP indicates the application about the procedure status using callback.pathMTUDiscoveryResult(status) -callback method.

Specified by:
startPMTU in interface RMPClientAPI

Parameters:
`callback` - - Reference to the application interface.

---

clientSend

```
private boolean clientSend()
```

Parameters:
`callback` -
Returns:

---

clientReceive

```
private boolean clientReceive()
```

---

rmp

Class RMPFragmentationReassembling

```
java.lang.Object
  └ rmp.RMPFragmentationReassembling
```

```
public class RMPFragmentationReassembling
extends java.lang.Object
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This class is used to fragment and reassemble RMP packets

Field Summary

private byte[]	**completeMessage**
private RMPPacket	**completePacket**
private int	**fragCounter**
private int	**fragmentAmount**
private RMPPacket[]	**fragments**
private boolean	**isCompleted**
private boolean	**isFragmented**
private int	**messageDataLength**
private int	**msgID**
private int	**msgType**

private int	**PMTU**
private long	**reassemblingSessionStartTime**
private int	**sessionID**
private int	**sizeOfTheLastFragment**
private InputOutput	**streams**

Constructor Summary

RMPFragmentationReassembling() Constructs an empty object
RMPFragmentationReassembling(RMPPacket completePacket, int PMTU) Constructs an fragmentation object
RMPFragmentationReassembling(RMPPacket frag, int fragAmount, int fragNumber, int msgID, long startTime) Constructs a reassembling object.

Method Summary

void	**calculateFragmentAmount**() calculates the total amount of fragments.
void	**fragmentComplete**() Fragments the complete message into number of fragments defined by the PMTU and the message length.
byte[]	**getCompleteMessage**()
RMPPacket	**getCompletePacket**()
int	**getFragmentAmount**()
RMPPacket[]	**getFragments**()
int	**getMessageDataLength**()
int	**getmsgID**()
int	**getMsgType**()
int	**getPMTU**()

long	**getReassemblingSessionStartTime**()
int	**getSessionID**()
boolean	**hasAllFragments**() Return true if the object has all the fragments of the complete message
void	**initiateTheFragmentArray**(int fragmentAmount) Initiates the length of the fragment array
boolean	**isCompleted**()
boolean	**isFragmented**()
void	**reAssembleFragments**() Reassembles the fragments added into the object.
void	**setCompleted**(boolean isCompleted)
void	**setCompleteMessage**(byte[] completeMessage) set also the message data length
void	**setCompletePacket**(RMPPacket completePacket)
void	**setFragmentAmount**(int fragmentAmount)
void	**setFragmented**(boolean isFragmented)
void	**setFragments**(RMPPacket[] fragments)
void	**setMessageDataLength**(int messageDataLength)
void	**setmsgID**(int msgID)
void	**setMsgType**(int msgType)
void	**setNewFragment**(RMPPacket frag, int fragNumber) Add a fragment into the fragment array to the given position i
void	**setPMTU**(int pMTU)
void	**setReassemblingSessionStartTime**(long reassemblingSessionStartTime)
void	**setSessionID**(int sessionID)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

completePacket

private RMPPacket **completePacket**

---

completeMessage

private byte[] **completeMessage**

---

fragments

private RMPPacket[] **fragments**

---

isFragmented

private boolean **isFragmented**

---

isCompleted

private boolean **isCompleted**

---

messageDataLength

private int **messageDataLength**

PMTU

private int **PMTU**

---

fragmentAmount

private int **fragmentAmount**

---

sizeOfTheLastFragment

```
private int sizeOfTheLastFragment
```

---

streams

```
private InputOutput streams
```

---

sessionID

```
private int sessionID
```

---

msgID

```
private int msgID
```

---

reassemblingSessionStartTime

```
private long reassemblingSessionStartTime
```

---

fragCounter

```
private int fragCounter
```

---

msgType

```
private int msgType
```

Constructor Detail

RMPFragmentationReassembling

```
public RMPFragmentationReassembling()
```

Constructs an empty object

RMPFragmentationReassembling

```
public RMPFragmentationReassembling(RMPPacket completePacket,
                                    int PMTU)
```

Constructs an fragmentation object

Parameters:

`complete` - message to be fragmented

RMPFragmentationReassembling

```
public RMPFragmentationReassembling(RMPPacket frag,
                                    int fragAmount,
                                    int fragNumber,
                                    int msgID,
                                    long startTime)
```

Constructs a reassembling object. Initiates the fragment array based on the value of fragmentAmount

Parameters:

`frag` - first received fragment
`i` - fragment number

Method Detail

initiateTheFragmentArray

```
public void initiateTheFragmentArray(int fragmentAmount)
```

Initiates the length of the fragment array

Parameters:

`fragmentAmount` - length of the array

---

getCompletePacket

```
public RMPPacket getCompletePacket()
```

Returns:

the completePacket

---

setCompletePacket

```
public void setCompletePacket(RMPPacket completePacket)
```

Parameters:

`completePacket` - the completePacket to set

---

getCompleteMessage

```
public byte[] getCompleteMessage()
```

Returns:
the completeMessage

---

setCompleteMessage

```
public void setCompleteMessage(byte[] completeMessage)
```

set also the message data length

Parameters:
`completeMessage` - the completeMessage to set

---

getFragments

```
public RMPPacket[] getFragments()
```

Returns:
the fragments

---

setFragments

```
public void setFragments(RMPPacket[] fragments)
```

Parameters:
`fragments` - the fragments to set

---

getMessageDataLength

```
public int getMessageDataLength()
```

Returns:
the messageDataLength

---

setMessageDataLength

```
public void setMessageDataLength(int messageDataLength)
```

Parameters:
`messageDataLength` - the messageDataLength to set

---

getPMTU

```
public int getPMTU()
```

Returns:

the pMTU

---

setPMTU

```
public void setPMTU(int pMTU)
```

Parameters:
`pMTU` - the pMTU to set

---

getFragmentAmount

```
public int getFragmentAmount()
```

Returns:
the fragmentAmount

---

setFragmentAmount

```
public void setFragmentAmount(int fragmentAmount)
```

Parameters:
`fragmentAmount` - the fragmentAmount to set

---

getSessionID

```
public int getSessionID()
```

Returns:
the sessionID

---

setSessionID

```
public void setSessionID(int sessionID)
```

Parameters:
`sessionID` - the sessionID to set

---

getmsgID

```
public int getmsgID()
```

Returns:
the reassemblingID

---

setmsgID

```
public void setmsgID(int msgID)
```

Parameters:

`reassemblingID` - the reassemblingID to set

---

getReassemblingSessionStartTime

```
public long getReassemblingSessionStartTime()
```

Returns:

the reassemblingSessionStartTime

---

setReassemblingSessionStartTime

```
public void setReassemblingSessionStartTime(long reassemblingSessionStartTime)
```

Parameters:

`reassemblingSessionStartTime` - the reassemblingSessionStartTime to set

---

setNewFragment

```
public void setNewFragment(RMPPacket frag,
                           int fragNumber)
```

Add a fragment into the fragment array to the given position i

Parameters:

`frag` -

`i` -

---

getMsgType

```
public int getMsgType()
```

Returns:

the msgType

---

setMsgType

```
public void setMsgType(int msgType)
```

Parameters:

`msgType` - the msgType to set

---

hasAllFragments

```
public boolean hasAllFragments()
```

Return true if the object has all the fragments of the complete message

Returns:
True: All fragments False: some or all the fragments are missing

---

isFragmented

```
public boolean isFragmented()
```

Returns:
the isFragmented

---

setFragmented

```
public void setFragmented(boolean isFragmented)
```

Parameters:
`isFragmented` - the isFragmented to set

---

isCompleted

```
public boolean isCompleted()
```

Returns:
the isCompleted

---

setCompleted

```
public void setCompleted(boolean isCompleted)
```

Parameters:
`isCompleted` - the isCompleted to set

---

calculateFragmentAmount

```
public void calculateFragmentAmount()
```

calculates the total amount of fragments. PMTU and MessgeDataLength must be set before hand

---

fragmentComplete

`public void` **`fragmentComplete()`**

Fragments the complete message into number of fragments defined by the PMTU and the message length. Creates a new RMPFragment object with corresponding fragment data (fragamount, fragnumber) for each fragment

---

reAssembleFragments

`public void` **`reAssembleFragments()`**

Reassembles the fragments added into the object. It must be sure that all the fragments have been added into the fragments array

---

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

rmp

Class RMPPacket

```
java.lang.Object
  └ rmp.RMPPacket
```

```
public class RMPPacket
extends java.lang.Object
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This class holds the methods and data of RMP packet.

Field Summary

private int	deviceCategory
private int	deviceID
private int	flag
private int	fragmentAmount
private int	fragmentNumber
private java.net.InetAddress	IP
private int	msgDataLength
private int	msgID
private int	msgType
private byte[]	payload

private int	**reassemblingID**
private int	**sessionID**
private InputOutput	**streams**
private int	**udpPort**

Constructor Summary

RMPPacket()
constructs an empty fragment object

RMPPacket(int deviceCategory, boolean AF, boolean RACK, int msgType, int msgID, int deviceID, byte[] payload, java.net.InetAddress iP, int udpPort)
Cosnstructs a RMPPacket from given parameters.

RMPPacket(int deviceCategory, boolean RACK, int msgType, int msgID, int deviceID, byte[] payload, java.net.InetAddress iP, int udpPort)
Cosnstructs a RMPPacket from given parameters.

RMPPacket(int deviceCategory, int flag, int msgType, int msgID, int deviceID, int msgDataLength, int fragmentAmount, int fragmentNumber)
Constructs a header object

RMPPacket(int deviceCategory, int flag, int msgType, int msgID, int deviceID, int msgDataLength, int fragmentAmount, int fragmentNumber, byte[] payload, java.net.InetAddress IP, int udpPort)
constructs a fragment object Constructs a RMPPacket from the given parameters

RMPPacket(int deviceCategory, int flag, int msgType, int msgID, int deviceID, int msgDataLength, int fragmentAmount, int fragmentNumber, byte[] payload, java.net.InetAddress IP, int udpPort, int sessionID)
Constructs a RMPPacket with all the field values

RMPPacket(int deviceCategory, int flag, int msgType, int msgID, int deviceID, int msgDataLength, int fragmentAmount, int fragmentNumber, java.net.InetAddress iP, int udpPort)
Constructs a header object

Method Summary

RMPPacket	**clone**() Creates a new instance from the object
RMPPacket	**cloneHeader**() Creates a new header instance from the object
void	**generateReassemblingID**() Generates a session ID for the packet
void	**generateSessionID**() Generates a reassembling ID for the packet

boolean	**getAF**() Gets the Action Flag bit from the RMP header
int	**getDeviceCategory**()
int	**getDeviceID**()
int	**getFlag**()
int	**getFragmentAmount**()
int	**getFragmentNumber**()
java.net.InetAddress	**getIP**()
int	**getMsgDataLength**()
int	**getMsgID**()
int	**getMsgType**()
byte[]	**getPayload**()
boolean	**getPDUType**() Gets the status PDU type flag from the RMP header
int	**getReassemblingID**()
boolean	**getRMPMode**() Gets the status ACK required bit from the RMP header
int	**getSessionID**()
int	**getUdpPort**()
private boolean	**readBit**(int flag, int bitPos) Read the bit from the flag byte in a given position.
void	**setAF**(boolean mode) Sets the Action Flag bit in the RMP header
private int	**setBit**(int flag, int bitPos, boolean stat) Sets or clears a bit in a byte
void	**setDeviceCategory**(int deviceCategory)
void	**setDeviceID**(int deviceID)

void	**setFlag**(int flag)
void	**setFragmentAmount**(int fragmentAmount)
void	**setFragmentNumber**(int fragmentNumber)
void	**setIP**(java.net.InetAddress iP)
void	**setMsgDataLength**(int msgDataLength)
void	**setMsgID**(int msgID)
void	**setMsgType**(int msgType)
void	**setPayload**(byte[] payload)
void	**setPDUType**(boolean mode) Sets the PDU type flag in the RMP header
void	**setRMPMode**(boolean mode) Sets the ACK required bit in the RMP header
void	**setUdpPort**(int udpPort)

Methods inherited from class java.lang.Object
equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

deviceCategory

private int **deviceCategory**

---

flag

private int **flag**

---

msgType

private int **msgType**

---

msgID

```
private int msgID
```

---

deviceID

```
private int deviceID
```

---

msgDataLength

```
private int msgDataLength
```

---

fragmentAmount

```
private int fragmentAmount
```

---

fragmentNumber

```
private int fragmentNumber
```

---

payload

```
private byte[] payload
```

IP

```
private java.net.InetAddress IP
```

---

udpPort

```
private int udpPort
```

---

sessionID

```
private int sessionID
```

---

reassemblingID

```
private int reassemblingID
```

---

streams

```
private InputOutput streams
```

Constructor Detail

RMPPacket

```
public RMPPacket()
```

constructs an empty fragment object

RMPPacket

```
public RMPPacket(int deviceCategory,
                 boolean RACK,
                 int msgType,
                 int msgID,
                 int deviceID,
                 byte[] payload,
                 java.net.InetAddress iP,
                 int udpPort)
```

Cosnstructs a RMPPacket from given parameters. used at the client end

Parameters:

deviceCategory -
RACK -
msgType -
msgID -
deviceID -
payload -
iP -
udpPort -

RMPPacket

```
public RMPPacket(int deviceCategory,
                 boolean AF,
                 boolean RACK,
                 int msgType,
                 int msgID,
                 int deviceID,
                 byte[] payload,
                 java.net.InetAddress iP,
                 int udpPort)
```

Cosnstructs a RMPPacket from given parameters. used at the server end

Parameters:

deviceCategory -
AF -
msgType -
msgID -
deviceID -
payload -
iP -
udpPort -

RMPPacket

```
public RMPPacket(int deviceCategory,
                 int flag,
                 int msgType,
                 int msgID,
                 int deviceID,
                 int msgDataLength,
                 int fragmentAmount,
                 int fragmentNumber,
                 java.net.InetAddress iP,
                 int udpPort)
```

Constructs a header object

Parameters:

deviceCategory -
flag -
msgType -
msgID -
deviceID -
msgDataLength -
fragmentAmount -
fragmentNumber -
payload -
iP -
udpPort -

RMPPacket

```
public RMPPacket(int deviceCategory,
                 int flag,
                 int msgType,
                 int msgID,
                 int deviceID,
                 int msgDataLength,
                 int fragmentAmount,
                 int fragmentNumber,
                 byte[] payload,
                 java.net.InetAddress IP,
                 int udpPort,
                 int sessionID)
```

Constructs a RMPPacket with all the field values

Parameters:

deviceCategory -
flag -
msgType -
msgID -
deviceID -
msgDataLength -
fragmentAmount -
fragmentNumber -
payload -
iP -
udpPort -
sessionID -

RMPPacket

```
public RMPPacket(int deviceCategory,
                 int flag,
                 int msgType,
                 int msgID,
                 int deviceID,
                 int msgDataLength,
                 int fragmentAmount,
                 int fragmentNumber,
                 byte[] payload,
                 java.net.InetAddress IP,
                 int udpPort)
```

constructs a fragment object Constructs a RMPPacket from the given parameters

Parameters:

deviceCategory -
flag -
msgType -
msgID -
deviceID -
msgDataLength -
fragmentAmount -
fragmentNumber -
payload -
iP -
udpPort -
sessionID -

RMPPacket

```
public RMPPacket(int deviceCategory,
                 int flag,
                 int msgType,
```

```
                int msgID,
                int deviceID,
                int msgDataLength,
                int fragmentAmount,
                int fragmentNumber)
```

Constructs a header object

Parameters:

`deviceCategory` -
`flag` -
`msgType` -
`msgID` -
`deviceID` -
`msgDataLength` -
`fragmentAmount` -
`fragmentNumber` -
`payload` -

Method Detail

getDeviceCategory

```
public int getDeviceCategory()
```

Returns:
the deviceCategory

---

setDeviceCategory

```
public void setDeviceCategory(int deviceCategory)
```

Parameters:
`deviceCategory` - the deviceCategory to set

---

getFlag

```
public int getFlag()
```

Returns:
the flag

---

setFlag

```
public void setFlag(int flag)
```

Parameters:
`flag` - the flag to set

---

setPDUType

```
public void setPDUType(boolean mode)
```

Sets the PDU type flag in the RMP header

Parameters:
`flag` -

---

getPDUType

```
public boolean getPDUType()
```

Gets the status PDU type flag from the RMP header

Returns:
True: ack PDU, False: data PDU

---

setRMPMode

```
public void setRMPMode(boolean mode)
```

Sets the ACK required bit in the RMP header

Parameters:
`True:` - RMP acknowledged mode, False: RMP unacknowledged mode

---

getRMPMode

```
public boolean getRMPMode()
```

Gets the status ACK required bit from the RMP header

Returns:
True: RMP acknowledged mode, False: RMP unacknowledged mode

---

setAF

```
public void setAF(boolean mode)
```

Sets the Action Flag bit in the RMP header

Parameters:
`True:` - AF set, False: AF cleared

---

getAF

```
public boolean getAF()
```

Gets the Action Flag bit from the RMP header

Returns:
True: action Flag set, False: action flag clearded

---

getMsgType

```
public int getMsgType()
```

Returns:
the msgType

---

setMsgType

```
public void setMsgType(int msgType)
```

Parameters:
`msgType` - the msgType to set

---

getMsgID

```
public int getMsgID()
```

Returns:
the msgID

---

setMsgID

```
public void setMsgID(int msgID)
```

Parameters:
`msgID` - the msgID to set

---

getDeviceID

```
public int getDeviceID()
```

Returns:
the deviceID

---

setDeviceID

```
public void setDeviceID(int deviceID)
```

Parameters:
`deviceID` - the deviceID to set

---

getMsgDataLength

```
public int getMsgDataLength()
```

Returns:
the msgDataLength

---

setMsgDataLength

```
public void setMsgDataLength(int msgDataLength)
```

Parameters:
`msgDataLength` - the msgDataLength to set

---

getFragmentAmount

```
public int getFragmentAmount()
```

Returns:
the fragmentAmount

---

setFragmentAmount

```
public void setFragmentAmount(int fragmentAmount)
```

Parameters:
`fragmentAmount` - the fragmentAmount to set

---

getFragmentNumber

```
public int getFragmentNumber()
```

Returns:
the fragmentNumber

---

setFragmentNumber

```
public void setFragmentNumber(int fragmentNumber)
```

Parameters:
`fragmentNumber` - the fragmentNumber to set

---

getPayload

```
public byte[] getPayload()
```

Returns:
the fragment

---

setPayload

```
public void setPayload(byte[] payload)
```

Parameters:
`fragment` - the fragment to set

---

getIP

```
public java.net.InetAddress getIP()
```

Returns:
the iP

---

setIP

```
public void setIP(java.net.InetAddress iP)
```

Parameters:
`iP` - the iP to set

---

getUdpPort

```
public int getUdpPort()
```

Returns:
the udpPort

---

setUdpPort

```
public void setUdpPort(int udpPort)
```

Parameters:
`udpPort` - the udpPort to set

---

getSessionID

```
public int getSessionID()
```

Returns:
the sessionID

---

generateSessionID

```
public void generateSessionID()
```

Generates a reassembling ID for the packet

---

getReassemblingID

```
public int getReassemblingID()
```

Returns:
the reassemblingID

---

generateReassemblingID

```
public void generateReassemblingID()
```

Generates a session ID for the packet

---

clone

```
public RMPPacket clone()
```

Creates a new instance from the object

Overrides:
`clone` in class `java.lang.Object`

---

cloneHeader

```
public RMPPacket cloneHeader()
```

Creates a new header instance from the object

---

setBit

```
private int setBit(int flag,
                   int bitPos,
```

```
                boolean stat)
```

Sets or clears a bit in a byte

Parameters:

`flag` - byte where the bit is to be set or cleared
`bitPos` - Defines the position of the bit in flag byte
`stat` - True: set the bit, False: clear the bit

Returns:

---

readBit

```
private boolean readBit(int flag,
                        int bitPos)
```

Read the bit from the flag byte in a given position.

Parameters:

`flag` - the byte from where the bit is to be read
`bitPos` - defines the position of the bit in the flag byte

Returns:

True: the bit is set, False: the bit is cleared

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

rmp

Class RMPPacketEncapsulationToDatagramPacket

```
java.lang.Object
  └ rmp.RMPPacketEncapsulationToDatagramPacket
```

```
public class RMPPacketEncapsulationToDatagramPacket
extends java.lang.Object
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This class holds the methods to encode and decode RMP messages to/from UDP packets

Field Summary

private ApplicationMessage	**appMsg**
private RMPPacket	**rmpPacket**
private InputOutput	**streams**
private java.net.DatagramPacket	**udpPacket**

Constructor Summary

RMPPacketEncapsulationToDatagramPacket() Constructs an empty encapsulation object
RMPPacketEncapsulationToDatagramPacket(ApplicationMessageAPI appMsg) Constructs a encapsulation object from ApplicationMessage to RMPPacket
RMPPacketEncapsulationToDatagramPacket(java.net.DatagramPacket udpPacket) Constructs a decapsulation object from DatagramPacket to RMPPacket
RMPPacketEncapsulationToDatagramPacket(RMPPacket rmpPacket) Constructs an encapsulation object from RMPPacket to DatagramPacket or decapsulation object from RMPPacket to ApplicationMessage

Method Summary

ApplicationMessageAPI	**getAppMsg**()
RMPPacket	**getRmpPacket**()
InputOutput	**getStreams**()
java.net.DatagramPacket	**getUdpPacket**()
void	**setAppMsg**(ApplicationMessageAPI appMsg)
void	**setRmpPacket**(RMPPacket rmpPacket)
void	**setStreams**(InputOutput streams)
void	**setUdpPacket**(java.net.DatagramPacket udpPacket)
void	**toApplicationMessageFromRMPPacket**() Decapsulates the RMPPacket to ApplicationMessage
void	**toRMPPacketFromApplicationMessage**() Encapsulate the ApplicationMessage into the RMPPacket
void	**toRMPPacketFromUDPPacket**() Decapsultes the RMPPacket from the DatagramPacket.
void	**toUDPPacketFromRMPPacket**() Encapsulates the RMPPacket into a DatagramPacket

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

udpPacket

private java.net.DatagramPacket **udpPacket**

---

rmpPacket

private RMPPacket **rmpPacket**

appMsg

private ApplicationMessage **appMsg**

streams

private InputOutput **streams**

Constructor Detail

RMPPacketEncapsulationToDatagramPacket

public **RMPPacketEncapsulationToDatagramPacket**()

Constructs an empty encapsulation object

RMPPacketEncapsulationToDatagramPacket

public **RMPPacketEncapsulationToDatagramPacket**(RMPPacket rmpPacket)

Constructs an encapsulation object from RMPPacket to DatagramPacket or decapsulation object from RMPPacket to ApplicationMessage

Parameters:

rmpPacket -

RMPPacketEncapsulationToDatagramPacket

public **RMPPacketEncapsulationToDatagramPacket**(java.net.DatagramPacket udpPacket)

Constructs a decapsulation object from DatagramPacket to RMPPacket

Parameters:

udpPacket -

RMPPacketEncapsulationToDatagramPacket

public **RMPPacketEncapsulationToDatagramPacket**(ApplicationMessageAPI appMsg)

Constructs a encapsulation object from ApplicationMessage to RMPPacket

Parameters:

appMsg -

Method Detail

getUdpPacket

```
public java.net.DatagramPacket getUdpPacket()
```

Returns:
the udpPacket

---

setUdpPacket

```
public void setUdpPacket(java.net.DatagramPacket udpPacket)
```

Parameters:
`udpPacket` - the udpPacket to set

---

getRmpPacket

```
public RMPPacket getRmpPacket()
```

Returns:
the rmpPacket

---

setRmpPacket

```
public void setRmpPacket(RMPPacket rmpPacket)
```

Parameters:
`rmpPacket` - the rmpPacket to set

---

getAppMsg

```
public ApplicationMessageAPI getAppMsg()
```

Returns:
the appMsg

---

setAppMsg

```
public void setAppMsg(ApplicationMessageAPI appMsg)
```

Parameters:
`appMsg` - the appMsg to set

---

getStreams

```
public InputOutput getStreams()
```

Returns:
the streams

---

setStreams

```
public void setStreams(InputOutput streams)
```

Parameters:
`streams` - the streams to set

---

toUDPPacketFromRMPPacket

```
public void toUDPPacketFromRMPPacket()
```

Encapsulates the RMPPacket into a DatagramPacket

---

toRMPPacketFromUDPPacket

```
public void toRMPPacketFromUDPPacket()
```

Decapsultes the RMPPacket from the DatagramPacket.

---

toRMPPacketFromApplicationMessage

```
public void toRMPPacketFromApplicationMessage()
```

Encapsulate the ApplicationMessage into the RMPPacket

---

toApplicationMessageFromRMPPacket

```
public void toApplicationMessageFromRMPPacket()
```

Decapsulates the RMPPacket to ApplicationMessage

---

rmp

Class RMPServer

```
java.lang.Object
  └ rmp.RMP
      └ rmp.RMPServer
```

All Implemented Interfaces:

RMPServerAPI

```
public class RMPServer
extends RMP
implements RMPServerAPI
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This class holds the methods of RMP specific for the server.

Field Summary

private int[]	**idTable**
private int	**localUDPPort**
private RMPFragmentationReassembling[]	**reass**
private int	**reassembleCounter**
private java.util.ArrayList	**reAssemblingList**
private java.net.DatagramPacket	**servReceive**
private RMPPacket	**servReceived**
private byte[]	**UDPBuffer**

Fields inherited from class rmp.RMP
counter, counter2, encore, error, msgID, PMTU, reassemblingTimeoutPeriod, receive, receiving, receivingUDPBuffer, receivingUDPBufferLength, retransmissionCount, reTXT, send, sending, sessionid, toReceive, toSend, TX_delay, udp, updPort

Constructor Summary

RMPServer()

Method Summary

int	**receiveMsg**(ApplicationMessageAPI receiveAppMsg) Non-blocking method for receiving an application message.
int	**sendMsg**(ApplicationMessageAPI sendAppMsg) Non-blocking method for sending an application message.
private boolean	**serverReceive**() Server receiving protocol
private boolean	**serverSend**()
int	**setParameters**(int PMTU, int RMS_PORT, int TX_delay, int reTXT, int udpBufferLength, int reTransmisssionRetryCount, int reassemblingTimeout) Sets the RMP server parameters.

Methods inherited from class rmp.RMP
generateMsgID, getFromUDP, passToUDP, receive, receiveAcknowledgment, send, sendAcknowledgment

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

localUDPPort

private int **localUDPPort**

---

reass

private RMPFragmentationReassembling[] **reass**

---

idTable

```
private int[] idTable
```

---

reassembleCounter

```
private int reassembleCounter
```

UDPBuffer

```
private byte[] UDPBuffer
```

---

servReceive

```
private java.net.DatagramPacket servReceive
```

---

servReceived

```
private RMPPacket servReceived
```

---

reAssemblingList

```
private java.util.ArrayList reAssemblingList
```

Constructor Detail

RMPServer

```
public RMPServer()
```

Method Detail

receiveMsg

```
public int receiveMsg(ApplicationMessageAPI receiveAppMsg)
```

Description copied from interface: RMPServerAPI
Non-blocking method for receiving an application message. RMP creates a new thread for the receiving process and returns. Once a message is received, RMP calls the msgReceived -callback method to pass the reference of the message object to the application. (callback.msgReceived(msg))

Specified by:
receiveMsg in interface RMPServerAPI

Returns:
-1 if error occurs and 0: for successful sending

---

sendMsg

```
public int sendMsg(ApplicationMessageAPI sendAppMsg)
```

Description copied from interface: RMPServerAPI
Non-blocking method for sending an application message. RMP creates a new thread for the sending process, copies the message object, and returns.

Specified by:
sendMsg in interface RMPServerAPI

Parameters:
sendAppMsg - object that holds the application message data to be sent
Returns:
-1 if error occurs and 0: for successful sending

---

setParameters

```
public int setParameters(int PMTU,
                         int RMS_PORT,
                         int TX_delay,
                         int reTXT,
                         int udpBufferLength,
                         int reTransmisssionRetryCount,
                         int reassemblingTimeout)
```

Description copied from interface: RMPServerAPI
Sets the RMP server parameters.

Specified by:
setParameters in interface RMPServerAPI

Returns:
-1 if error occurs and 0: for successful sending

---

serverReceive

```
private boolean serverReceive()
```

Server receiving protocol

Returns:
Error code

serverSend

```
private boolean serverSend()
```

Overview **Package** Class **Use** **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

rmp

Class ServerAppDemo

```
java.lang.Object
  └ rmp.ServerAppDemo
```

```
public class ServerAppDemo
extends java.lang.Object
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This is the RMP server application demonstrations. The server can receive client messages and set some actions such as re-configurations of RMS IP-address information.

Constructor Summary

ServerAppDemo()

Method Summary

static void	**main**(java.lang.String[] args)
private static void	**printReceivedmsg**(ServerAppGUI gui, ApplicationMessageAPI receiving)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ServerAppDemo

```
public ServerAppDemo()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Parameters:

args -

---

printReceivedmsg

```
private static void printReceivedmsg(ServerAppGUI gui,
                                     ApplicationMessageAPI receiving)
```

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

rmp

Class ServerAppGUI

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
                  └javax.swing.JFrame
                      └rmp.ServerAppGUI
```

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

```
public class ServerAppGUI
extends javax.swing.JFrame
```

Author:

Antti Siirilä
0602137
Turku University of Applied Sciences
antti.siirila@mbnet.fi
+358 45 1396013 This is the user interface of the RMP Server Demo GUI shows the received messages and allows the user to set some actions for the clients.

See Also:

Serialized Form

Nested Class Summary

class	**ServerAppGUI.EventListener** Private inner class that handles the event when the user clicks the Calculate button.

Nested classes/interfaces inherited from class javax.swing.JFrame
javax.swing.JFrame.AccessibleJFrame

Nested classes/interfaces inherited from class java.awt.Frame
java.awt.Frame.AccessibleAWTFrame

Nested classes/interfaces inherited from class java.awt.Window

java.awt.Window.AccessibleAWTWindow

Nested classes/interfaces inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BaselineResizeBehavior, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

private static int	**ACTION**
private static int	**ACTION_REQUEST**
private static int	**ACTION_RESPONSE**
private javax.swing.JPanel	**actionPanel**
private javax.swing.JTextField[]	**actionPanelInput**
private javax.swing.ButtonGroup	**actions**
private javax.swing.JRadioButton[]	**actionTypes**
private boolean	**AF**
private java.net.InetAddress	**AS_IP**
private int	**AS_PORT**
private static int	**AUTHENTICATING**
private static int	**BOM_REQUEST**
private static int	**BOM_RESPONSE**
private javax.swing.JPanel	**buttonPanel**
private javax.swing.JButton[]	**buttonPanelB**

javax.swing.JTextArea	**cipherText1**
javax.swing.JTextArea	**cipherText2**
private ServerAppGUI.EventListener	**clientListener**
private static int	**CONFIGURATION**
private static int	**CONFIGURATION_REQUEST**
private static int	**CONFIGURATION_RESPONSE**
private javax.swing.JButton[]	**configurationButton**
private javax.swing.JPanel	**configurationPanel**
private static int	**CONFIGURING**
private int	**deviceCategory**
private int	**deviceID**
private static int	**DISABLED**
private static int	**FAULT_OCCURRED_REQUEST**
private static int	**FAULT_OCCURRED_RESPONSE**
private static int	**FAULT_UPDATE_REQUEST**
private static int	**FAULT_UPDATE_RESPONSE**
private static int	**GO_TO_IDLE**
private static int	**IDLE**
private byte[]	**ip**
private int	**LOCAL_PORT**
private java.lang.String	**msg**

private int	**msgType**
private javax.swing.JTextField[]	**parameterInput**
private javax.swing.JLabel[]	**parameters**
private int	**PMTU**
private javax.swing.JCheckBox	**RACK**
private int	**reassemblingTimeout**
private static int	**RECEIVE**
private static int	**RECEIVED**
private static int	**RECEIVING**
private static int	**REGISTERING**
private static int	**REGISTRATION_REQUEST**
private static int	**REGISTRATION_RESPONSE**
private int	**reTransmisssionRetryCount**
private int	**reTX_T**
private boolean	**RMPMode**
private java.net.InetAddress	**RMS_IP**
private int	**RMS_PORT**
private javax.swing.JScrollPane	**scrollingArea1**
private javax.swing.JScrollPane	**scrollingArea2**
private static int	**SEND**
private static int	**SENDING**

private javax.swing.JButton[]	**set**
private int	**State**
private static int	**STATISTICS_REPORT_REQUEST**
private static int	**STATISTICS_REPORT_RESPONSE**
private javax.swing.JTabbedPane	**systems**
private javax.swing.JPanel	**TextArea**
private int	**TX_delay**
private int	**udpBufferLength**

Fields inherited from class javax.swing.JFrame
accessibleContext, EXIT_ON_CLOSE, rootPane, rootPaneCheckingEnabled

Fields inherited from class java.awt.Frame
CROSSHAIR_CURSOR, DEFAULT_CURSOR, E_RESIZE_CURSOR, HAND_CURSOR, ICONIFIED, MAXIMIZED_BOTH, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MOVE_CURSOR, N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR, SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR, W_RESIZE_CURSOR, WAIT_CURSOR

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface javax.swing.WindowConstants
DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

ServerAppGUI()
Constructs the client GUI

Method Summary

private void	**buildActionPanel**() build the Server Action Panel
private void	**buildButtonPanel**() The buildButtonPanel method builds the button panel.
private void	**buildConfigurationPanel**() Build the server configuration panel with required input fields
private void	**buildTabPane**() build the Tab pane and add the panel into it
private void	**buildTextArea**() Build the text areas
java.net.InetAddress	**getAS_IP**()
int	**getAS_PORT**()
int	**getDeviceCategory**()
int	**getDeviceID**()
byte[]	**getIp**()
int	**getLOCAL_PORT**()
java.lang.String	**getMsg**()
int	**getMsgType**()
int	**getPMTU**()
int	**getReassemblingTimeout**()
int	**getReTransmisssionRetryCount**()
int	**getReTX_T**()
java.net.InetAddress	**getRMS_IP**()
int	**getRMS_PORT**()
int	**getState**() Synchronized method to get the client state
int	**getTX_delay**()

int	**getUdpBufferLength**()
boolean	**isAF**()
boolean	**isRMPMode**()
void	**setAF**(boolean aF)
void	**setAS_IP**(java.net.InetAddress aSIP)
void	**setAS_PORT**(int aSPORT)
void	**setDeviceCategory**(int deviceCategory)
void	**setDeviceID**(int deviceID)
void	**setIp**(byte[] ip)
void	**setLOCAL_PORT**(int lOCALPORT)
void	**setMsg**(java.lang.String msg)
void	**setMsgType**(int msgType)
void	**setPMTU**(int pMTU)
void	**setReassemblingTimeout**(int reassemblingTimeout)
void	**setReTransmisssionRetryCount**(int reTransmisssionRetryCount)
void	**setReTX_T**(int reTXT)
void	**setRMPMode**(boolean rMPMode)
void	**setRMS_IP**(java.net.InetAddress rMSIP)
void	**setRMS_PORT**(int rMSPORT)
void	**setState**(int State) Synchronized method to set the client state

void	**setTX_delay**(int tXDelay)
void	**setUdpBufferLength**(int udpBufferLength)

Methods inherited from class javax.swing.JFrame

addImpl, createRootPane, frameInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getGraphics, getJMenuBar, getLayeredPane, getRootPane, getTransferHandler, isDefaultLookAndFeelDecorated, isRootPaneCheckingEnabled, paramString, processWindowEvent, remove, repaint, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setIconImage, setJMenuBar, setLayeredPane, setLayout, setRootPane, setRootPaneCheckingEnabled, setTransferHandler, update

Methods inherited from class java.awt.Frame

addNotify, getCursorType, getExtendedState, getFrames, getIconImage, getMaximizedBounds, getMenuBar, getTitle, isResizable, isUndecorated, remove, removeNotify, setCursor, setExtendedState, setMaximizedBounds, setMenuBar, setResizable, setTitle, setUndecorated

Methods inherited from class java.awt.Window

addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getGraphicsConfiguration, getIconImages, getInputContext, getListeners, getLocale, getModalExclusionType, getMostRecentFocusOwner, getOwnedWindows, getOwner, getOwnerlessWindows, getToolkit, getWarningString, getWindowFocusListeners, getWindowListeners, getWindows, getWindowStateListeners, hide, isActive, isAlwaysOnTop, isAlwaysOnTopSupported, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isShowing, pack, paint, postEvent, processEvent, processWindowFocusEvent, processWindowStateEvent, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, reshape, setAlwaysOnTop, setBounds, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setIconImages, setLocationByPlatform, setLocationRelativeTo, setMinimumSize, setModalExclusionType, setSize, setSize, setVisible, show, toBack, toFront

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component
action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBaseline, getBaselineResizeBehavior, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isOpaque, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resize, resize, setBackground, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setLocation, setLocation, setMaximumSize, setName, setPreferredSize, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.awt.MenuContainer
getFont, postEvent

Field Detail

deviceCategory

private int **deviceCategory**

---

deviceID

```
private int deviceID
```

PMTU

```
private int PMTU
```

RMS_IP

```
private java.net.InetAddress RMS_IP
```

AS_IP

```
private java.net.InetAddress AS_IP
```

---

ip

```
private byte[] ip
```

RMS_PORT

```
private int RMS_PORT
```

AS_PORT

```
private int AS_PORT
```

LOCAL_PORT

```
private int LOCAL_PORT
```

TX_delay

```
private int TX_delay
```

---

reTX_T

```
private int reTX_T
```

---

udpBufferLength

```
private int udpBufferLength
```

---

reTransmisssionRetryCount

```
private int reTransmisssionRetryCount
```

---

reassemblingTimeout

```
private int reassemblingTimeout
```

State

```
private int State
```

---

msgType

```
private int msgType
```

---

msg

```
private java.lang.String msg
```

RMPMode

```
private boolean RMPMode
```

AF

```
private boolean AF
```

DISABLED

```
private static final int DISABLED
```

See Also:

Constant Field Values

AUTHENTICATING

```
private static final int AUTHENTICATING
```

See Also:
Constant Field Values

REGISTERING

```
private static final int REGISTERING
```

See Also:
Constant Field Values

IDLE

```
private static final int IDLE
```

See Also:
Constant Field Values

SENDING

```
private static final int SENDING
```

See Also:
Constant Field Values

RECEIVING

```
private static final int RECEIVING
```

See Also:
Constant Field Values

CONFIGURING

```
private static final int CONFIGURING
```

See Also:
Constant Field Values

GO_TO_IDLE

```
private static final int GO_TO_IDLE
```

See Also:
Constant Field Values

RECEIVED

```
private static final int RECEIVED
```

See Also:
Constant Field Values

RECEIVE

```
private static final int RECEIVE
```

See Also:
Constant Field Values

ACTION

```
private static final int ACTION
```

See Also:
Constant Field Values

SEND

```
private static final int SEND
```

See Also:
Constant Field Values

REGISTRATION_REQUEST

```
private static final int REGISTRATION_REQUEST
```

See Also:
Constant Field Values

CONFIGURATION_REQUEST

```
private static final int CONFIGURATION_REQUEST
```

See Also:
Constant Field Values

FAULT_OCCURRED_REQUEST

```
private static final int FAULT_OCCURRED_REQUEST
```

See Also:
Constant Field Values

FAULT_UPDATE_REQUEST

```
private static final int FAULT_UPDATE_REQUEST
```

See Also:
Constant Field Values

STATISTICS_REPORT_REQUEST

```
private static final int STATISTICS_REPORT_REQUEST
```

See Also:
Constant Field Values

ACTION_REQUEST

```
private static final int ACTION_REQUEST
```

See Also:
Constant Field Values

BOM_REQUEST

```
private static final int BOM_REQUEST
```

See Also:
Constant Field Values

REGISTRATION_RESPONSE

```
private static final int REGISTRATION_RESPONSE
```

See Also:
Constant Field Values

CONFIGURATION

```
private static final int CONFIGURATION
```

See Also:

Constant Field Values

CONFIGURATION_RESPONSE

```
private static final int CONFIGURATION_RESPONSE
```

See Also:
Constant Field Values

FAULT_OCCURRED_RESPONSE

```
private static final int FAULT_OCCURRED_RESPONSE
```

See Also:
Constant Field Values

FAULT_UPDATE_RESPONSE

```
private static final int FAULT_UPDATE_RESPONSE
```

See Also:
Constant Field Values

STATISTICS_REPORT_RESPONSE

```
private static final int STATISTICS_REPORT_RESPONSE
```

See Also:
Constant Field Values

ACTION_RESPONSE

```
private static final int ACTION_RESPONSE
```

See Also:
Constant Field Values

BOM_RESPONSE

```
private static final int BOM_RESPONSE
```

See Also:
Constant Field Values

---

cipherText1

```
public javax.swing.JTextArea cipherText1
```

---

cipherText2

```
public javax.swing.JTextArea cipherText2
```

---

buttonPanel

```
private javax.swing.JPanel buttonPanel
```

TextArea

```
private javax.swing.JPanel TextArea
```

---

buttonPanelB

```
private javax.swing.JButton[] buttonPanelB
```

---

configurationButton

```
private javax.swing.JButton[] configurationButton
```

---

set

```
private javax.swing.JButton[] set
```

---

scrollingArea1

```
private javax.swing.JScrollPane scrollingArea1
```

---

scrollingArea2

```
private javax.swing.JScrollPane scrollingArea2
```

---

parameterInput

```
private javax.swing.JTextField[] parameterInput
```

---

actionPanelInput

```
private javax.swing.JTextField[] actionPanelInput
```

---

actionTypes

```
private javax.swing.JRadioButton[] actionTypes
```

---

actions

```
private javax.swing.ButtonGroup actions
```

RACK

```
private javax.swing.JCheckBox RACK
```

---

parameters

```
private javax.swing.JLabel[] parameters
```

---

configurationPanel

```
private javax.swing.JPanel configurationPanel
```

---

systems

```
private javax.swing.JTabbedPane systems
```

---

actionPanel

```
private javax.swing.JPanel actionPanel
```

---

clientListener

```
private ServerAppGUI.EventListener clientListener
```

Constructor Detail

ServerAppGUI

```
public ServerAppGUI()
```

Constructs the client GUI

Method Detail

buildTabPane

```
private void buildTabPane()
```

build the Tab pane and add the panel into it

---

buildConfigurationPanel

```
private void buildConfigurationPanel()
```

Build the server configuration panel with required input fields

---

buildActionPanel

```
private void buildActionPanel()
```

build the Server Action Panel

---

buildTextArea

```
private void buildTextArea()
```

Build the text areas

---

buildButtonPanel

```
private void buildButtonPanel()
```

The buildButtonPanel method builds the button panel.

---

setState

```
public void setState(int State)
```

Synchronized method to set the client state

Overrides:

`setState` in class `java.awt.Frame`

Parameters:

`State` - state value

---

getState

```
public int getState()
```

Synchronized method to get the client state

Overrides:
`getState` in class `java.awt.Frame`

Returns:

---

getDeviceCategory

```
public int getDeviceCategory()
```

Returns:
the deviceCategory

---

setDeviceCategory

```
public void setDeviceCategory(int deviceCategory)
```

Parameters:
`deviceCategory` - the deviceCategory to set

---

getDeviceID

```
public int getDeviceID()
```

Returns:
the deviceID

---

setDeviceID

```
public void setDeviceID(int deviceID)
```

Parameters:
`deviceID` - the deviceID to set

---

getPMTU

```
public int getPMTU()
```

Returns:

the pMTU

---

setPMTU

```
public void setPMTU(int pMTU)
```

Parameters:
`pMTU` - the pMTU to set

---

getRMS_IP

```
public java.net.InetAddress getRMS_IP()
```

Returns:
the rMS_IP

---

setRMS_IP

```
public void setRMS_IP(java.net.InetAddress rMSIP)
```

Parameters:
`rMSIP` - the rMS_IP to set

---

getAS_IP

```
public java.net.InetAddress getAS_IP()
```

Returns:
the aS_IP

---

setAS_IP

```
public void setAS_IP(java.net.InetAddress aSIP)
```

Parameters:
`aSIP` - the aS_IP to set

---

getRMS_PORT

```
public int getRMS_PORT()
```

Returns:
the rMS_PORT

---

setRMS_PORT

```
public void setRMS_PORT(int rMSPORT)
```

Parameters:
`rMSPORT` - the rMS_PORT to set

---

getAS_PORT

```
public int getAS_PORT()
```

Returns:
the aS_PORT

---

setAS_PORT

```
public void setAS_PORT(int aSPORT)
```

Parameters:
`aSPORT` - the aS_PORT to set

---

getLOCAL_PORT

```
public int getLOCAL_PORT()
```

Returns:
the lOCAL_PORT

---

setLOCAL_PORT

```
public void setLOCAL_PORT(int lOCALPORT)
```

Parameters:
`lOCALPORT` - the lOCAL_PORT to set

---

getTX_delay

```
public int getTX_delay()
```

Returns:
the tX_delay

---

setTX_delay

```
public void setTX_delay(int tXDelay)
```

Parameters:
`tXDelay` - the tX_delay to set

---

getReTX_T

```
public int getReTX_T()
```

Returns:
the reTX_T

---

setReTX_T

```
public void setReTX_T(int reTXT)
```

Parameters:
`reTXT` - the reTX_T to set

---

getUdpBufferLength

```
public int getUdpBufferLength()
```

Returns:
the udpBufferLength

---

setUdpBufferLength

```
public void setUdpBufferLength(int udpBufferLength)
```

Parameters:
`udpBufferLength` - the udpBufferLength to set

---

getReTransmisssionRetryCount

```
public int getReTransmisssionRetryCount()
```

Returns:
the reTransmisssionRetryCount

---

setReTransmisssionRetryCount

```
public void setReTransmisssionRetryCount(int reTransmisssionRetryCount)
```

Parameters:
`reTransmisssionRetryCount` - the reTransmisssionRetryCount to set

---

getReassemblingTimeout

```
public int getReassemblingTimeout()
```

Returns:
the reassemblingTimeout

---

setReassemblingTimeout

```
public void setReassemblingTimeout(int reassemblingTimeout)
```

Parameters:
`reassemblingTimeout` - the reassemblingTimeout to set

---

getMsgType

```
public int getMsgType()
```

Returns:
the msgType

---

setMsgType

```
public void setMsgType(int msgType)
```

Parameters:
`msgType` - the msgType to set

---

getMsg

```
public java.lang.String getMsg()
```

Returns:
the msg

---

setMsg

```
public void setMsg(java.lang.String msg)
```

Parameters:
`msg` - the msg to set

---

isRMPMode

```
public boolean isRMPMode()
```

Returns:
the rMPMode

---

setRMPMode

```
public void setRMPMode(boolean rMPMode)
```

Parameters:
`rMPMode` - the rMPMode to set

---

isAF

```
public boolean isAF()
```

Returns:
the aF

---

setAF

```
public void setAF(boolean aF)
```

Parameters:
`aF` - the aF to set

---

getIp

```
public byte[] getIp()
```

Returns:
the ip

---

setIp

```
public void setIp(byte[] ip)
```

Parameters:
`ip` - the ip to set

APPENDIX III

JAVA-source code of the reference implementation of RMP

Antti Siirilä
Turku University of Applied Sciences

```
/**
 *
 */
package rmp;

import java.io.EOFException;
import java.io.IOException;
import java.net.InetAddress;

import rmpAPI.ApplicationMessageAPI;

/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 * This class holds the methods and data of application level messages
 */
public class ApplicationMessage implements ApplicationMessageAPI
{

    private int deviceCategory;
    private boolean AF;
    private boolean RACK;
    private int msgType;
    private int deviceID;
    private byte[] appMsg;
    private byte[] field;
    private InetAddress clientIP;
    private int clientPort;
    private boolean status;
    private int PMTUSize;
    private boolean messageState;
    private int offSet;


    //Tag of the application message field
    private static final int T_REGISTRATION_REQUEST = 1;
    private static final int T_REGISTRATION_RESPONSE = 2;
    private static final int T_CONFIGURATION_REQUEST = 3;
    private static final int T_AUHTENTICATON_SERVER = 4;
    private static final int T_AUHTENTICATION_PERIOD = 5;
    private static final int T_RMS_ADDRESS = 6;
    private static final int T_PMTU = 7;
    private static final int T_FRAGMENT_ACKNOWLEDGEMENT = 8;
    private static final int T_DATE_TIME = 9;
    private static final int T_STATISTICS_REPORT_TIME = 10;
    private static final int T_FAULT_FILTER = 11;
    private static final int T_STATUS_LOG_FILTER = 12;
    private static final int T_RESEND_INTERVAL = 13;
    private static final int T_CONFIGURATION_RESPONSE = 14;
    private static final int T_FAULT_CODE = 15;
    private static final int T_TIME_STAMP = 16;
```

```
    private static final int T_STATUS = 17;
    private static final int T_STATUS_BEFORE = 18;
    private static final int T_STATUS_AFTER = 19;
    private static final int T_COUNTER = 20;
    private static final int T_MEASURE = 21;
    private static final int T_STATISTICS_RESPONSE = 22;
    private static final int T_ACTION_REQUEST = 23;
    private static final int T_ACTION = 24;
    private static final int T_BOM = 25;
    private static final int T_BOM_RESPONSE = 26;


    public ApplicationMessage()
    {
        //message is empty and therefore read
        this.setMessageState(true);
        //init the offSet
        this.offSet = 0;

    }
    /**
     * Client receive
     * @param appMsgReceived
     * @param AF
     * @param msgType
     */
    public ApplicationMessage(byte[] appMsgReceived, boolean AF, int msgType)
    {
        this.appMsg = appMsgReceived.clone();
        this.AF = AF;
        this.msgType = msgType;
        this.offSet = 0;
    }
    /**
     * Server receive
     * @param appMsgReceived
     * @param AF
     * @param msgType
     * @param deviceID
     * @param sourceIP
     * @param sourcePort
     */
    public ApplicationMessage(byte[] appMsgReceived, boolean AF, int msgType, int
deviceID, InetAddress sourceIP, int sourcePort)
    {
        this.appMsg = appMsgReceived.clone();
        this.AF = AF;
        this.msgType = msgType;
        this.deviceID = deviceID;
        this.clientIP = sourceIP;
        this.clientPort = sourcePort;
        this.offSet = 0;
```

```
}
/* (non-Javadoc)
 * @see rmpAPI.ApplicationMessageAPI#getAF()
 */
@Override
public boolean getAF() {
    // TODO Auto-generated method stub
    return this.AF;
}

public void setAF(boolean AF)
{
    this.AF = AF;
}

public void setRMPMode(boolean RACK)
{
    this.RACK = RACK;
}

public boolean getRMPMode()
{
    return this.RACK;
}
/* (non-Javadoc)
 * @see rmpAPI.ApplicationMessageAPI#getAppMsg()
 */
@Override
public byte[] getAppMsg() {
    // TODO Auto-generated method stub
    return this.appMsg;
}

/* (non-Javadoc)
 * @see rmpAPI.ApplicationMessageAPI#getClientIP()
 */
@Override
public InetAddress getClientIP() {
    // TODO Auto-generated method stub
    return this.clientIP;
}

/* (non-Javadoc)
 * @see rmpAPI.ApplicationMessageAPI#getClientPort()
 */
@Override
public int getClientPort() {
    // TODO Auto-generated method stub
    return this.clientPort;
}

/* (non-Javadoc)
```

```
     * @see rmpAPI.ApplicationMessageAPI#getDeviceCategory()
     */
    @Override
    public int getDeviceCategory() {
        // TODO Auto-generated method stub
        return this.deviceCategory;
    }

    /* (non-Javadoc)
     * @see rmpAPI.ApplicationMessageAPI#getDeviceID()
     */
    @Override
    public int getDeviceID() {
        // TODO Auto-generated method stub
        return this.deviceID;
    }

    /* (non-Javadoc)
     * @see rmpAPI.ApplicationMessageAPI#getField(java.lang.String)
     */
    @Override
    public byte[] getField(int tag)
    {
        // TODO Auto-generated method stub
        //initate an input stream for the application message
        InputOutput in = new InputOutput(this.appMsg);

        //read first tag
        try {
            //Check that the tag is valid
                if(tag > 0 && tag < 27)
                {
                    in.din.mark(this.appMsg.length);
                    //search the tag
                    while(true)
                    {
                        try
                        {
                        //check if the next tag matches with the requested tag
                        if(tag == in.din.readUnsignedByte())
                        {
                            //if match is found read the lenght
                            int length = in.din.readUnsignedShort();
                            //initiate the field array
                            this.field = new byte[length];
                            //read the value
                            in.din.read(this.field, 0, length);
                            in.closeInputStream();
                            //return the value
                            return this.field;

                        }
```

```
                        else
                        {
                            in.din.skip(in.din.readUnsignedShort());
                        }
                        }
                        catch(EOFException i)
                        {
                            break;
                        }

                    }
                }
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            in.closeInputStream();
            return this.field;
        }

        /* (non-Javadoc)
         * @see rmpAPI.ApplicationMessageAPI#getMsgType()
         */
        @Override
        public int getMsgType() {
            // TODO Auto-generated method stub
            return this.msgType;
        }

        /* (non-Javadoc)
         * @see rmpAPI.ApplicationMessageAPI#msgReceived(boolean, int, int, byte[])
         */
        @Override
        public void msgReceived(boolean AF, int msgType, int deviceID, byte[] appMsg) {
            // TODO Auto-generated method stub

            this.AF = AF;
            this.msgType = msgType;
            this.deviceID = deviceID;
            this.appMsg = appMsg.clone();
            this.setMessageState(false);
        }

        /* (non-Javadoc)
         * @see rmpAPI.ApplicationMessageAPI#msgReceived(int, boolean, int, int, byte[],
    java.net.InetAddress, int)
         */
        @Override
        public void msgReceived(int deviceCategory, int msgType,
                int deviceID, byte[] appMsg, InetAddress clientIP, int clientPort) {
            // TODO Auto-generated method stub
```

```
        this.deviceCategory = deviceCategory;
        this.msgType = msgType;
        this.deviceID = deviceID;
        try
        {
        this.appMsg = appMsg.clone();
        }
        catch(NullPointerException e)
        {

        }
        this.clientIP = clientIP;
        this.clientPort = clientPort;
        this.setMessageState(false);
    }

    /* (non-Javadoc)
     * @see rmpAPI.ApplicationMessageAPI#pathMTUDiscoveryResult(boolean, int)
     */
    @Override
    public void pathMTUDiscoveryResult(boolean status, int PMTUSize)
    {
        // TODO Auto-generated method stub
        this.status = status;
        this.PMTUSize = PMTUSize;

    }


    /* (non-Javadoc)
     * @see rmpAPI.ApplicationMessageAPI#setField(java.lang.String, byte[])
     */
    @Override
    public boolean setField(int tag, byte[] value) {
        // TODO Auto-generated method stub
        //check that the tag is valid
        if(tag > 26 && tag < 1)
            return false;
        //init offSet
        this.offSet = 0;
        //initiate inputouput
        InputOutput out = new InputOutput();
        //start OuputStream
        out.stratOutputStream();
        //if some fields are already set save them first
        if(!(this.appMsg == null))
        {
            try
            {
            out.dout.write(this.appMsg);
            }
            catch (IOException e)
```

```
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }
        try
        {
            //write the tag into the stream
            out.dout.writeByte(tag);
            //write length field
            out.dout.writeShort((short)value.length);
            //write the value into the stream
            out.dout.write(value);
        }
        catch (IOException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        //save the new message
        this.appMsg = out.bout.toByteArray();

        //close outputstream
        out.closeOutputStream();
        return true;
    }

    public int getNextTag()
    {
        System.out.println("Offset is: " + this.offSet);
        //check if there still are fields in the messge
        if(this.offSet >= this.appMsg.length)
            return -1;
        //initiate the input stream
        InputOutput in = new InputOutput(this.appMsg);
        while(true)
        {
        try
        {
            //skip to the next tag
            in.din.skip(this.offSet);
            //read the next tag
            int help = (int)in.din.readUnsignedByte();
            //read the length of this value filed and update the offSet value
            this.offSet += 3 + in.din.readUnsignedShort();
            //return the tag
            System.out.println("Tag is: " + help);
            return help;
        }
        catch (IOException e) {
```

```
            // TODO Auto-generated catch block
            e.printStackTrace();
            //return error
            return -1;
        }
        }
    }

    public int getPMTUSize()
    {
        return this.PMTUSize;
    }

    /**
     * sets the PMTU size
     * @param PMTUSize
     */
    public void setPMTUSize(int PMTUSize)
    {
        this.PMTUSize = PMTUSize;
    }

    /**
     * returns the state of the PMTU Discovery
     * @return True: Inactive False: Active
     */
    public boolean getPMTUDiscoveryState()
    {
        return this.status;
    }

    /**
     * Composes an application message for the server
     */
    public void composeMsg(int deviceCategory, boolean RACK, boolean AF, int msgType,
int deviceID, byte[] appMsg, InetAddress clientIP, int clientPort)
    {
        this.deviceCategory = deviceCategory;
        this.RACK = RACK;
        this.AF = AF;
        this.appMsg = appMsg.clone();
        this.msgType = msgType;
        this.deviceID = deviceID;
        this.clientIP = clientIP;
        this.clientPort = clientPort;
        this.setMessageState(false);
    }

    /**
     * Composes an application message for the client
     */
    public void composeMsg(boolean RACK, int msgType, byte[] appMsg)
```

```
{

    this.RACK = RACK;
    this.appMsg = appMsg.clone();
    this.msgType = msgType;
    this.setMessageState(false);

}

/**
 * return the state of the message
 */
public boolean isRead()
{
    return this.isSyncRead();
}

/**
 * Synchronized message state
 * @return
 */
private synchronized boolean isSyncRead()
{
    return this.messageState;
}

/**
 * sets the message state value
 * @param state True: read (application). False: not read (RMP)
 */
public void setMessageState(boolean state)
{
    this.syncMessageState(state);
}

private synchronized void syncMessageState(boolean state)
{
    this.messageState = state;
}
@Override
public void initMsg(int deviceCategory, boolean RACK, boolean AF,
        int msgType, int deviceID, InetAddress clientIP, int clientPort) {
    // TODO Auto-generated method stub
    this.deviceCategory = deviceCategory;
    this.RACK = RACK;
    this.AF = AF;
    this.msgType = msgType;
    this.deviceID = deviceID;
    this.clientIP = clientIP;
    this.clientPort = clientPort;
    this.setMessageState(false);
```

```
        }
        @Override
        public void initMsg(boolean RACK, int msgType) {
            // TODO Auto-generated method stub
            this.RACK = RACK;
            this.msgType = msgType;
            this.setMessageState(false);

        }
        @Override
        public void setOffSet(int offSet)
        {
            // TODO Auto-generated method stub
            this.offSet = offSet;

        }


    }
```

```
package rmpAPI;

import java.net.InetAddress;
/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 *This is the application message interface for the RMP client and server
applications
 */
public interface ApplicationMessageAPI
{


    /**
     * Copies the received message data into the client application
     * @param AF    True: Action Flag set, False: Action Flag cleared
     * @param msgType   Application message type
     * @param deviceID  ID of the client device
     * @param appMsg    application data from RMP PDU
     *
     */
    public void msgReceived(boolean AF, int msgType, int deviceID, byte[] appMsg);

    /**
     * Copies the received message data into the server application
     *
     * @param deviceCategory    Category of the client device
     * @param AF    Action Flag
     * @param msgType   Application message type
     * @param deviceID  ID of the client device
     * @param appMsg    Application message data
     * @param clientIP  IP address of the client
     * @param clientPort    UDP port of client's RMP process.
     */
    public void msgReceived(int deviceCategory, int msgType, int deviceID, byte[]
appMsg, InetAddress clientIP, int clientPort);

    /**
     * Indicates the application about the status of the PMTU discovery process.
     * @param status    True: PMTU discovery successfully performed
     * @param   PMTUSize
     */
    public void pathMTUDiscoveryResult(boolean status, int PMTUSize);

    /**
     * Return the value of device category of the client device
     * @return  device category
     */
    public int getDeviceCategory();

    /**
```

```
     * Return the state of the action flag
     * @return  True: action flag set, False: action flag cleared
     */
    public boolean getAF();

    /**
     * sets the action flag (only server)
     * @param AF
     */
    public void setAF(boolean AF);
    /**
     * Return the value of message type
     * @return  message type
     */
    public int getMsgType();

    /**
     * Return the value of device ID of the client device
     * @return  device ID
     */
    public int getDeviceID();

    /**
     * Returns the value of the field matching with the given tag from the received
application message
     * @param tag
     * @return the field data in byte array
     */
    public byte[] getField(int tag);

    /**
     * Set the data provided in the value-array with the given tag into the
application message
     * @param tag
     * @return True: success False: error
     */
    public boolean setField(int tag, byte[] value);


    /**
     * Return the next tag value in the application message
     * @return next tag value || -1 if error
     */
    public int getNextTag();

    /**
     * Returns the encoded application message to be send for RMP
     * @return Application message data (RMP payload)
     */
    public byte[] getAppMsg();

    /**
```

```
     * Return the IP-address of the client device from where the packet was sent
     * @return Client IP
     */
    public InetAddress getClientIP();

    /**
     * Return the UDP port of the client device from where the packet was sent
     * @return UDP port value
     */
    public int getClientPort();

    /**
     * sets the RMP mode
     * @param mode
     */
    public void setRMPMode(boolean mode);

    /**
     * return the size of the PMTU
     * @return PMTU value
     */
    public int getPMTUSize();

    /**
     * sets the PMTU size
     * @param PMTUSize
     */
    public void setPMTUSize(int PMTUSize);

    /**
     * Compose an application message to be sent from the server
     * @param deviceCategory
     * @param AF
     * @param msgType
     * @param deviceID
     * @param appMsg
     * @param clientIP
     * @param clientPort
     */
    public void composeMsg(int deviceCategory,boolean RACK, boolean AF, int msgType,
int deviceID, byte[] appMsg, InetAddress clientIP, int clientPort);


    /**
     * Initializes an application message to be sent from the server
     * @param deviceCategory
     * @param AF
     * @param msgType
     * @param deviceID
     * @param clientIP
     * @param clientPort
     */
```

```
    public void initMsg(int deviceCategory,boolean RACK, boolean AF, int msgType, int
deviceID, InetAddress clientIP, int clientPort);

    /**
     * Compose an application message to be sent from the client
     * @param RACK
     * @param msgType
     * @param appMsg
     */
    public void composeMsg( boolean RACK, int msgType, byte[] appMsg);

    /**
     * Initializes an application message for the client
     * @param RACK RMP mode
     * @param msgType Message type
     */
    public void initMsg( boolean RACK, int msgType);

    /**
     * returns the state of the PMTU Discovery
     * @return True: Inactive False: Active
     */
    public boolean getPMTUDiscoveryState();

    /**
     * Return the RMP mode information
     * @return True: RMP acknowledged. False: RMP unacknowledged
     */
    public boolean getRMPMode();

    /**
     * Indicates the status of the received application message.
     * @return True: message is read and can be over written (application sets).
False: unread message (RMP sets)
     */
    public boolean isRead();

    /**
     * This the offset of the TLV tags in the application message. When the
getNextTag has returned -1, which indicates
     * that the last tag was read, this method can be used to reset the offset to
zero.
     * @param offSet
     */
    public void setOffSet(int offSet);

}
```

```
package rmp;
/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 * This class holds the information of application message tags. The class can be
used to convert a integer tag to a string name.
 */
public class ApplicationMessageTags
{

    //Tag of the application message field
    private static final int T_REGISTRATION_REQUEST = 1;
    private static final int T_REGISTRATION_RESPONSE = 2;
    private static final int T_CONFIGURATION_REQUEST = 3;
    private static final int T_AUHTENTICATON_SERVER = 4;
    private static final int T_AUHTENTICATION_PERIOD = 5;
    private static final int T_RMS_ADDRESS = 6;
    private static final int T_PMTU = 7;
    private static final int T_FRAGMENT_ACKNOWLEDGEMENT = 8;
    private static final int T_DATE_TIME = 9;
    private static final int T_STATISTICS_REPORT_TIME = 10;
    private static final int T_FAULT_FILTER = 11;
    private static final int T_STATUS_LOG_FILTER = 12;
    private static final int T_RESEND_INTERVAL = 13;
    private static final int T_CONFIGURATION_RESPONSE = 14;
    private static final int T_FAULT_CODE = 15;
    private static final int T_TIME_STAMP = 16;
    private static final int T_STATUS = 17;
    private static final int T_STATUS_BEFORE = 18;
    private static final int T_STATUS_AFTER = 19;
    private static final int T_COUNTER = 20;
    private static final int T_MEASURE = 21;
    private static final int T_STATISTICS_RESPONSE = 22;
    private static final int T_ACTION_REQUEST = 23;
    private static final int T_ACTION = 24;
    private static final int T_BOM = 25;
    private static final int T_BOM_RESPONSE = 26;

    private String tagName;

    public ApplicationMessageTags(int tag)
    {
        //check the tag name
        switchTagToName(tag);
    }

    private void switchTagToName(int tag)
    {
        switch(tag)
        {
        case T_REGISTRATION_REQUEST:
        {
```

```
            this.tagName = "Registration request";
            break;
        }
        case T_REGISTRATION_RESPONSE:
        {
            this.tagName = "Registration response";
            break;
        }
        case T_CONFIGURATION_REQUEST:
        {
            this.tagName = "Configuration request";
            break;
        }
        case T_AUHTENTICATON_SERVER:
        {
            this.tagName = "Address of the authentication server";
            break;
        }
        case T_AUHTENTICATION_PERIOD:
        {
            this.tagName = "The length of the authentication period";
            break;
        }
        case T_RMS_ADDRESS:
        {
            this.tagName = "Address of the RMS";
            break;
        }
        case T_PMTU:
        {
            this.tagName = "Size of the Path MTU";
            break;
        }
        case T_FRAGMENT_ACKNOWLEDGEMENT:
        {
            this.tagName = "RMP mode";
            break;
        }
        case T_DATE_TIME:
        {
            this.tagName = "Date and Time";
            break;
        }
        case T_STATISTICS_REPORT_TIME:
        {
            this.tagName = "The time of day for the statistics report";
            break;
        }
        case T_FAULT_FILTER:
        {
            this.tagName = "Fault filter";
            break;
```

```
        }
        case T_STATUS_LOG_FILTER:
        {
            this.tagName = "Status log filter";
            break;
        }
        case T_RESEND_INTERVAL:
        {
            this.tagName = "Resend interval";
            break;
        }
        case T_CONFIGURATION_RESPONSE:
        {
            this.tagName = "Configuration completed";
            break;
        }
        case T_FAULT_CODE:
        {
            this.tagName = "Fault code";
            break;
        }
        case T_TIME_STAMP:
        {
            this.tagName = "Time stamp";
            break;
        }
        case T_STATUS:
        {
            this.tagName = "Status";
            break;
        }
        case T_STATUS_BEFORE:
        {
            this.tagName = "Status before the fault";
            break;
        }
        case T_STATUS_AFTER:
        {
            this.tagName = "Status after the fault";
            break;
        }
        case T_COUNTER:
        {
            this.tagName = "Counter";
            break;
        }
        case T_MEASURE:
        {
            this.tagName = "Mesure";
            break;
        }
        case T_STATISTICS_RESPONSE:
```

```
            {
                this.tagName = "Statistics received";
                break;
            }
            case T_ACTION_REQUEST:
            {
                this.tagName = "Action request";
                break;
            }
            case T_ACTION:
            {
                this.tagName = "Action type";
                break;
            }
            case T_BOM:
            {
                this.tagName = "BOM";
                break;
            }
            case T_BOM_RESPONSE:
            {
                this.tagName = "BOM received";
                break;
            }
            default:
            {
                this.tagName = "Tag does not exist";
                break;
            }
            }
        }

        /**
         * @return the tagName
         */
        public synchronized String getTagName() {
            return tagName;
        }

        /**
         * @param tagName the tagName to set
         */
        public synchronized void setTagName(String tagName) {
            this.tagName = tagName;
        }
    }
```

```
package rmp;


    import javax.swing.*;
    import javax.swing.text.BadLocationException;

    import java.awt.*;
    import java.awt.event.*;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.text.DecimalFormat;
/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 * This is the client application GUI to control client activities.
 */

public class ClientAppGUI extends JFrame
{

       //RMP parameters
       private int deviceCategory;
       private int deviceID;
       private int PMTU;
       private InetAddress RMS_IP;
       private InetAddress AS_IP;
       private int RMS_PORT;
       private int AS_PORT;
       private int LOCAL_PORT;
       private int TX_delay;
       private int reTX_T;
       private int udpBufferLength;
       private int reTransmisssionRetryCount;
       private int reassemblingTimeout;

       private int State;
       private int msgType;
       private String msg;
       private boolean RMPMode;


    private static final int DISABLED = 1;
    private static final int AUTHENTICATING = 2;
    private static final int REGISTERING = 3;
    private static final int IDLE = 4;
    private static final int SENDING = 5;
    private static final int RECEIVING = 6;
    private static final int CONFIGURING = 7;
    private static final int GO_TO_IDLE = 8;
    private static final int RECEIVED = 9;
    private static final int RECEIVE = 10;
```

```
private static final int ACTION = 11;
private static final int SEND = 12;


//Application request message types
private static final int REGISTRATION_REQUEST = 6;
private static final int CONFIGURATION_REQUEST = 8;
private static final int FAULT_OCCURRED_REQUEST = 12;
private static final int FAULT_UPDATE_REQUEST = 14;
private static final int STATISTICS_REPORT_REQUEST = 16;
private static final int ACTION_REQUEST = 18;
private static final int BOM_REQUEST = 20;
//Application response message types
private static final int REGISTRATION_RESPONSE = 7;
private static final int CONFIGURATION = 10;
private static final int CONFIGURATION_RESPONSE = 11;
private static final int FAULT_OCCURRED_RESPONSE = 13;
private static final int FAULT_UPDATE_RESPONSE = 15;
private static final int STATISTICS_REPORT_RESPONSE = 17;
private static final int ACTION_RESPONSE = 19;
private static final int BOM_RESPONSE = 21;




    public JTextArea cipherText1;
    public JTextArea cipherText2;
    private JPanel buttonPanel;    // To hold the buttons
    private JPanel TextArea;    // To hold the buttons
   private JButton[] buttonPanelB;    // To calculate the cost
   private JButton[] configurationButton;    // To exit the application
   private JButton[] send;    // To exit the application
   private JScrollPane scrollingArea1;
   private JScrollPane scrollingArea2;
   private JTextField[] parameterInput;
   private JTextField sendingPanelInput;
   private JRadioButton[] msgTypes;
   private ButtonGroup messages;
   private JCheckBox RACK;
   private JLabel[] parameters;
   private JPanel configurationPanel;
   private JTabbedPane systems;
   private JPanel sendingPanel;
   private EventListener clientListener;

   /**
    * Constructs the client GUI
    */
   public ClientAppGUI()
   {
       this.setState(DISABLED);//set the state to disabled
       this.clientListener = new EventListener();//Initiate the EventListener
```

```
object
          // Display a title.
           setTitle("RMP Client Demo");

           // Specify an action for the close button.
           setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

           // Create a BorderLayout manager.
           setLayout(new BorderLayout());

           // Create the custom panels.
           buildTextArea();
          // Create the button panel.
           buildButtonPanel();
           buildTabPane();
           // Add the components to the content pane.

           add(TextArea, BorderLayout.CENTER);
           add(buttonPanel, BorderLayout.SOUTH);
           add(systems, BorderLayout.EAST);

           // Pack the contents of the window and display it.
           pack();
           setVisible(true);
        }

        /**
         * build the Tab pane and add the panel into it
         */
        private void buildTabPane()
        {
            systems = new JTabbedPane();
         //      buildRadioButtons();
            buildConfigurationPanel();
               buildSendingPanel();
               JPanel blanc = new JPanel(new GridBagLayout());


                GridBagConstraints c = new GridBagConstraints();

                c.gridwidth = 32;
                c.gridx = 0;
                c.gridy = 0;
                c.gridheight = 24;

                blanc.add(new JLabel("Analyzer Disabled
"), c);



               systems.addTab("Disabled",null, blanc,"Analyzer not enabled");
               systems.setMnemonicAt(0, KeyEvent.VK_1);
```

```
    }

      /**
       * Build the client configuration panel with required input fields
       */
      private void buildConfigurationPanel()
      {
          configurationPanel = new JPanel(new GridBagLayout());

          GridBagConstraints c = new GridBagConstraints();

          c.gridx = 0;
          c.gridy = 0;
          c.gridwidth = 16;
          c.gridheight = 24;

          //initiate the configure button
          configurationButton = new JButton[1];
          configurationButton[0] = new JButton("Configure");
          //initiate the input fields for the parameters
          parameterInput = new JTextField[21];
          //initiate the parameter labels
          parameters = new JLabel[15];
          //initiate the labels
          String[] labelText = {"Device Category","Device ID","Path MTU",
"Transmission Delay","Retransmission Delay","Size of the Receiving UDP Buffer",
"Number of Retransmissions","Reassembling Timeout", "Server IP", "Server Name",
"Server Port","Authentication Server IP", "Authentication Server Name",
"Authentication Server Port","Local Port"};
          String[] defaultConfiguration = {"1","2010","1278","2000","10000","9092",
"3","30000", "127", "0", "0", "1", "RMS","9046", "127", "0", "0", "1", "A_Server",
"10111","10222"};

          int help = 0;
          int helpAgain = 16;
          for(int i = 0; i < parameterInput.length; i++)
          {

              if((i > 7 && i < 12) || (i > 13 && i < 18))
              {

                  if(i == 8 || i == 14)
                  {
                      c.gridx = 0;
                      //initiate the JLabel with the corresponding string
                      parameters[help] = new JLabel(labelText[help]);
                      //add the label
                      configurationPanel.add(parameters[help], c);
                      help++;
                  }
                  //initiate the input field for IP
```

```
                    parameterInput[i] = new JTextField(defaultConfiguration[i], 3);
                    //increment the x-constrain to next column
                    c.gridwidth = 3;
                    c.gridx = helpAgain;
                    //add the input. inputs in index 8, 9, 10, and 11 and 14, 15, 16,
and 17 are for the IP addresses
                    configurationPanel.add(parameterInput[i], c);
                    helpAgain += 3;
                    if(i == 11 || i == 17)
                        c.gridy += 24;
                    continue;
                }
                c.gridwidth = 16;
                helpAgain = 16;
                c.gridx = 0;
                //initiate the JLabel with the corresponding string
                parameters[help] = new JLabel(labelText[help]);
                //add the label
                configurationPanel.add(parameters[help], c);
                //initiate the input field
                parameterInput[i] = new JTextField(defaultConfiguration[i], 16);
                //set the x-constrain to second column
                c.gridx = 16;
                //add the input
                configurationPanel.add(parameterInput[i], c);
                //next row
                c.gridy += 24;
                help++;
            }

            c.gridx = 0;
//          c.gridy = 24;
            c.fill = GridBagConstraints.HORIZONTAL;
            configurationPanel.add(configurationButton[0], c);
            //add action listener for the button
            configurationButton[0].addActionListener(this.clientListener);

        }

        /**
         * build the client sendigPanel
         */
        private void buildSendingPanel()
        {
            //initiate the sending panel with GridBackLayout
            this.sendingPanel = new JPanel(new GridBagLayout());
            //initiate the constrains for the GridBackLayout
            GridBagConstraints c = new GridBagConstraints();
            //Set the constrains
            c.gridx = 0;
            c.gridy = 0;
            c.gridwidth = 12;
```

```
            c.gridheight = 24;


            //initiate a panel array for inner panels
            JPanel innerPanels[] = new JPanel[2];
            innerPanels[0] = new JPanel(new GridLayout(4, 1));
            innerPanels[1] = new JPanel(new GridLayout(4, 1));
            //set borders around the panels
            innerPanels[0].setBorder(BorderFactory.createTitledBorder("Messages
Procedures"));
            innerPanels[1].setBorder(BorderFactory.createTitledBorder("RMP Mode"));

            //initiate the JRadioButton array for the application messages
            this.msgTypes = new JRadioButton[4];
            //initiate the radio buttons
            this.msgTypes[0] = new JRadioButton("Fault Occurred", true);
            this.msgTypes[1] = new JRadioButton("Fault Update");
            this.msgTypes[2] = new JRadioButton("Statistics Report");
            this.msgTypes[3] = new JRadioButton("Confiugration");

            //initiate the messages ButtonGroup
            this.messages = new ButtonGroup();

            //add the buttons into the ButtonGroup and into the sendingpanel
            for(int i = 0; i < this.msgTypes.length; i++)
            {
                this.messages.add(this.msgTypes[i]);
                innerPanels[0].add(this.msgTypes[i]);
            }

            //initiate a checkBox for RMP mode bit
             this.RACK = new JCheckBox("RMP Acknowledged Mode");
            //initiate a panel for the check box
            innerPanels[1].add(this.RACK);

            //add the Rbuttons panel into the sending panel
            this.sendingPanel.add(innerPanels[0], c);
            //set the x-constrain
            c.gridx = 12;
            //add the check box panel into sending panel
            this.sendingPanel.add(innerPanels[1], c);

            //set the constrains
            c.gridx = 0;
            c.gridy += 24;
            //initiates the send button array
            this.send = new JButton[1];
            //initiate the input field for the application message
            this.sendingPanelInput = new JTextField(15);
            //add an informative label into the panel
            this.sendingPanel.add(new JLabel("Application Msg:"),c);
            //set the x-constrain
```

```
        c.gridx = 12;
        //add the message input into the panel
        this.sendingPanel.add(this.sendingPanelInput, c);
        //initiate the send button
        this.send[0] = new JButton("Send");
        //add actionlistener for the button
        this.send[0].addActionListener(this.clientListener);
        //set the constrains
        c.gridwidth = 24;
        c.gridx = 0;
        c.gridy += 24;
        //add the button into the panel
        this.sendingPanel.add(send[0], c);



    }



    /**
     * Build the text areas
     */
    private void buildTextArea()
    {
           TextArea = new JPanel();
           cipherText1 = new JTextArea(30,20);
           cipherText1.setLineWrap(true);
           cipherText1.setWrapStyleWord(true);
           cipherText2 = new JTextArea(30,20);
           cipherText2.setLineWrap(true);
           cipherText2.setWrapStyleWord(true);

           scrollingArea1 = new JScrollPane(cipherText1);
           scrollingArea2 = new JScrollPane(cipherText2);
           TextArea.add(scrollingArea1);
           TextArea.add(scrollingArea2);
    }
    /**
       The buildButtonPanel method builds the button panel.
    */

    private void buildButtonPanel()
    {
       // Create a panel for the buttons.
       buttonPanel = new JPanel();

       // Create the buttons.
       buttonPanelB = new JButton[3];
       buttonPanelB[0] = new JButton("Exit");
       buttonPanelB[1] = new JButton("Enable");
       buttonPanelB[2] = new JButton("Disable");
```

```
            // Register the action listeners.
            buttonPanelB[0].addActionListener(this.clientListener);
            buttonPanelB[1].addActionListener(this.clientListener);
            buttonPanelB[2].addActionListener(this.clientListener);

            // Add the buttons to the button panel.
            buttonPanel.add(buttonPanelB[0]);
            buttonPanel.add(buttonPanelB[1]);
            buttonPanel.add(buttonPanelB[2]);

        }


        /**
           Private inner class that handles the event when
           the user clicks the Calculate button.
        */

        public class EventListener implements ActionListener
        {

            private boolean[] notBeenHere;


            /**
             * Constructs an EventListener object. Sets the client state to disabled
             */
            public EventListener()
            {
                setState(DISABLED);//
                this.notBeenHere = new boolean[4];
                this.notBeenHere[0] = false;
                this.notBeenHere[1] = true;
                this.notBeenHere[2] = false;
                this.notBeenHere[3] = false;
            }
            /**
             * Action performed method
             */
           public void actionPerformed(ActionEvent e)
           {
               //Configure button pressed
               if(e.getSource() == configurationButton[0])
               {
                   //make sure that the client is disabled
//                 setState(DISABLED);
                   //read the given parameter values
                   try
                   {
                       setDeviceCategory(Integer.valueOf(parameterInput[0].getText()))
;
```

```
                        setDeviceID(Integer.valueOf(parameterInput[1].getText()));
                        setPMTU(Integer.valueOf(parameterInput[2].getText()));
                        setTX_delay(Integer.valueOf(parameterInput[3].getText()));
                        setReTX_T(Integer.valueOf(parameterInput[4].getText()));
                        setUdpBufferLength(Integer.valueOf(parameterInput[5].getText())
);
                        setReTransmisssionRetryCount(Integer.valueOf(parameterInput[6].
getText()));
                        setReassemblingTimeout(Integer.valueOf(parameterInput[7].
getText()));
                        byte[] ip = new byte[4]; // declare an array for the ip-
addresses

                      for(int i = 0; i < ip.length; i++)
                          ip[i] = (byte)Integer.parseInt(parameterInput[i + 8].getText
());//read the server ip-octets
                        RMS_IP = InetAddress.getByAddress(parameterInput[12].getText(),
ip);//initiate the InetAddress for RMS
                        setRMS_PORT(Integer.valueOf(parameterInput[13].getText()));
                        for(int i = 0; i < ip.length; i++)
                            ip[i] = (byte)Integer.parseInt(parameterInput[i + 14].
getText());//read the authentication server ip-octets
                        AS_IP = InetAddress.getByAddress(parameterInput[18].getText(),
ip);//initiate the InetAddress for AS
                        setAS_PORT(Integer.valueOf(parameterInput[19].getText()));
                        setLOCAL_PORT(Integer.valueOf(parameterInput[20].getText()));
                    }
                    catch(NullPointerException e1)
                    {
                        JOptionPane.showConfirmDialog(null, "Some of the parameters
were not correctly assigned");
                    }
//            cipherText2.append("Results of Frequently Analysis\n**********
******\nAlphabets\tCount\tPercent\n");
                    catch (UnknownHostException e2)
                    {
                          // TODO Auto-generated catch block
                          e2.printStackTrace();
                    }

                    if(this.notBeenHere[0])
                    {
                        //Add the sending tab after the parameters have been saved
                        systems.addTab("Send", null, sendingPanel, "Sending options");
                        systems.setMnemonicAt(1, KeyEvent.VK_2);
                        this.notBeenHere[0] = false;
                        //set the client state to configuring
                        setState(CONFIGURING);
                    }
                }
                //exit button
                else if(e.getSource() == buttonPanelB[0])
```

```
                {

                       System.exit(0);
                }
                //Enable button
                else if(e.getSource() == buttonPanelB[1])
                {

                    if(this.notBeenHere[1])
                    {
                        systems.removeTabAt(0);
                        //make configuration available
                        systems.addTab("Configuration",null, configurationPanel,"The
parameters for configuring the RMP client");
                        systems.setMnemonicAt(0, KeyEvent.VK_1);
                        // Pack the contents of the window and display it.
                        pack();
                        setVisible(true);
                        this.notBeenHere[0] = true;
                        this.notBeenHere[1] = false;
                        this.notBeenHere[2] = true;
                    }

                }
                //Disable button
                else if(e.getSource() == buttonPanelB[2])
                {

                    if(this.notBeenHere[2])
                    {
                        //disable the analyzer
                        setState(DISABLED);
                        //remove the configuration and send tabs
                        remove(systems);
                        //build disabled tab
                        buildTabPane();
                        add(systems, BorderLayout.EAST);
                        pack();
                        setVisible(true);
                        this.notBeenHere[0] = false;
                        this.notBeenHere[1] = true;
                        this.notBeenHere[2] = false;
                    }


                }
                //sending button
                else if(e.getSource() == send[0])
                {
                    //check the message type
                    if(msgTypes[0].isSelected())
                    {
```

```
                //set the correct message type
                setMsgType(FAULT_OCCURRED_REQUEST);
            }
            else if(msgTypes[1].isSelected())
            {
                //set the correct message type
                setMsgType(FAULT_UPDATE_REQUEST);

            }
            else if(msgTypes[2].isSelected())
            {
                //set the correct message type
                setMsgType(STATISTICS_REPORT_REQUEST);

            }
            else if(msgTypes[3].isSelected())
            {
                //set the correct message type
                setMsgType(CONFIGURATION_REQUEST);

            }

            //set the RMPMode if required
            if(RACK.isSelected())
            {
                setRMPMode(true);
            }
            else
            {
                setRMPMode(false);
            }
            //Get the message
            setMsg(sendingPanelInput.getText());
            cipherText1.append("\n***************\n");
            //change the client state to sending
                setState(SENDING);
        }
    }
}



    /**
     * Synchronized method to set the client state
     * @param State state value
     */
    public synchronized void setState(int State)
    {
        this.State = State;
    }

    /**
```

```
     * Synchronized method to get the client state
     * @return
     */
    public synchronized int getState()
    {
        return this.State;
    }
/**
 * @return the deviceCategory
 */
public synchronized int getDeviceCategory() {
    return deviceCategory;
}
/**
 * @param deviceCategory the deviceCategory to set
 */
public synchronized void setDeviceCategory(int deviceCategory) {
    this.deviceCategory = deviceCategory;
}
/**
 * @return the deviceID
 */
public synchronized int getDeviceID() {
    return deviceID;
}
/**
 * @param deviceID the deviceID to set
 */
public synchronized void setDeviceID(int deviceID) {
    this.deviceID = deviceID;
}
/**
 * @return the pMTU
 */
public synchronized int getPMTU() {
    return PMTU;
}
/**
 * @param pMTU the pMTU to set
 */
public synchronized void setPMTU(int pMTU) {
    PMTU = pMTU;
}
/**
 * @return the rMS_IP
 */
public synchronized InetAddress getRMS_IP() {
    return RMS_IP;
}
/**
 * @param rMSIP the rMS_IP to set
 */
```

```
public synchronized void setRMS_IP(InetAddress rMSIP) {
    RMS_IP = rMSIP;
}
/**
 * @return the aS_IP
 */
public synchronized InetAddress getAS_IP() {
    return AS_IP;
}
/**
 * @param aSIP the aS_IP to set
 */
public synchronized void setAS_IP(InetAddress aSIP) {
    AS_IP = aSIP;
}
/**
 * @return the rMS_PORT
 */
public synchronized int getRMS_PORT() {
    return RMS_PORT;
}
/**
 * @param rMSPORT the rMS_PORT to set
 */
public synchronized void setRMS_PORT(int rMSPORT) {
    RMS_PORT = rMSPORT;
}
/**
 * @return the aS_PORT
 */
public synchronized int getAS_PORT() {
    return AS_PORT;
}
/**
 * @param aSPORT the aS_PORT to set
 */
public synchronized void setAS_PORT(int aSPORT) {
    AS_PORT = aSPORT;
}
/**
 * @return the lOCAL_PORT
 */
public synchronized int getLOCAL_PORT() {
    return LOCAL_PORT;
}
/**
 * @param lOCALPORT the lOCAL_PORT to set
 */
public synchronized void setLOCAL_PORT(int lOCALPORT) {
    LOCAL_PORT = lOCALPORT;
}
/**
```

```
     * @return the tX_delay
     */
    public synchronized int getTX_delay() {
        return TX_delay;
    }
    /**
     * @param tXDelay the tX_delay to set
     */
    public synchronized void setTX_delay(int tXDelay) {
        TX_delay = tXDelay;
    }
    /**
     * @return the reTX_T
     */
    public synchronized int getReTX_T() {
        return reTX_T;
    }
    /**
     * @param reTXT the reTX_T to set
     */
    public synchronized void setReTX_T(int reTXT) {
        reTX_T = reTXT;
    }
    /**
     * @return the udpBufferLength
     */
    public synchronized int getUdpBufferLength() {
        return udpBufferLength;
    }
    /**
     * @param udpBufferLength the udpBufferLength to set
     */
    public synchronized void setUdpBufferLength(int udpBufferLength) {
        this.udpBufferLength = udpBufferLength;
    }
    /**
     * @return the reTransmisssionRetryCount
     */
    public synchronized int getReTransmisssionRetryCount() {
        return reTransmisssionRetryCount;
    }
    /**
     * @param reTransmisssionRetryCount the reTransmisssionRetryCount to set
     */
    public synchronized void setReTransmisssionRetryCount(
            int reTransmisssionRetryCount) {
        this.reTransmisssionRetryCount = reTransmisssionRetryCount;
    }
    /**
     * @return the reassemblingTimeout
     */
    public synchronized int getReassemblingTimeout() {
```

```
            return reassemblingTimeout;
        }
        /**
         * @param reassemblingTimeout the reassemblingTimeout to set
         */
        public synchronized void setReassemblingTimeout(int reassemblingTimeout) {
            this.reassemblingTimeout = reassemblingTimeout;
        }
        /**
         * @return the msgType
         */
        public synchronized int getMsgType() {
            return this.msgType;
        }
        /**
         * @param msgType the msgType to set
         */
        public synchronized void setMsgType(int msgType) {
            this.msgType = msgType;
        }
        /**
         * @return the msg
         */
        public synchronized String getMsg() {
            return msg;
        }
        /**
         * @param msg the msg to set
         */
        public synchronized void setMsg(String msg) {
            this.msg = msg;
        }
        /**
         * @return the rMPMode
         */
        public synchronized boolean isRMPMode() {
            return RMPMode;
        }
        /**
         * @param rMPMode the rMPMode to set
         */
        public synchronized void setRMPMode(boolean rMPMode) {
            RMPMode = rMPMode;
        }




}
```

```
package rmp;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.InetAddress;
import java.net.SocketTimeoutException;
import java.net.UnknownHostException;

import javax.swing.JOptionPane;


import rmp.*;

import rmpAPI.*;


/**
 *
 */

/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 *This is the RMP client application demonstration. The client registers with the
active server and sends fault, statistics, and configuration
 *messages when requested from the client GUI. In addition, the client performs the
action procedure if server so requires.
 *
 */
public class ClientApplicationDemo {

    /**
     * @param args
     */
    public static void main(String[] args) throws SocketTimeoutException
    {
        //RMP client constants
        final int deviceCategory = 1;
        final int deviceID = 2010;
        final int PMTU = 1278;
        final int rmsPort = 9046;
        final int asPort = 21111;
        final int localPort = 12345;
        final int TX_delay = 2000;
        final int reTXT = 10000;
        final int udpBuffer = 9092;
        final int reTransmissionCount = 3;
```

```
final int reassemblingTimeout = 30000;

//Application constans
final int reTransmissionRetryCount = 10;

//client states
final int DISABLED = 1;
final int AUTHENTICATING = 2;
final int REGISTERING = 3;
final int IDLE = 4;
final int SENDING = 5;
final int RECEIVING = 6;
final int CONFIGURING = 7;
final int GO_TO_IDLE = 8;
final int RECEIVED = 9;
final int RECEIVE = 10;
final int ACTION = 11;
final int SEND = 12;

//Application request message types
final int REGISTRATION_REQUEST = 6;
final int CONFIGURATION_REQUEST = 8;
final int FAULT_OCCURRED_REQUEST = 12;
final int FAULT_UPDATE_REQUEST = 14;
final int STATISTICS_REPORT_REQUEST = 16;
final int ACTION_REQUEST = 18;
final int BOM_REQUEST = 20;
//Application response message types
final int REGISTRATION_RESPONSE = 7;
final int CONFIGURATION = 10;
final int CONFIGURATION_RESPONSE = 11;
final int FAULT_OCCURRED_RESPONSE = 13;
final int FAULT_UPDATE_RESPONSE = 15;
final int STATISTICS_REPORT_RESPONSE = 17;
final int ACTION_RESPONSE = 19;
final int BOM_RESPONSE = 21;



//Actions to performe
final String configure = "CONFIGURATION_REQUEST";
final String sendBOM    = "BOM_REQUEST";

//Tag of the application message field
final int T_REGISTRATION_REQUEST = 1;
final int T_REGISTRATION_RESPONSE = 2;
final int T_CONFIGURATION_REQUEST = 3;
final int T_AUHTENTICATON_SERVER = 4;
final int T_AUHTENTICATION_PERIOD = 5;
final int T_RMS_ADDRESS = 6;
final int T_PMTU = 7;
final int T_FRAGMENT_ACKNOWLEDGEMENT = 8;
```

```
        final int T_DATE_TIME = 9;
        final int T_STATISTICS_REPORT_TIME = 10;
        final int T_FAULT_FILTER = 11;
        final int T_STATUS_LOG_FILTER = 12;
        final int T_RESEND_INTERVAL = 13;
        final int T_CONFIGURATION_RESPONSE = 14;
        final int T_FAULT_CODE = 15;
        final int T_TIME_STAMP = 16;
        final int T_STATUS = 17;
        final int T_STATUS_BEFORE = 18;
        final int T_STATUS_AFTER = 19;
        final int T_COUNTER = 20;
        final int T_MEASURE = 21;
        final int T_STATISTICS_RESPONSE = 22;
        final int T_ACTION_REQUEST = 23;
        final int T_ACTION = 24;
        final int T_BOM = 25;
        final int T_BOM_RESPONSE = 26;

        int clientState = 1;


        // TODO Auto-generated method stub

        //initiate new RMP Client object
        RMPClientAPI rmpClient = new RMPClient();

        //Declare the message APIs
        ApplicationMessageAPI sending;
        ApplicationMessageAPI receiving;

        //declare the RMS ip address
        InetAddress RMS_IP = null;
        //set the action state
        boolean ACTION_ON = false;
        //set the re-registration to true
        boolean RE_REGISTER = true;
        //initiate the RMS IP address
        try {
            //set IP addresses
            RMS_IP = InetAddress.getByAddress("RMS", new byte[]{(byte)127,(byte)0,0,1
});
        } catch (UnknownHostException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }

        ClientAppGUI gui = new ClientAppGUI();

        //Configure the client RMP
        gui.cipherText1.append("\nStarting the Idle loop");
        //init the application messages for sending and receiving
```

```
sending = new ApplicationMessage();
receiving = new ApplicationMessage();

//Start the main loop
while(true)
{
    receiving = new ApplicationMessage();
    gui.cipherText1.append("\nClient state: " + gui.getState());

    //wait here while analyzer is disabled
    while(gui.getState() == DISABLED)
    {
        //gui.cipherText1.append("\nDisabled");
        try
        {
            //delay to let the GUI set the parameter
            Thread.sleep(10);
        } catch (InterruptedException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    //Idle state
    while(gui.getState() == IDLE)
    {
        //If action flag is on, start action procedure
        if(ACTION_ON)
        {
            gui.setState(ACTION);
            continue;
        }
        try
        {
            //delay to let the GUI set the parameter
            Thread.sleep(10);
        } catch (InterruptedException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        //Receive while idling
        receiving = new ApplicationMessage();
        //if message was not received continue idling
        try
        {
            if(0 > rmpClient.receiveMsg(receiving, 10))
                continue;
        }
        catch (NullPointerException e)
        {
            continue;
```

```
                }
                //set the client state to received if a message was received
                gui.setState(RECEIVED);

            }

            //if there are not procedures going on and the received message was not
action response, start the action procedure after this procedure
            if(gui.getState() == ACTION)
            {
                gui.cipherText1.append("\nComposing action request");
                String stat = "send me the action";
                //init a new ApplicationMessage with correct parameter
                sending = new ApplicationMessage();
                sending.initMsg(gui.isRMPMode(), ACTION_REQUEST);
                //set required fields into the message
                sending.setField(T_ACTION_REQUEST, stat.getBytes());
                //send message
                gui.setState(SEND);
                //go directly to send

            }




            if(gui.getState() == AUTHENTICATING)
            {

            }
            else if(gui.getState() == REGISTERING)
            {
                gui.cipherText1.append("\nRegisterring");
                //compose the registration request message
                String stat = "2010";
                sending = new ApplicationMessage();
                sending.initMsg(false, REGISTRATION_REQUEST);
                sending.setField(T_REGISTRATION_REQUEST, stat.getBytes());
                gui.setState(SEND);

            }
            else if(gui.getState() == CONFIGURING)
            {
                gui.cipherText1.append("\nconfiguring");


                rmpClient.configure(gui.getDeviceCategory(), gui.getDeviceID(), gui.
getPMTU(),gui.getRMS_IP(), gui.getAS_IP(), gui.getRMS_PORT(), gui.getAS_PORT(), gui.
getLOCAL_PORT(), gui.getTX_delay() ,gui.getReTX_T(), gui.getUdpBufferLength(), gui.
getReTransmisssionRetryCount(), gui.getReassemblingTimeout());
                //if registering is required set the state accordingly
                if(RE_REGISTER)
```

```
            {
                gui.cipherText1.append("\nGoing to reRegister");
                gui.setState(REGISTERING);
                RE_REGISTER = false;
            }
            else
            {
                gui.setState(IDLE);
            }
            ACTION_ON = false;
            continue;
        }

        //SENDING STATE: to compose required messages
        else if (gui.getState() == SENDING)
        {
            //Compose a Fault Update message
            if(gui.getMsgType() == FAULT_UPDATE_REQUEST)
            {
                //init a new ApplicationMessage with correct parameter
                sending = new ApplicationMessage();
                sending.initMsg(gui.isRMPMode(), gui.getMsgType());
                //set required fields into the message
                sending.setField(T_FAULT_CODE, gui.getMsg().getBytes());
                //start input stream to write a long into the field array
                InputOutput out = new InputOutput();
                out.stratOutputStream();
                try
                {
                    out.dout.writeLong(System.currentTimeMillis());
                }
                catch (IOException e)
                {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                sending.setField(T_TIME_STAMP, out.bout.toByteArray());
                out.closeOutputStream();
                //send message
                gui.setState(SEND);

            }
            //Compose a Statistics Report request message
            if(gui.getMsgType() == STATISTICS_REPORT_REQUEST)
            {
                //init a new ApplicationMessage with correct parameter
                sending = new ApplicationMessage();
                sending.initMsg(gui.isRMPMode(), gui.getMsgType());
                //set required fields into the message
                sending.setField(T_COUNTER, gui.getMsg().getBytes());
                //send message
                gui.setState(SEND);
```

```
                }
                //Compose a Fault Occurred Request message
                if(gui.getMsgType() == FAULT_OCCURRED_REQUEST)
                {
                    //init a new ApplicationMessage with correct parameter
                    sending = new ApplicationMessage();
                    sending.initMsg(gui.isRMPMode(), gui.getMsgType());
                    //set required fields into the message
                    sending.setField(T_FAULT_CODE, gui.getMsg().getBytes());
                    //start input stream to write a long into the field array
                    InputOutput out = new InputOutput();
                    out.stratOutputStream();
                    try
                    {
                        out.dout.writeLong(System.currentTimeMillis());
                    }
                    catch (IOException e)
                    {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    sending.setField(T_TIME_STAMP, out.bout.toByteArray());
                    out.closeOutputStream();
                    //send message
                    gui.setState(SEND);

                }
                //Compose a Configuration Request message
                if(gui.getMsgType() == CONFIGURATION_REQUEST)
                {
                    //init a new ApplicationMessage with correct parameter
                    sending = new ApplicationMessage();
                    sending.initMsg(gui.isRMPMode(), gui.getMsgType());
                    //set required fields into the message
                    sending.setField(T_CONFIGURATION_REQUEST, gui.getMsg().getBytes()
);
                    //send message
                    gui.setState(SEND);
                }
                //Set the RMP acknowledged mode if required
                if(gui.isRMPMode())
                {
                    sending.setRMPMode(true);
                }

            }
            //init a counter for retransmission
            int count = 0;
            //if a message was received while idling, skip the send state
            if(gui.getState() == RECEIVED)
            {
```

```
            }
            else if(gui.getState() == SEND)
            {

                //start the retransmission loop
                while(true)
                {
                    //try to send and receive, and Catch null pointer if message was
not received
                    try
                    {
                        gui.cipherText1.append("\nSending started");
                        //Send
                        if(0>rmpClient.sendMsg(sending, gui.getReTX_T()))
                        {
                            gui.cipherText1.append("\nsend error");
                            //if sending error, do the retransmission routine
                            count++;
                            if(count == gui.getReTransmisssionRetryCount())
                            {
                                //if registration did not succeed, go to disable
                                if(sending.getMsgType() == REGISTRATION_REQUEST)
                                    gui.setState(DISABLED);
                                break;//break the retransmission if the limit is
reached
                            }
                            continue;//continue to retransmission
                        }

                        //receive only if a response is expected
                        if((sending.getMsgType()&1) == 1)
                        {
                            if(sending.getMsgType() == CONFIGURATION_RESPONSE)
                            {
                                gui.cipherText1.append("\nConfiguration complete
received. Going to re configure");
                                gui.setState(CONFIGURING);
                            }
                            else
                            {
                                //set state to idle
                                gui.setState(GO_TO_IDLE);
                            }
                            ACTION_ON = false;
                            break;
                        }
                        else
                        {
                            gui.cipherText1.append("\nReceiving");
                            if(0>rmpClient.receiveMsg(receiving, gui.getReTX_T()))
                            {
                                //if receiving error, do the retransmission routine
```

```
                                count++;
                                if(count == reTransmissionRetryCount)
                                {
                                    //if registration did not succeed, go to disable
                                    if(sending.getMsgType() == REGISTRATION_REQUEST)
                                        gui.setState(DISABLED);
                                    break;//break the retransmission if the limit is
reached
                                }
                                continue;//continue to retransmission
                            }
                            gui.cipherText1.append("\nMSG type received: " +
receiving.getMsgType());


                        }
                        break;//if message was send and no
                    }
                    catch (NullPointerException e)
                    {
                        //if message was not received, do the retransmission routine
                        count++;
                        if(count == reTransmissionRetryCount)
                            break;//break the retransmission if the limit is reached
                        continue;//continue to retransmission
                    }
                }//end of retransmission loop
            }//end of send
            gui.cipherText1.append("\nReceived message has AF: " + receiving.getAF())
;
            //
            if(gui.getState() == GO_TO_IDLE)
            {
                gui.cipherText1.append("\ngoing to idle");
                //change the state to IDLE and go directly there
                gui.setState(IDLE);
                continue;
            }
            else if(gui.getState() == DISABLED || gui.getState() == CONFIGURING)
            {
                //skip the rest and go directly to the disabled state
                continue;
            }
            else
            {
                gui.cipherText1.append("\nhandling received message");

                //*******************************Start the pre-checking************
*****************************//
                //check that the received message was for this device
                if(!(receiving.getDeviceID() == gui.getDeviceID()))
                {
```

```
                    gui.cipherText1.append("\nMessage Dropped because mismatch in
device ID");

                    //if not, drop the message
                    receiving = null;
                    gui.setState(IDLE);
                    continue; //continue to the idle state
                }

                //if the received message has the action flag set, go to set the
action on parameter
                if(receiving.getAF())
                {
                    ACTION_ON = true;//ACTION_ON state true
                }
                else
                {
                    ACTION_ON = false;//ACTION_ON state false
                }
                //*******************************Handle the received message*********
**********************************//
                //if fault occurred response is received check the fault message to
sent and clear the send message
                switch(receiving.getMsgType())
                {
                    //REGISTRATION_RESPONSE
                    case REGISTRATION_RESPONSE:
                    {
                        if(new String(receiving.getField(T_REGISTRATION_RESPONSE)).
equals("SUCCESS"))
                        {
                            printReceivedmsg(gui, receiving);
                            gui.cipherText2.append("\n****************\nClient
registerred with the RMS\nName: " + gui.getRMS_IP().getHostName() + "\nAddress: " +
gui.getRMS_IP().getHostAddress() + "\n****************\n");
                        }
                        //next state: idle
                        gui.setState(IDLE);
                        //delete the sent message
                        sending = null;
                        break;
                    }
                    //CONFIGURATION
                    case CONFIGURATION:
                    {
                        gui.cipherText1.append("\nConfiguration message received");
                        //print all fields
                        printReceivedmsg(gui, receiving);
                        //reset the TLV-offSet
                        receiving.setOffSet(0);
                        //Start parsing
                        int nextTag;
```

```
                        while((nextTag = receiving.getNextTag()) != -1)
                        {
                            gui.cipherText1.append("\nparsing the configuration
message");
                            try
                            {

                                if(nextTag == T_RMS_ADDRESS)
                                {
                                    gui.cipherText1.append("\nNew RMS address
received");
                                    //get the new RMS address
                                    RMS_IP = InetAddress.getByAddress(receiving.
getField(T_RMS_ADDRESS));
                                    if(gui.getRMS_IP().equals(RMS_IP))
                                    {
                                        RE_REGISTER = false;
                                    }
                                    else
                                    {
                                        gui.cipherText1.append("\nRMS changes
reregister");
                                        //set the address
                                        gui.setRMS_IP(RMS_IP);
                                        //re-registering required
                                        RE_REGISTER = true;
                                    }

                                }
                                if(nextTag == T_DATE_TIME)
                                {
                                        //This demonstration is done. Do nothing

                                }
                            }
                            catch (UnknownHostException e)
                            {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                            }
                            catch(NullPointerException g)
                            {
                                break;
                            }

                        }
                        String stat = "CONFIGURATION_COMPLETE";
                        sending = new ApplicationMessage();
                        //init the configuration response message
                        sending.initMsg(gui.isRMPMode(), CONFIGURATION_RESPONSE);
                        //add field
                        sending.setField(T_CONFIGURATION_RESPONSE, stat.getBytes());
```

```
                        //change the state to SEND
                        gui.setState(SEND);
                        break;
                    }
                    //FAULT_OCCURRED_RESPONSE
                    case FAULT_OCCURRED_RESPONSE:
                    {
                        sending = null;
                        printReceivedmsg(gui, receiving);
                        gui.setState(IDLE);
                        break;
                    }
                    //FAULT_UPDATE_RESPONSE
                    case FAULT_UPDATE_RESPONSE:
                    {
                        sending = null;
                        printReceivedmsg(gui, receiving);
                        gui.setState(IDLE);
                        break;
                    }
                    //STATISTICS_REPORT_RESPONSE
                    case STATISTICS_REPORT_RESPONSE:
                    {
                        sending = null;
                        printReceivedmsg(gui, receiving);
                        gui.setState(IDLE);

                        break;
                    }
                    //ACTION_RESPONSE
                    case ACTION_RESPONSE:
                    {
                        gui.cipherText1.append("\nAction response received. Value is
" + new String(receiving.getField(T_ACTION)));
                        //if configuration download was requested
                        if(new String(receiving.getField(T_ACTION)).equals
("CONFIGURATION_REQUEST"))
                        {
                            //set the message type to configuration request
                            gui.setMsgType(CONFIGURATION_REQUEST);
                            gui.setMsg("Send me some config");
                            //go to sending
                            gui.setState(SENDING);
                            ACTION_ON = true;
                            break;
                        }
                        //if BOM was requested
                        if(new String(receiving.getField(T_ACTION)).equals
("BOM_REQUEST"))
                        {
                            gui.cipherText1.append("\nSending BOM");
                            String stat = "This is BOM data of the client: " + new
```

```
String("" + gui.getDeviceID());
                            sending = new ApplicationMessage();
                            //init the BOM request message
                            sending.initMsg(gui.isRMPMode(), BOM_REQUEST);
                            //add the data field
                            sending.setField(T_BOM, stat.getBytes());
                            gui.setState(SEND);
                            ACTION_ON = true;
                            break;

                        }
                        break;
                    }
                    //BOM_RESPONSE
                    case BOM_RESPONSE:
                    {
                        printReceivedmsg(gui, receiving);
                        receiving = null;
                        gui.setState(IDLE);
                        break;
                    }
                    //UNKNOWN_MESSAGE_TYPE
                    default:
                    {
                        //Drop
                        receiving = null;
                        gui.setState(IDLE);
                        break;
                    }
                }//end of message type switch
            }//end of GO_TO_IDLE


        }//end of main loop

    }//end of main

    private static void printReceivedmsg(ClientAppGUI gui, ApplicationMessageAPI
receiving)
    {
        //init tag switch
        ApplicationMessageTags getName;
        gui.cipherText2.append("\nReceived a message type of " +receiving.getMsgType
());
        int nextTag;
        while((nextTag = receiving.getNextTag()) != -1)
        {
            try
            {
            getName = new ApplicationMessageTags(nextTag);
            if(nextTag == 16)
            {
```

```
                InputOutput in = new InputOutput(receiving.getField(nextTag));
                try
                {

                    gui.cipherText2.append("\nTag: " + getName.getTagName() + "\
nValue: " + in.din.readLong());
                }
                catch (IOException e)
                {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                in.closeInputStream();
                continue;
            }
            if(nextTag == 6)
            {
                gui.cipherText2.append("\nTag: " + getName.getTagName() + "\nValue: "
);
                for(int i = 0; i < receiving.getField(nextTag).length; i++)
                {
                    gui.cipherText2.append("" + receiving.getField(nextTag)[i] + ".")
;
                }

            }

            gui.cipherText2.append("\nTag: " + getName.getTagName() + "\nValue: " +
new String(receiving.getField(nextTag)));
            }
            catch(NullPointerException e)
            {
                break;
            }
        }
        gui.cipherText2.append("\n***************\n");



    }

} //end of ClientApplicationDemo
```

```
package rmp;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 * This class is used when ever RMP needs input or output stream services
 */
public class InputOutput
{
    public ByteArrayInputStream bin = null;
    public ByteArrayOutputStream bout = null;
    public DataInputStream din = null;
    public DataOutputStream dout = null;

    /**
     * Constructs an output stream
     */
    public InputOutput()
    {
    }

    /**
     * Constructs an input stream from given array
     * @param input data
     */
    public InputOutput(byte[] input)
    {
        this.bin = new ByteArrayInputStream(input);
        this.din = new DataInputStream(this.bin);


    }

    /**
     * Starts an output stream
     */
    public void stratOutputStream()
    {
        this.bout = new ByteArrayOutputStream();
        this.dout = new DataOutputStream(this.bout);

    }

    /**
```

```
     * Closes the output stream
     */
    public void closeOutputStream()
    {
        try {
            this.dout.close();
            this.bout.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    /**
     * Start an input stream from the given array
     * @param input data
     */
    public void startInputStream(byte[] input)
    {
        this.bin = new ByteArrayInputStream(input);
        this.din = new DataInputStream(this.bin);

    }


    /**
     * Closes the input stream
     */
    public void closeInputStream()
    {
        try {
            this.din.close();
            this.bin.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

```
package rmp;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;

import rmpAPI.ApplicationMessageAPI;


/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 *This class holds the RMP level functions common for both the server and the client
 */
public class RMP
{
    protected DatagramPacket toSend;
    protected DatagramPacket toReceive;
    protected DatagramSocket udp;
    protected int TX_delay; //Transmission delay
    protected int reTXT;//Retransmission delay
    protected int retransmissionCount;
    protected int reassemblingTimeoutPeriod;
    protected boolean encore;
    protected int receivingUDPBufferLength;
    protected byte[] receivingUDPBuffer;
    protected int msgID;
    protected int PMTU;
    protected int error = 0;
    protected int sessionid;
    protected int updPort;
    protected ApplicationMessageAPI sending;
    protected ApplicationMessageAPI receiving;
    protected RMPPacket send;
    protected RMPPacket receive;
    int counter = 0;
    int counter2 = 0;


    public RMP()
    {
        this.msgID = 1;
        this.PMTU = 1000;
    }

    /**
     * Protocol for sending PDUs
     * @return -1 if error occurs during the sending process
     */
```

```
    protected boolean send(RMPPacket send)
    {
        boolean err = true;
        //generate a session ID for the packet
        send.generateSessionID();
        //Check the packet length
        System.out.println("Message DAata Length sendding: " + send.getMsgDataLength
();
        if(send.getMsgDataLength() > this.PMTU)
        {
            //if the message is to be fragmented
            //initiate the fragementation object
            RMPFragmentationReassembling frag = new RMPFragmentationReassembling(send
, this.PMTU);
            //send the address info
            //calculate the fragment amount
            frag.calculateFragmentAmount();
            //fragment the message
            frag.fragmentComplete();
            //start sending
            for(int i = 0; i < frag.getFragmentAmount(); i++)
            {
                //Initiate the encapsulation
                RMPPacketEncapsulationToDatagramPacket en = new
RMPPacketEncapsulationToDatagramPacket(frag.getFragments()[i]);
                //Encapsulate
                en.toUDPPacketFromRMPPacket();
                //send
                err = this.passToUDP(en.getUdpPacket(), frag.getFragments()[i].
getRMPMode(), false, frag.getFragments()[i].getMsgID(), this.reTXT);
                if(!err)
                    return err;
                if(i+1 == frag.getFragmentAmount())
                    break;//break the loop if the last fragment was sent
                try
                {
                    Thread.sleep(this.TX_delay);//Wait the transmission delay between
the fragments
                }
                catch (InterruptedException e)
                {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

            }
            return err;
        }
        //the complete message
        else
        {
            System.out.println("Sending RMP packet with AF set to " + send.getAF());
```

```
            //Initiate the encapsulation
            RMPPacketEncapsulationToDatagramPacket en = new
RMPPacketEncapsulationToDatagramPacket(send);
            //Encapsulate
            en.toUDPPacketFromRMPPacket();

            //send
            return this.passToUDP(en.getUdpPacket(), this.send.getRMPMode(), false,
this.send.getSessionID(), this.reTXT);
        }

    }


    /**
     * Passes the packet to udp. Includes the retransmission loop for RMP level ACKs.
     * @param toUDP packet to be passed to UDP
     * @param rmpMode State of the RMP mode bit on the packet
     * @param RMPType   State of the RMP packet type bit
     * @param sessionID session identification of the packet
     * @param reTransmissionTimout reTransmission timeout
     * @return True: packet successfully passed to UDP. False: packet was not passed
     */
    protected boolean passToUDP(DatagramPacket toUDP, boolean rmpMode, boolean
RMPType, int msgID, int reTransmissionTimout)
    {
            int count = this.retransmissionCount;
            //initiate encore
            this.encore = true;
            while(this.encore)
            {
                try
                {
                    //send
                    this.udp.send(toUDP);
                }
                catch (IOException e)
                {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                    return false;
                }
                if(RMPType)
                    break; //if the packet was a ACK packet break the loop
immediately
                //check if ack is expected
                if(rmpMode)
                {
                    //go to receive the ACK PDU

                    this.encore = this.receiveAcknowledgment(msgID,
```

```
reTransmissionTimout);
                }
                else
                {
                    //terminate the loop
                    this.encore = false;
                }
                //decrement the retransmission count and check if there is still
retries left
                if((--count) == 0)
                {

                    this.encore = false;//if not terminate the loop
                }
            }

            //return false if the retransmissions expired
            if(count == 0)
                return false;

            return true;//otherwise return success

    }

    /**
     * Receive DatagramPacket from UDP
     * @param timeout How long the UDP socket blocks for receiving
     * @return True: something was received. False: timeout occurred
     */
    protected boolean getFromUDP(int timeout)
    {
    //init the receiving DatagramPacket
        this.receivingUDPBuffer = new byte[this.receivingUDPBufferLength];
        this.toReceive = new DatagramPacket(this.receivingUDPBuffer, this.
receivingUDPBufferLength);

            //set timeout
            try
            {
                this.udp.setSoTimeout(timeout);
            }
            catch (SocketException e)
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        try
        {
            //receive
            this.udp.receive(this.toReceive);
        }catch (IOException e)
```

```
        {

            // TODO Auto-generated catch block
        //  e.printStackTrace();
            return false;
        }
        System.out.println("Received UDP packet with data lenght of " + this.
toReceive.getLength());
        return true;

    }

    /**
     * Generates and sends the ackPDU
     * @param toBeAcknowledged
     * @return see passToUDP
     */
    protected boolean sendAcknowledgment(RMPPacket toBeAcknowledged)
    {
        System.out.println("ACK SENT!!! " + counter2++);
        //create the ackPDU
        RMPPacket ackPDU = toBeAcknowledged.cloneHeader();
        //Set the flags
        ackPDU.setRMPMode(false);
        ackPDU.setPDUType(true);
        ackPDU.setAF(false);
//      ackPDU.setMsgDataLength(0);
        //initiate a local encapsulation object
        RMPPacketEncapsulationToDatagramPacket en = new
RMPPacketEncapsulationToDatagramPacket(ackPDU);
        //Encapsulate
        en.toUDPPacketFromRMPPacket();
        //send
        return this.passToUDP(en.getUdpPacket(), ackPDU.getRMPMode(), ackPDU.
getPDUType(), ackPDU.getSessionID(), 0);
    }

    /**
     * Receives and checks the ackPDU
     * @param toBeAcknowledged
     * @param timeout
     * @return
     */
    protected boolean receiveAcknowledgment(int msgID, int timeout)
    {

        System.out.println("ACK RECEIVED!!! " + counter++);


        //if udp reception fails
        if(!this.getFromUDP(timeout))
            return true; //receiving fails return true for retransmission
```

```
        //Decapsulate
        RMPPacketEncapsulationToDatagramPacket decap = new
RMPPacketEncapsulationToDatagramPacket(this.toReceive);
        decap.toRMPPacketFromUDPPacket();
        //check if the received packet is an ACK PDU
        System.out.println("ACK RECEIVED!!! type: " + decap.getRmpPacket().getPDUType
() + " received ID. " + decap.getRmpPacket().getMsgID() + " sent ID: " + msgID);
        if(decap.getRmpPacket().getPDUType())
        {
//          System.out.println("ACK RECEIVED!!! Match" + counter);
            //If true, check that it was a correct one
            if(msgID == decap.getRmpPacket().getMsgID())
            {
                System.out.println("ACK RECEIVED!!! Match" + counter);
            //ACK matches
                return false;//to cut the retransmission loop
            }
            else
            {
                //ACK did not match
                return true; //to continue the retransmission
            }
        }
        else
        {
            //a wrong packet was received
            return true;//to continue the retransmission
        }



    }

    /**
     * Receives the DatagramPacket, decapsulates it to RMPPacket, and sends an ACK
PDU if required
     * @return True: RMPPacket successfully received and stored in receive field.
False: error while receiving or while sending the ACK PDU
     */
    protected boolean receive()
    {
        //start revceiving
        if(!(getFromUDP(this.reTXT)))
            return false;
        //decapsulate the udp packet
        RMPPacketEncapsulationToDatagramPacket decap = new
RMPPacketEncapsulationToDatagramPacket(this.toReceive);
        //decapsulate
        decap.toRMPPacketFromUDPPacket();
        //get the RMPPacket
        this.receive = decap.getRmpPacket();
        System.out.println("Received RMP packet has AF set to " + this.receive.getAF
```

```
());
        //check if the packet need to be acknowledged
        if(this.receive.getRMPMode())
            return this.sendAcknowledgment(this.receive);
        //return
        return true;//return true for successful reception
    }

    /**
     * Generates the message ID and store it into msgID field
     */
    protected void generateMsgID()
    {
        if(this.msgID == 255)
        {
            this.msgID = 1;
        }
        else
        {
        this.msgID++;
        }

    }

}
```

```
package rmp;



import java.net.BindException;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;



import rmpAPI.ApplicationMessageAPI;
import rmpAPI.RMPClientAPI;

/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 *This class holds the RMP level functions specific for the client.
 */
public class RMPClient extends RMP implements RMPClientAPI {


        private int deviceCategory;
        private int deviceID;
        private int RMS_PORT;
        private int AS_PORT;
        private InetAddress RMS_IP;
        private InetAddress AS_IP;
        private boolean isOpen;




        public RMPClient()
        {
            //Construct the RMP
            super();
            this.isOpen = false;

            //set the client state to not configured

        }

    /* (non-Javadoc)
     * @see rmpAPI.RMPClientAPI#configure(int, int, int, java.net.InetAddress, java.
lang.String, java.net.InetAddress, java.lang.String, int, int, int, int, int)
     */
    @Override
    public int configure(int deviceCategory, int deviceID, int PMTU,
            InetAddress RMS_IP, InetAddress AS_IP, int RMS_PORT, int AS_PORT, int
```

```
LOCAL_PORT,
            int TX_delay, int reTXT, int udpBufferLength, int
retransmissionRetryCount, int reassemblingTimeout)
    {
        // TODO Auto-generated method stub
            this.deviceCategory = deviceCategory;
            this.deviceID = deviceID;
            this.RMS_IP = RMS_IP;
            this.AS_IP = AS_IP;
            this.RMS_PORT = RMS_PORT;
            this.AS_PORT = AS_PORT;
            super.updPort = LOCAL_PORT;
            super.TX_delay = TX_delay;
            super.reTXT = reTXT;
            super.PMTU = PMTU;
            super.receivingUDPBufferLength = udpBufferLength;
            super.retransmissionCount = retransmissionRetryCount;
            super.reassemblingTimeoutPeriod = reassemblingTimeout;

            if(this.isOpen)
                return 0;
            System.out.println("Client opens the socket");
            try
            {
                super.udp = new DatagramSocket(super.updPort);
                this.isOpen = true;
            }
            catch (SocketException e)
            {
                // TODO Auto-generated catch block

            }
            //System.out.println("Configured");
            return 0;
    }

    /* (non-Javadoc)
     * @see rmpAPI.RMPClientAPI#receiveMsg(rmpAPI.ApplicationMessageAPI, int)
     */
    @Override
    public int receiveMsg(ApplicationMessageAPI callback, int timeout)
    {
        // TODO Auto-generated method stub
            //save the message reference
            super.receiving = callback;
            super.reTXT = timeout;
            //receive
            if(this.clientReceive())
                return 0;//successful reception
            return -1; //error in reception
```

```
}

/* (non-Javadoc)
 * @see rmpAPI.RMPClientAPI#receiveMsg(rmpAPI.ApplicationMessageAPI)
 */
@Override
public int receiveMsg(ApplicationMessageAPI callback)
{
    // TODO Auto-generated method stub
        super.receiving = callback;
        return 0;
}

/* (non-Javadoc)
 * @see rmpAPI.RMPClientAPI#sendMsg(int, byte[], boolean, int)
 */
@Override
public int sendMsg(ApplicationMessageAPI callback, int timeout)
{
    // TODO Auto-generated method stub


        //save the message reference
        super.sending = callback;
        super.reTXT = timeout;
        //send
        if(this.clientSend())
            return 0;//message was successfully sent
        return -1; //error while sending

}

/* (non-Javadoc)
 * @see rmpAPI.RMPClientAPI#sendMsg(int, byte[], boolean)
 */
@Override
public int sendMsg(ApplicationMessageAPI callback)
{
    // TODO Auto-generated method stub
    return 0;
}

/* (non-Javadoc)
 * @see rmpAPI.RMPClientAPI#startPMTU(rmpAPI.ApplicationMessageAPI, int)
 */
@Override
public int startPMTU(ApplicationMessageAPI callback, int timeout)
{
    // TODO Auto-generated method stub
    return 0;

}
```

```
    /* (non-Javadoc)
     * @see rmpAPI.RMPClientAPI#startPMTU(rmpAPI.ApplicationMessageAPI)
     */
    @Override
    public int startPMTU(ApplicationMessageAPI callback)
    {
        // TODO Auto-generated method stub
        return 0;

    }

    /**
     *
     * @param callback
     * @return
     */
    private boolean clientSend()
    {

        //Generate a new message ID
        super.generateMsgID();
        //initiate a new RMPPacket
        super.send = new RMPPacket(this.deviceCategory, super.sending.getRMPMode(),
super.sending.getMsgType(), super.msgID, this.deviceID, super.sending.getAppMsg(),
this.RMS_IP, this.RMS_PORT);
        //send
        return super.send(super.send);
    }


    private boolean clientReceive()
    {
            //start receiving the packet
            if(!(super.receive()))
            {
                //Error while receiving return
                //close the socket
//              super.udp.close();
                return false;
            }
                        //if the receiving message is empty continue to receive a new
packet
                    //check if received packet is a fragment
                    if(super.receive.getFragmentAmount() > 0)
                    {

                        //initiate a reassembling object
                        RMPFragmentationReassembling reass = new
RMPFragmentationReassembling(super.receive, super.receive.getFragmentAmount(), super.
receive.getFragmentNumber(), super.receive.getMsgID(), System.currentTimeMillis());
                        //Set the message length for the session
```

```
                        reass.setMessageDataLength(super.receive.getMsgDataLength());
                        //set message type for the session
                        reass.setMsgType(super.receive.getMsgType());
                        //start receiving the rest of the fragments
                        while(!reass.hasAllFragments())
                        {
                            System.out.println("receiving fragments for " + reass.
getMsgType() + "type message" );
                            //if the reception fails or timeout occurs
                            if(!(super.receive()) || (System.currentTimeMillis() -
reass.getReassemblingSessionStartTime()) > super.reassemblingTimeoutPeriod)
                            {
//                              //close the socket
//                              super.udp.close();
                                return false; //error while receiving
                            }
                            //compare message IDs
                            if(reass.getmsgID() != super.receive.getMsgID())
                            {
                                //if no match break the reassembling and save the
received message of the new type for the application
                                super.receiving.msgReceived(super.receive.getAF(),
super.receive.getMsgType(), super.receive.getDeviceID(), super.receive.getPayload());
//                              //close the socket
//                              super.udp.close();
                                return true;
                            }
                            //The packet seems to be valid. Add the packet into the
reassembling object
                            reass.setNewFragment(super.receive, super.receive.
getFragmentNumber());
                        }

                        //when all the fragments are received reassemble the
fragments
                        reass.reAssembleFragments();
                        //Save the complete message for the application
                        super.receiving.msgReceived(super.receive.getAF(), super.
receive.getMsgType(), super.receive.getDeviceID(), reass.getCompleteMessage());
//                      //close the socket
//                      super.udp.close();
                        return true;
                }
                //if the message is complete
                else
                {
                    //save the message to the application and return for successful
reception
                    super.receiving.msgReceived(super.receive.getAF(), super.receive.
getMsgType(), super.receive.getDeviceID(), super.receive.getPayload());
//                  //close the socket
//                  super.udp.close();
```

```
                    return true;
                }


        }


    }
```

```
package rmpAPI;

import java.net.InetAddress;
import java.net.UnknownHostException;


/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 * The RMP client implements the RMPClientAPI-interface. The client application can
configure RMP parameters, and send and receive messages.
 */

public interface RMPClientAPI
{


    /**
     * Sets the RMP client system parameters
     * @param deviceCategory
     * @param deviceID
     * @param PMTU
     * @param RMS_IP
     * @param AS_IP
     * @param RMS_PORT
     * @param AS_PORT
     * @param LOCAL_PORT
     * @param TX_delay
     * @param reTX_T
     * @param udpBufferLength
     * @param reTransmisssionRetryCount
     * @param reassemblingTimeout
     * @return
     */
    public int configure(int deviceCategory, int deviceID, int PMTU, InetAddress
RMS_IP, InetAddress AS_IP, int RMS_PORT, int AS_PORT, int LOCAL_PORT, int TX_delay,
int reTX_T, int udpBufferLength, int reTransmisssionRetryCount, int
reassemblingTimeout);

    /**
     * Blocking method for sending an application message. Returns after the message
is passed to UDP. Application shall perform the memory management for msg.
     * @param msg - Reference to the RMP_Msg object
     * @return - -1: Error
     */
    public int sendMsg(ApplicationMessageAPI callback, int timeout);

    /**
     * Non-blocking method for sending an application message. RMP creates a new
thread for the sending process, copies the message object, and returns.
```

```
     * @param msg - Reference to the message object. Application shall allocate the
memory for the message Object. RMP shall remove the message object from the memory.
     * @param RACK - True: RMP acknowledged mode. False: RMP unacknowledged mode
     * @return - -1: Error
     */
    public int sendMsg(ApplicationMessageAPI callback);


    /**
     * Blocking method for receiving an application message. Returns if a message is
received or after the timeout. RMP shall allocate the memory for the message object.
Application shall remove the message object from the memory.
     * @param timeout - Defines the period after when the method shall return if no
message has been received.
     * @return - the reference to the received message object
     */
    public int receiveMsg(ApplicationMessageAPI callback, int timeout);

    /**
     * Non-blocking method for receiving an application message. RMP creates a new
thread for the receiving process and returns. Once a message is received, RMP calls
the msgReceived -callback method to pass the reference of the message object to the
application. (callback.msgReceived(msg))
     * @param callback - Reference to the application interface.
     * @return - -1: Error
     */
    public int receiveMsg(ApplicationMessageAPI callback);

    /**
     * Blocking type: start the PMTU discovery procedure. RMP indicates the
application about the procedure status using callback.pathMTUDiscoveryResult(status)
-callback method.
     * @param callback - Reference to the application interface.
     */
    public int startPMTU(ApplicationMessageAPI callback, int timeout);

    /**
     * Non-blocking type: start the PMTU discovery procedure. RMP indicates the
application about the procedure status using callback.pathMTUDiscoveryResult(status)
-callback method.
     * @param callback - Reference to the application interface.
     */
    public int startPMTU(ApplicationMessageAPI callback);

}
```

```
package rmp;

import java.io.IOException;


/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 * This class is used to fragment and reassemble RMP packets
 */
public class RMPFragmentationReassembling
{
    private RMPPacket completePacket;
    private byte[] completeMessage;
    private RMPPacket[] fragments;
    private boolean isFragmented;
    private boolean isCompleted;
    private int messageDataLength;
    private int PMTU;
    private int fragmentAmount;
    private int sizeOfTheLastFragment;
    private InputOutput streams;
    private int sessionID;
    private int msgID;
    private long reassemblingSessionStartTime;
    private int fragCounter;
    private int msgType;

    /**
     * Constructs an empty object
     */
    public RMPFragmentationReassembling()
    {
        // TODO Auto-generated constructor stub
    }

    /**
     * Constructs an fragmentation object
     * @param complete message to be fragmented
     */
    public RMPFragmentationReassembling(RMPPacket completePacket, int PMTU)
    {
        // TODO Auto-generated constructor stub
        this.setCompletePacket(completePacket);
        this.setCompleteMessage(this.completePacket.getPayload());
        this.setMessageDataLength(this.completeMessage.length);
        this.setPMTU(PMTU);
    }

    /**
```

```
     * Constructs a reassembling object. Initiates the fragment array based on the
value of fragmentAmount
     * @param frag first received fragment
     * @param i fragment number
     */
    public RMPFragmentationReassembling(RMPPacket frag, int fragAmount, int
fragNumber, int msgID, long startTime)
    {
        // TODO Auto-generated constructor stub
        this.fragCounter = 0;
        this.setFragmentAmount(fragAmount);
        this.initiateTheFragmentArray(this.fragmentAmount);
        this.setNewFragment(frag, fragNumber);
        this.setmsgID(msgID);
        this.setReassemblingSessionStartTime(startTime);
    }

    /**
     * Initiates the length of the fragment array
     * @param fragmentAmount length of the array
     */
    public void initiateTheFragmentArray(int fragmentAmount)
    {
        this.fragments = new RMPPacket[fragmentAmount];
    }
    /**
     * @return the completePacket
     */
    public RMPPacket getCompletePacket() {
        return completePacket;
    }

    /**
     * @param completePacket the completePacket to set
     */
    public void setCompletePacket(RMPPacket completePacket) {
        this.completePacket = completePacket;
    }

    /**
     * @return the completeMessage
     */
    public byte[] getCompleteMessage() {
        return completeMessage;
    }
    /**
     * set also the message data length
     * @param completeMessage the completeMessage to set
     */
    public void setCompleteMessage(byte[] completeMessage) {
        this.completeMessage = completeMessage;
        this.messageDataLength = this.completeMessage.length;
```

```
}
/**
 * @return the fragments
 */
public RMPPacket[] getFragments() {
    return fragments;
}
/**
 * @param fragments the fragments to set
 */
public void setFragments(RMPPacket[] fragments) {
    this.fragments = fragments;
}

/**
 * @return the messageDataLength
 */
public int getMessageDataLength() {
    return messageDataLength;
}

/**
 * @param messageDataLength the messageDataLength to set
 */
public void setMessageDataLength(int messageDataLength) {
    this.messageDataLength = messageDataLength;
}

/**
 * @return the pMTU
 */
public int getPMTU() {
    return PMTU;
}

/**
 * @param pMTU the pMTU to set
 */
public void setPMTU(int pMTU) {
    PMTU = pMTU;
}

/**
 * @return the fragmentAmount
 */
public int getFragmentAmount() {
    return fragmentAmount;
}

/**
 * @param fragmentAmount the fragmentAmount to set
 */
```

```
public void setFragmentAmount(int fragmentAmount) {
    this.fragmentAmount = fragmentAmount;
}

/**
 * @return the sessionID
 */
public int getSessionID() {
    return sessionID;
}

/**
 * @param sessionID the sessionID to set
 */
public void setSessionID(int sessionID) {
    this.sessionID = sessionID;
}

/**
 * @return the reassemblingID
 */
public int getmsgID() {
    return msgID;
}

/**
 * @param reassemblingID the reassemblingID to set
 */
public void setmsgID(int msgID) {
    this.msgID = msgID;
}

/**
 * @return the reassemblingSessionStartTime
 */
public long getReassemblingSessionStartTime() {
    return reassemblingSessionStartTime;
}

/**
 * @param reassemblingSessionStartTime the reassemblingSessionStartTime to set
 */
public void setReassemblingSessionStartTime(long reassemblingSessionStartTime) {
    this.reassemblingSessionStartTime = reassemblingSessionStartTime;
}

/**
 * Add a fragment into the fragment array to the given position i
 * @param frag
 * @param i
 */
public void setNewFragment(RMPPacket frag, int fragNumber)
```

```
{
    this.fragments[(fragNumber-1)] = frag;
    this.fragCounter++;
}

/**
 * @return the msgType
 */
public int getMsgType() {
    return msgType;
}

/**
 * @param msgType the msgType to set
 */
public void setMsgType(int msgType) {
    this.msgType = msgType;
}

/**
 * Return true if the object has all the fragments of the complete message
 * @return True: All fragments False: some or all the fragments are missing
 */
public synchronized boolean hasAllFragments()
{
        if(this.fragmentAmount == this.fragCounter)
        {

            return true;
        }
        else
        {
            return false;
        }




}
/**
 * @return the isFragmented
 */
public synchronized boolean isFragmented() {
    return isFragmented;
}
/**
 * @param isFragmented the isFragmented to set
 */
public synchronized void setFragmented(boolean isFragmented) {
    this.isFragmented = isFragmented;
}
/**
```

```
     * @return the isCompleted
     */
    public synchronized boolean isCompleted() {
        return isCompleted;
    }
    /**
     * @param isCompleted the isCompleted to set
     */
    public synchronized void setCompleted(boolean isCompleted) {
        this.isCompleted = isCompleted;
    }

    /**
     * calculates the total amount of fragments. PMTU and MessgeDataLength must be
set before hand
     */
    public void calculateFragmentAmount()
    {
        //calc the number of full fragments
        this.fragmentAmount = this.messageDataLength / this.PMTU;
        //calc the size of the last fragment
        this.sizeOfTheLastFragment = this.messageDataLength % this.PMTU;

        //resolve the need for increment
        if(this.sizeOfTheLastFragment != 0)
        {
            this.fragmentAmount++;
        }

    }


    /**
     *Fragments the complete message into number of fragments defined by the PMTU and
the message length.
     *Creates a new RMPFragment object with corresponding fragment data (fragamount,
fragnumber) for each fragment
     */
    public void fragmentComplete()
    {
        //Allocate the RMPFragment array
        this.fragments = new RMPPacket[this.fragmentAmount];
        //Initiate a local byte count
        int byteCount = 0;
        //generate the full fragments
        for(int i = 0; i < (this.fragmentAmount - 1); i++)
        {
            //Initiate the InputOutput and open the output stream
            this.streams = new InputOutput();
            this.streams.stratOutputStream();
            try
            {
```

```
                this.streams.dout.write(this.completeMessage, byteCount, this.PMTU);
            }
            catch (IOException e)
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            //create a new RMPfragment object from the output stream and the given
parameters
            this.fragments[i] = new RMPPacket(this.completePacket.getDeviceCategory()
, this.completePacket.getFlag(), this.completePacket.getMsgType(), this.
completePacket.getMsgID(), this.completePacket.getDeviceID(), this.messageDataLength,
this.fragmentAmount,(i+1), this.streams.bout.toByteArray(), this.completePacket.getIP
(), this.completePacket.getUdpPort());
            //close the output stream
            this.streams.closeOutputStream();
            //generate a new session ID for the fragment
            this.fragments[i].generateSessionID();
            //update the counter
            byteCount += this.PMTU;
        }
        //generate the final fragment if needed
        if(this.sizeOfTheLastFragment > 0)
        {
            //Initiate the InputOutput and open the output stream
            this.streams = new InputOutput();
            this.streams.stratOutputStream();
            try
            {
                this.streams.dout.write(this.completeMessage, byteCount, this.
sizeOfTheLastFragment);
            }
            catch (IOException e)
            {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            System.out.println("fragments length: " + fragments.length);
            //create a new RMPfragment object from the output stream and the given
parameters
            fragments[this.fragmentAmount-1] = new RMPPacket(this.completePacket.
getDeviceCategory(), this.completePacket.getFlag(), this.completePacket.getMsgType(),
this.completePacket.getMsgID(), this.completePacket.getDeviceID(), this.
messageDataLength, this.fragmentAmount,this.fragmentAmount, this.streams.bout.
toByteArray(), this.completePacket.getIP(), this.completePacket.getUdpPort(), this.
sessionID);
            //close the output stream
            this.streams.closeOutputStream();

        }

    }
```

```
    /**
     * Reassembles the fragments added into the object. It must be sure that all the
fragments have been added
     * into the fragments array
     */
    public void reAssembleFragments()
    {

        //Initite the complete message array
        this.completeMessage = new byte[this.messageDataLength];
        //Initiate a offSet field
        int offSet = 0;
        //Loop as many times there are fragments
        for(int i = 0; i < this.fragments.length; i++)
        {
            //check the length of the fragment
            int fragLength = this.fragments[i].getPayload().length;
            //init a counter for fragment array
            int count = 0;
            //Add fragment into the complete array
            System.out.println("fragments length; " +this.fragments.length +"offSet:
" + offSet + " fragLength: " + fragLength + " count: " + count + " Message data
length: " + this.messageDataLength);
            for(int j = offSet; j < (offSet+fragLength); j++ )
            {
                this.completeMessage[j] = this.fragments[i].getPayload()[count++];
            }
            //adjust the offset for the next fragment
            offSet = fragLength;
        }

    }


}
```

```
package rmp;

import java.io.IOException;
import java.net.InetAddress;

/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 * This class holds the methods and data of RMP packet.
 */
public class RMPPacket
{

        /**
         *
         */
        private int deviceCategory;
        private int flag;
        private int msgType;
        private int msgID;
        private int deviceID;
        private int msgDataLength;
        private int fragmentAmount;
        private int fragmentNumber;
        private byte[] payload;
        private InetAddress IP;
        private int udpPort;
        private int sessionID;
        private int reassemblingID;
        private InputOutput streams;

        /**
         * constructs an empty fragment object
         */
        public RMPPacket()
        {
            //set fragAmount to zero
            this.fragmentAmount = 0;
            this.fragmentNumber = 0;
        }




        /**
         * Cosnstructs a RMPPacket from given parameters. used at the client end
         * @param deviceCategory
         * @param RACK
         * @param msgType
```

```
         * @param msgID
         * @param deviceID
         * @param payload
         * @param iP
         * @param udpPort
         */
        public RMPPacket(int deviceCategory, boolean RACK, int msgType, int msgID,
int deviceID, byte[] payload, InetAddress iP, int udpPort) {

            this.deviceCategory = deviceCategory;
            this.setRMPMode(RACK);
            this.msgType = msgType;
            this.msgID = msgID;
            this.deviceID = deviceID;
            this.payload = payload;
            IP = iP;
            this.udpPort = udpPort;
            try
            {
            this.msgDataLength = this.payload.length;
            }
            catch(NullPointerException e)
            {

            }
            //set fragAmount to zero
            this.fragmentAmount = 0;
            this.fragmentNumber = 0;
        }
        /**
         * Cosnstructs a RMPPacket from given parameters. used at the server end
         * @param deviceCategory
         * @param AF
         * @param msgType
         * @param msgID
         * @param deviceID
         * @param payload
         * @param iP
         * @param udpPort
         */
        public RMPPacket(int deviceCategory, boolean AF, boolean RACK, int msgType,
int msgID, int deviceID, byte[] payload, InetAddress iP, int udpPort) {

            this.deviceCategory = deviceCategory;
            this.setAF(AF);
            this.setRMPMode(RACK);
            this.msgType = msgType;
            this.msgID = msgID;
            this.deviceID = deviceID;
            this.payload = payload;
            IP = iP;
            this.udpPort = udpPort;
```

```
            this.msgDataLength = this.payload.length;
            //set fragAmount to zero
            this.fragmentAmount = 0;
            this.fragmentNumber = 0;
        }




        /**
         * Constructs a header object
         * @param deviceCategory
         * @param flag
         * @param msgType
         * @param msgID
         * @param deviceID
         * @param msgDataLength
         * @param fragmentAmount
         * @param fragmentNumber
         * @param payload
         * @param iP
         * @param udpPort
         */
        public RMPPacket(int deviceCategory, int flag, int msgType, int msgID, int
deviceID, int msgDataLength, int fragmentAmount, int fragmentNumber, InetAddress iP,
int udpPort)
        {
            this.deviceCategory = deviceCategory;
            this.flag = flag;
            this.msgType = msgType;
            this.msgID = msgID;
            this.deviceID = deviceID;
            this.msgDataLength = msgDataLength;
            this.fragmentAmount = fragmentAmount;
            this.fragmentNumber = fragmentNumber;
            this.IP = iP;
            this.udpPort = udpPort;
            //set fragAmount to zero

        }




        /**
         * Constructs a RMPPacket with all the field values
         * @param deviceCategory
         * @param flag
         * @param msgType
         * @param msgID
```

```
         * @param deviceID
         * @param msgDataLength
         * @param fragmentAmount
         * @param fragmentNumber
         * @param payload
         * @param iP
         * @param udpPort
         * @param sessionID
         */
        public RMPPacket(int deviceCategory, int flag, int msgType, int msgID, int
deviceID, int msgDataLength, int fragmentAmount, int fragmentNumber, byte[] payload,
InetAddress IP, int udpPort, int sessionID) {
            this.deviceCategory = deviceCategory;
            this.flag = flag;
            this.msgType = msgType;
            this.msgID = msgID;
            this.deviceID = deviceID;
            this.msgDataLength = msgDataLength;
            this.fragmentAmount = fragmentAmount;
            this.fragmentNumber = fragmentNumber;
            this.payload = payload;
            this.IP = IP;
            this.udpPort = udpPort;
            this.sessionID = sessionID;
        }

        /**
         * constructs a fragment object
         * Constructs a RMPPacket from the given parameters
         * @param deviceCategory
         * @param flag
         * @param msgType
         * @param msgID
         * @param deviceID
         * @param msgDataLength
         * @param fragmentAmount
         * @param fragmentNumber
         * @param payload
         * @param iP
         * @param udpPort
         * @param sessionID
         */
        public RMPPacket(int deviceCategory, int flag, int msgType, int msgID, int
deviceID, int msgDataLength, int fragmentAmount, int fragmentNumber, byte[] payload,
InetAddress IP, int udpPort)
        {
            this.deviceCategory = deviceCategory;
            this.flag = flag;
            this.msgType = msgType;
            this.msgID = msgID;
            this.deviceID = deviceID;
            this.msgDataLength = msgDataLength;
```

```
            this.fragmentAmount = fragmentAmount;
            this.fragmentNumber = fragmentNumber;
            this.payload = payload;
            this.IP = IP;
            this.udpPort = udpPort;

        }




        /**
         * Constructs a header object
         * @param deviceCategory
         * @param flag
         * @param msgType
         * @param msgID
         * @param deviceID
         * @param msgDataLength
         * @param fragmentAmount
         * @param fragmentNumber
         * @param payload
         */
        public RMPPacket(int deviceCategory, int flag, int msgType, int msgID, int
deviceID, int msgDataLength, int fragmentAmount, int fragmentNumber)
        {
            super();
            this.deviceCategory = deviceCategory;
            this.flag = flag;
            this.msgType = msgType;
            this.msgID = msgID;
            this.deviceID = deviceID;
            this.msgDataLength = msgDataLength;
            this.fragmentAmount = fragmentAmount;
            this.fragmentNumber = fragmentNumber;

        }


        /**
         * @return the deviceCategory
         */
        public int getDeviceCategory() {
            return deviceCategory;
        }


        /**
         * @param deviceCategory the deviceCategory to set
         */
        public void setDeviceCategory(int deviceCategory) {
```

```
        this.deviceCategory = deviceCategory;
    }


    /**
     * @return the flag
     */
    public int getFlag() {
        return flag;
    }


    /**
     * @param flag the flag to set
     */
    public void setFlag(int flag) {
        this.flag = flag;
    }


    /**
     * Sets the PDU type flag in the RMP header
     * @param flag
     */
    public void setPDUType(boolean mode)
    {

        this.flag = setBit(this.flag, 0, mode);
    }

    /**
     * Gets the status PDU type flag from the RMP header
     * @return True: ack PDU, False: data PDU
     */
    public boolean getPDUType()
    {

        return readBit(this.flag, 0);
    }

    /**
     * Sets the ACK required bit in the RMP header
     * @param True: RMP acknowledged mode, False: RMP unacknowledged mode
     */
    public void setRMPMode(boolean mode)
    {

        this.flag = setBit(this.flag, 1, mode);
    }

    /**
     * Gets the status ACK required bit from the RMP header
```

```
     * @return True: RMP acknowledged mode, False: RMP unacknowledged mode
     */
    public boolean getRMPMode()
    {

        return readBit(this.flag, 1);
    }

    /**
     * Sets the Action Flag bit in the RMP header
     * @param True: AF set, False: AF cleared
     */
    public void setAF(boolean mode)
    {

        this.flag = setBit(this.flag, 2, mode);
    }
    /**
     * Gets the Action Flag bit from the RMP header
     * @return True: action Flag set, False: action flag clearded
     */
    public boolean getAF()
    {
        return readBit(this.flag, 2);
    }


    /**
     * @return the msgType
     */
    public int getMsgType() {
        return msgType;
    }


    /**
     * @param msgType the msgType to set
     */
    public void setMsgType(int msgType) {
        this.msgType = msgType;
    }


    /**
     * @return the msgID
     */
    public int getMsgID() {
        return msgID;
    }


    /**
```

```
     * @param msgID the msgID to set
     */
    public void setMsgID(int msgID) {
        this.msgID = msgID;
    }


    /**
     * @return the deviceID
     */
    public int getDeviceID() {
        return deviceID;
    }


    /**
     * @param deviceID the deviceID to set
     */
    public void setDeviceID(int deviceID) {
        this.deviceID = deviceID;
    }


    /**
     * @return the msgDataLength
     */
    public int getMsgDataLength() {
        return msgDataLength;
    }

    /**
     * @param msgDataLength the msgDataLength to set
     */
    public void setMsgDataLength(int msgDataLength) {
        this.msgDataLength = msgDataLength;
    }

    /**
     * @return the fragmentAmount
     */
    public int getFragmentAmount() {
        return fragmentAmount;
    }

    /**
     * @param fragmentAmount the fragmentAmount to set
     */
    public void setFragmentAmount(int fragmentAmount) {
        this.fragmentAmount = fragmentAmount;
    }

    /**
```

```
     * @return the fragmentNumber
     */
    public int getFragmentNumber() {
        return fragmentNumber;
    }

    /**
     * @param fragmentNumber the fragmentNumber to set
     */
    public void setFragmentNumber(int fragmentNumber) {
        this.fragmentNumber = fragmentNumber;
    }

    /**
     * @return the fragment
     */
    public byte[] getPayload() {
        return payload;
    }

    /**
     * @param fragment the fragment to set
     */
    public void setPayload(byte[] payload) {
        this.payload = payload;
    }

    /**
     * @return the iP
     */
    public InetAddress getIP() {
        return this.IP;
    }


    /**
     * @param iP the iP to set
     */
    public void setIP(InetAddress iP) {
        this.IP = iP;
    }


    /**
     * @return the udpPort
     */
    public int getUdpPort() {
        return this.udpPort;
    }


    /**
```

```
     * @param udpPort the udpPort to set
     */
    public void setUdpPort(int udpPort) {
        this.udpPort = udpPort;
    }


    /**
     * @return the sessionID
     */
    public int getSessionID() {
        return sessionID;
    }


    /**
     *
     * Generates a reassembling ID for the packet
     */
    public void generateSessionID()
    {
        //declare a local byte array
        byte[] help;
        //initiate a output stream
        this.streams = new InputOutput();
        this.streams.stratOutputStream();

        //read session ID parameters into the output stream
        try
        {
            this.streams.dout.writeByte(this.deviceCategory);
            this.streams.dout.writeByte(this.msgType);
            this.streams.dout.writeByte(this.msgID);
            this.streams.dout.writeInt(this.deviceID);
            this.streams.dout.writeByte(this.msgDataLength);
            this.streams.dout.writeByte(this.fragmentAmount);
            this.streams.dout.writeInt(this.fragmentNumber);
        }
        catch (IOException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        //write the output into the array
        help = this.streams.bout.toByteArray();
        //close the output stream
        this.streams.closeOutputStream();
        //generate the session ID
        this.sessionID = help.hashCode();
    }
    /**
     * @return the reassemblingID
```

```
 */
public int getReassemblingID() {
    return reassemblingID;
}




/**
 *
 * Generates a session ID for the packet
 */
public void generateReassemblingID()
{
    //declare a local byte array
    byte[] help;
    //initiate a output stream
    this.streams = new InputOutput();
    this.streams.stratOutputStream();

    //read session ID parameters into the output stream
    try
    {
        this.streams.dout.writeByte(this.deviceCategory);
        this.streams.dout.writeByte(this.msgType);
        this.streams.dout.writeByte(this.msgID);
        this.streams.dout.writeInt(this.deviceID);
        this.streams.dout.writeByte(this.msgDataLength);
        this.streams.dout.writeByte(this.fragmentAmount);
    }
    catch (IOException e)
    {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    //write the output into the array
    help = this.streams.bout.toByteArray();
    //close the output stream
    this.streams.closeOutputStream();
    //generate the session ID
    this.reassemblingID = help.hashCode();
}


/**
 * Creates a new instance from the object
 */
public RMPPacket clone()
{

    return new RMPPacket(this.deviceCategory, this.flag, this.msgType, this.
```

```
msgID, this.deviceID, this.msgDataLength, this.fragmentAmount, this.fragmentNumber,
this.payload, this.IP, this.udpPort, this.sessionID);


        }

        /**
         * Creates a new header instance from the object
         */
        public RMPPacket cloneHeader()
        {

            return new RMPPacket(this.deviceCategory, this.flag, this.msgType, this.
msgID, this.deviceID, this.msgDataLength, this.fragmentAmount, this.fragmentNumber,
this.IP, this.udpPort);


        }

        /**
         * Sets or clears a bit in a byte
         * @param flag  byte where the bit is to be set or cleared
         * @param bitPos    Defines the position of the bit in flag byte
         * @param stat  True: set the bit, False: clear the bit
         * @return
         */
        private int setBit(int flag, int bitPos, boolean stat)
        {

            if(stat == true)
            {
            return flag |= 1 << bitPos;
            }
            else
            {
            return flag &= ~(1 << bitPos);
            }
        }

        /**
         * Read the bit from the flag byte in a given position.
         * @param flag  the byte from where the bit is to be read
         * @param bitPos    defines the position of the bit in the flag byte
         * @return  True: the bit is set, False: the bit is cleared
         */
        private boolean readBit(int flag, int bitPos)
        {
            int mask = 1;
            mask =  (mask<<bitPos);
            if((flag & mask) == mask)
            {
                return true;
```

```
                }
                else
                {
                    return false;
                }

            }


    }
```

```
package rmp;

import java.io.IOException;
import java.net.DatagramPacket;


import rmpAPI.ApplicationMessageAPI;

/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 * This class holds the methods to encode and decode RMP messages to/from UDP packets
 */
public class RMPPacketEncapsulationToDatagramPacket
{
    private DatagramPacket udpPacket;
    private RMPPacket   rmpPacket;
    private ApplicationMessage appMsg;
    private InputOutput streams;



    /**
     * Constructs an empty encapsulation object
     */
    public RMPPacketEncapsulationToDatagramPacket()
    {

    }



    /**
     * Constructs an encapsulation object from RMPPacket to DatagramPacket
     * or decapsulation object from RMPPacket to ApplicationMessage
     * @param rmpPacket
     */
    public RMPPacketEncapsulationToDatagramPacket(RMPPacket rmpPacket)
    {
        this.rmpPacket = rmpPacket;
    }

    /**
     * Constructs a decapsulation object from DatagramPacket to RMPPacket
     * @param udpPacket
     */
    public RMPPacketEncapsulationToDatagramPacket(DatagramPacket udpPacket)
    {
        this.udpPacket = udpPacket;
    }
```

```
/**
 * Constructs a encapsulation object from ApplicationMessage to RMPPacket
 * @param appMsg
 */
public RMPPacketEncapsulationToDatagramPacket(ApplicationMessageAPI appMsg)
{
    this.appMsg = (ApplicationMessage)appMsg;
}

/**
 * @return the udpPacket
 */
public DatagramPacket getUdpPacket() {
    return udpPacket;
}
/**
 * @param udpPacket the udpPacket to set
 */
public void setUdpPacket(DatagramPacket udpPacket) {
    this.udpPacket = udpPacket;
}
/**
 * @return the rmpPacket
 */
public RMPPacket getRmpPacket() {
    return rmpPacket;
}
/**
 * @param rmpPacket the rmpPacket to set
 */
public void setRmpPacket(RMPPacket rmpPacket) {
    this.rmpPacket = rmpPacket;
}

/**
 * @return the appMsg
 */
public ApplicationMessageAPI getAppMsg() {
    return appMsg;
}

/**
 * @param appMsg the appMsg to set
 */
public void setAppMsg(ApplicationMessageAPI appMsg) {
    this.appMsg = (ApplicationMessage)appMsg;
}

/**
 * @return the streams
 */
public InputOutput getStreams() {
```

```
        return streams;
    }



    /**
     * @param streams the streams to set
     */
    public void setStreams(InputOutput streams) {
        this.streams = streams;
    }



    /**
     * Encapsulates the RMPPacket into a DatagramPacket
     */
    public void toUDPPacketFromRMPPacket()
    {
        //Open a output stream
        this.streams = new InputOutput();
        this.streams.stratOutputStream();
        //write the RMPPacket into the output stream

        try {
            this.streams.dout.writeByte(this.rmpPacket.getDeviceCategory());
            this.streams.dout.writeByte(this.rmpPacket.getFlag());
            this.streams.dout.writeByte(this.rmpPacket.getMsgType());
            this.streams.dout.writeByte(this.rmpPacket.getMsgID());
            this.streams.dout.writeInt(this.rmpPacket.getDeviceID());
            this.streams.dout.writeShort(this.rmpPacket.getMsgDataLength());
            this.streams.dout.writeByte(this.rmpPacket.getFragmentAmount());
            this.streams.dout.writeByte(this.rmpPacket.getFragmentNumber());
            //write the payload only if there is one
            if(this.rmpPacket.getMsgDataLength() != 0 && !this.rmpPacket.getPDUType()
)
                this.streams.dout.write(this.rmpPacket.getPayload());
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            // write the output stream into a byte array
            byte[] data = this.streams.bout.toByteArray();
            //close the ouput stream
            this.streams.closeOutputStream();
            //Initiate a DatagramPacket with the RMPPacket data and the address
information
            this.udpPacket = new DatagramPacket(data, data.length, this.rmpPacket.
getIP(), this.rmpPacket.getUdpPort());
    }

    /**
```

```
     * Decapsultes the RMPPacket from the DatagramPacket.
     */
    public void toRMPPacketFromUDPPacket()
    {
        //read the data from the DatagramPacket to a byte array
        byte[] data = this.udpPacket.getData();
        //Open a output stream
        this.streams = new InputOutput();
        this.streams.startInputStream(data);

        //construct a new RMPPacket from the input stream data
        try
        {
            this.rmpPacket = new RMPPacket(this.streams.din.readUnsignedByte(), this.
streams.din.readUnsignedByte(), this.streams.din.readUnsignedByte(), this.streams.din
.readUnsignedByte(), this.streams.din.readInt(), this.streams.din.readUnsignedShort()
, this.streams.din.readUnsignedByte(), this.streams.din.readUnsignedByte(), this.
udpPacket.getAddress(), this.udpPacket.getPort());
            //read the paylod and add it into the packet if there is one
            if((this.udpPacket.getLength() - 12) != 0)
            {
                //declare and initiate a local field for the RMP payload
                byte[] payload = new byte[(this.udpPacket.getLength() - 12)];
                //read the payload from the input stream
                this.streams.din.read(payload);
                //add the payload into the packet
                this.rmpPacket.setPayload(payload);
            }
        }
        catch (IOException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        //close the input stream
        this.streams.closeInputStream();

    }

    /**
     * Encapsulate the ApplicationMessage into the RMPPacket
     */
    public void toRMPPacketFromApplicationMessage()
    {
        //Initiate the rmpPacket
        this.rmpPacket = new RMPPacket();
        //set values from the application message
        this.rmpPacket.setAF(this.appMsg.getAF());
        this.rmpPacket.setRMPMode(this.appMsg.getRMPMode());
        this.rmpPacket.setDeviceCategory(this.appMsg.getDeviceCategory());
        this.rmpPacket.setDeviceID(this.appMsg.getDeviceID());
        this.rmpPacket.setIP(this.appMsg.getClientIP());
```

```
            this.rmpPacket.setUdpPort(this.appMsg.getClientPort());
            this.rmpPacket.setMsgType(this.appMsg.getMsgType());
            this.rmpPacket.setPayload(this.appMsg.getAppMsg());

        }

        /**
         * Decapsulates the RMPPacket to ApplicationMessage
         */
        public void toApplicationMessageFromRMPPacket()
        {
            //initiate a new application message object
            this.appMsg = new ApplicationMessage();
            //add the values from the RMPPacket
            this.appMsg.composeMsg(this.rmpPacket.getDeviceCategory(), this.rmpPacket.
    getRMPMode(), this.rmpPacket.getAF(), this.rmpPacket.getMsgType(), this.rmpPacket.
    getDeviceID(), this.rmpPacket.getPayload(), this.rmpPacket.getIP(), this.rmpPacket.
    getUdpPort());
        }

    }
```

```
package rmp;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;
import java.util.Arrays;
import java.util.ArrayList;


import rmpAPI.ApplicationMessageAPI;
import rmpAPI.RMPServerAPI;
/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 * This class holds the methods of RMP specific for the server.
 */

public class RMPServer extends RMP implements RMPServerAPI
{
    private int localUDPPort;
    private RMPFragmentationReassembling reass[];
    private int[] idTable;
    private int reassembleCounter;
    //init the receiving DatagramPacket
    private byte[] UDPBuffer;
    private DatagramPacket servReceive;
    private RMPPacket servReceived;
    private ArrayList reAssemblingList;


    public RMPServer()
    {
        super();
        super.receiving = new ApplicationMessage();

    }

    @Override
    public int receiveMsg(ApplicationMessageAPI receiveAppMsg)
    {
        // TODO Auto-generated method stub
        super.receiving = receiveAppMsg;
        if(this.serverReceive())
        {
            return 0;

        }
        return -1;
    }
```

```
    @Override
    public int sendMsg(ApplicationMessageAPI sendAppMsg)
    {
        // TODO Auto-generated method stub

        super.sending = sendAppMsg;
//      //open the socket
//      try
//      {
//          super.udp = new DatagramSocket();
//      }
//      catch (SocketException e)
//      {
//          // TODO Auto-generated catch block
//          e.printStackTrace();
//          return -1;
//      }
        if(this.serverSend())
        {
            //super.udp.close();
            return 0;
        }
        //super.udp.close();
        return -1;
    }

    @Override
    public int setParameters(int PMTU, int RMS_PORT, int TX_delay, int reTXT, int
udpBufferLength, int reTransmisssionRetryCount, int reassemblingTimeout)
    {
        // TODO Auto-generated method stub
        super.PMTU = PMTU;
        this.localUDPPort = RMS_PORT;
        super.TX_delay = TX_delay;
        super.reTXT = reTXT;
        super.receivingUDPBufferLength = udpBufferLength;
        super.retransmissionCount = reTransmisssionRetryCount;
        super.reassemblingTimeoutPeriod = reassemblingTimeout;
        this.reass = new RMPFragmentationReassembling[10];
        this.reAssemblingList = new ArrayList(10);
        this.idTable = new int[10];
        this.reassembleCounter = 0;
        //opent the server socket
        try
        {
            super.udp = new DatagramSocket(this.localUDPPort);
            //System.out.println("server port "+ super.udp.getPort());
        }
        catch (SocketException e1)
        {
            // TODO Auto-generated catch block
            e1.printStackTrace();
```

```
            return -1;
        }

        return 0;
    }

    /**
     * Server receiving protocol
     * @return Error code
     */
    private boolean serverReceive()
    {
        boolean help;

        while(true)
        {

            //reset the reassembling counter if needed
            if((this.reassembleCounter+1) >= this.reass.length)
                this.reassembleCounter = 0;

            //start receiving the packets from server socket
                super.receive();
            //Decapsulate
            RMPPacketEncapsulationToDatagramPacket decap = new
RMPPacketEncapsulationToDatagramPacket(super.toReceive);
            decap.toRMPPacketFromUDPPacket();
            //get the RMPPacket
            super.receive = decap.getRmpPacket();
            //check if received packet is a fragment
            if(super.receive.getFragmentAmount() > 0)
            {
                if(this.reAssemblingList.isEmpty())
                {
                    //initiate a reassembling object
                    this.reass[0] = new RMPFragmentationReassembling(super.receive,
super.receive.getFragmentAmount(), super.receive.getFragmentNumber(), super.receive.
getMsgID(), System.currentTimeMillis());
                    this.reass[0].setMessageDataLength(super.receive.getMsgDataLength
());
                     this.reAssemblingList.add(this.reass[0]);
                     continue;
                }
                else
                {
                    help = false;
                    for(int i = 0; i < this.reAssemblingList.size();i++)
                    {
                        //compare the fragments that they match
                        if(((RMPFragmentationReassembling) this.reAssemblingList.get
(i)).getmsgID() == super.receive.getMsgID())
                        {
```

```
                            ((RMPFragmentationReassembling) this.reAssemblingList.get
(i)).setNewFragment(super.receive, super.receive.getFragmentNumber());
                            if(((RMPFragmentationReassembling) this.reAssemblingList.
get(i)).hasAllFragments())
                            {
                                System.out.println("reassembling index: " + i + "data
length" + ((RMPFragmentationReassembling) this.reAssemblingList.get(i)).getFragments
()[1].getMsgDataLength());
                                ((RMPFragmentationReassembling) this.reAssemblingList
.get(i)).reAssembleFragments();
//                              super.receiving = new ApplicationMessage();

                                super.receiving.msgReceived(super.receive.
getDeviceCategory(), super.receive.getMsgType(), super.receive.getDeviceID(),
((RMPFragmentationReassembling) this.reAssemblingList.get(i)).getCompleteMessage(),
super.receive.getIP(), super.receive.getUdpPort());
                                this.reAssemblingList.remove(i);
                                return true;
                            }
                            else
                            {
                                help = true;
                                break;
                            }
                        }

                    }
                    if(help)
                        continue;
                    this.reass[++this.reassembleCounter] = new
RMPFragmentationReassembling(super.receive, super.receive.getFragmentAmount(), super.
receive.getFragmentNumber(), super.receive.getMsgID(), System.currentTimeMillis());
                    this.reass[++this.reassembleCounter].setMessageDataLength(super.
receive.getMsgDataLength());
                    this.reAssemblingList.add(this.reass[++this.reassembleCounter]);

                    continue;
                }

            }
//          super.receiving = new ApplicationMessage();
            try
            {
            super.receiving.msgReceived(super.receive.getDeviceCategory(), super.
receive.getMsgType(), super.receive.getDeviceID(), super.receive.getPayload(), super.
receive.getIP(), super.receive.getUdpPort());
            }
            catch(NullPointerException r)
            {
                System.out.println("Null pointer error");
                r.printStackTrace();
            }
```

```
                return true;
            }
        }



        private boolean serverSend()
        {
            System.out.println("server local port send "+ super.udp.getLocalPort());
            //Generate a new message ID
            super.generateMsgID();
            //initiate a new RMPPacket
            super.send = new RMPPacket(super.sending.getDeviceCategory(), super.sending.
getAF(), super.sending.getRMPMode(), super.sending.getMsgType(), super.msgID, super.
sending.getDeviceID(), super.sending.getAppMsg(), super.sending.getClientIP(), super.
sending.getClientPort());
            //send
            return super.send(super.send);


        }


    }
```

```
/**
 *
 */
package rmpAPI;

/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 */

public interface RMPServerAPI
{
    /**
     * Sets the RMP server parameters.
     * @param PMTU
     * @param RMS_PORT
     * @param TX_delay
     * @param reTX_T
     * @param udpBufferLength
     * @param reTransmisssionRetryCount
     * @param reassemblingTimeout
     * @return -1 if error occurs and 0: for successful sending
     */
    public int setParameters(int PMTU, int RMS_PORT, int TX_delay, int reTX_T, int
udpBufferLength, int reTransmisssionRetryCount, int reassemblingTimeout);

    /**
     * Non-blocking method for sending an application message. RMP creates a new
thread for the sending process, copies the message object, and returns.
     * @param sendAppMsg object that holds the application message data to be sent
     * @return -1 if error occurs and 0: for successful sending
     */
    public int sendMsg(ApplicationMessageAPI sendAppMsg);


    /**
     * Non-blocking method for receiving an application message. RMP creates a new
thread for the receiving process and returns. Once a message is received, RMP calls
the msgReceived -callback method to pass the reference of the message object to the
application. (callback.msgReceived(msg))
     * @param callback object that will hold the received application message data
     * @return -1 if error occurs and 0: for successful sending
     */
    public int receiveMsg(ApplicationMessageAPI receiveAppMsg);


}
```

```
package rmp;

import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;

import javax.swing.JOptionPane;

import rmpAPI.ApplicationMessageAPI;
import rmpAPI.RMPClientAPI;
import rmpAPI.RMPServerAPI;

/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 *
 *This is the RMP server application demonstrations. The server can receive client
messages and set some actions such as re-configurations of
 *RMS IP-address information.
 *
 */
public class ServerAppDemo
{

    // TODO Auto-generated method stub
    //initiate new RMP Client object

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        //client constants
        final int PMTU = 1278;
        final int localPort = 9046;
        final int TX_delay = 3000;
        final int reTXT = 0;
        final int udpBuffer = 9092;
        final int reTransmissionCount = 3;
        final int reassemblingTimeout = 30000;

        //Tag of the application message field
        final int T_REGISTRATION_REQUEST = 1;
        final int T_REGISTRATION_RESPONSE = 2;
        final int T_CONFIGURATION_REQUEST = 3;
        final int T_AUHTENTICATON_SERVER = 4;
        final int T_AUHTENTICATION_PERIOD = 5;
        final int T_RMS_ADDRESS = 6;
        final int T_PMTU = 7;
        final int T_FRAGMENT_ACKNOWLEDGEMENT = 8;
```

```
final int T_DATE_TIME = 9;
final int T_STATISTICS_REPORT_TIME = 10;
final int T_FAULT_FILTER = 11;
final int T_STATUS_LOG_FILTER = 12;
final int T_RESEND_INTERVAL = 13;
final int T_CONFIGURATION_RESPONSE = 14;
final int T_FAULT_CODE = 15;
final int T_TIME_STAMP = 16;
final int T_STATUS = 17;
final int T_STATUS_BEFORE = 18;
final int T_STATUS_AFTER = 19;
final int T_COUNTER = 20;
final int T_MEASURE = 21;
final int T_STATISTICS_RESPONSE = 22;
final int T_ACTION_REQUEST = 23;
final int T_ACTION = 24;
final int T_BOM = 25;
final int T_BOM_RESPONSE = 26;

//Application request message types
final int REGISTRATION_REQUEST = 6;
final int CONFIGURATION_REQUEST = 8;
final int FAULT_OCCURRED_REQUEST = 12;
final int FAULT_UPDATE_REQUEST = 14;
final int STATISTICS_REPORT_REQUEST = 16;
final int ACTION_REQUEST = 18;
final int BOM_REQUEST = 20;
//Application request message types
final int REGISTRATION_RESPONSE = 7;
final int CONFIGURATION = 10;
final int CONFIGURATION_RESPONSE = 11;
final int FAULT_OCCURRED_RESPONSE = 13;
final int FAULT_UPDATE_RESPONSE = 15;
final int STATISTICS_REPORT_RESPONSE = 17;
final int ACTION_RESPONSE = 19;
final int BOM_RESPONSE = 21;
//Server states
final int DISABLED = 1;
final int AUTHENTICATING = 2;
final int REGISTERING = 3;
final int IDLE = 4;
final int SENDING = 5;
final int RECEIVING = 6;
final int CONFIGURING = 7;
final int GO_TO_IDLE = 8;
final int RECEIVED = 9;
final int RECEIVE = 10;
final int ACTION = 11;
final int SEND = 12;

boolean AF;
```

```
        RMPServerAPI rmpSrv = new RMPServer();
        ServerAppGUI gui = new ServerAppGUI();
        ApplicationMessageAPI sending = new ApplicationMessage();
        ApplicationMessageAPI receiving = new ApplicationMessage();
;

        //wait for configuration
    boolean once = true;
        while(gui.getState() == DISABLED)
        {
            if(once)
            {
            gui.cipherText1.append("Server waiting for configuration");
            once = false;
            }
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        gui.cipherText1.append("\nServer configuring");
        //Configure the server
        rmpSrv.setParameters(gui.getPMTU(), gui.getLOCAL_PORT(), gui.getTX_delay(),
gui.getReTX_T(), gui.getUdpBufferLength(), gui.getReTransmisssionRetryCount(), gui.
getReassemblingTimeout());
        //set action flag to false
        AF = false;
        //Start receiving

        while(true)
        {
            gui.cipherText1.append("\nStart of the main loop");
            gui.cipherText1.append("\nAction Status: " + gui.isAF());
            gui.cipherText1.append("\nActions for the client: " + gui.getDeviceID());
            receiving = new ApplicationMessage();
            //receive from RMP
            if(0>rmpSrv.receiveMsg(receiving))
            {
                gui.cipherText1.append("\nserver receive error");
                continue;
            }
            //if received message is not valid, ignore it
            if(receiving.getDeviceID() == 0 || receiving.getMsgType() == 0)
                continue;
            //check if there are actions for the client
            if(gui.isAF())
            {
                gui.cipherText1.append("\nActions exist");
                //Check if there are actions for this client
```

```
                if(receiving.getDeviceID() == gui.getDeviceID())
                {
                    gui.cipherText1.append("\nActions for the client " + gui.
getDeviceID() + " exists");
                    AF = true;
                }
                else
                {
                    AF = false;
                }
            }

            gui.cipherText1.append("\nserver received packet from RMP");
            gui.cipherText1.append("\nMsg type received at the server: " + receiving.
getMsgType());
            printReceivedmsg(gui, receiving);

            if(receiving.getMsgType() == FAULT_OCCURRED_REQUEST)
            {
                sending.initMsg(receiving.getDeviceCategory(), gui.isRMPMode(), AF,
FAULT_OCCURRED_RESPONSE, receiving.getDeviceID(), receiving.getClientIP(), receiving.
getClientPort());
                sending.setField(T_FAULT_CODE, receiving.getField(T_FAULT_CODE));
                sending.setField(T_TIME_STAMP, receiving.getField(T_TIME_STAMP));

            }
            else if(receiving.getMsgType() == STATISTICS_REPORT_REQUEST)
            {

                String message = "SUCCESS";
                sending.initMsg(receiving.getDeviceCategory(), gui.isRMPMode(), AF,
STATISTICS_REPORT_RESPONSE, receiving.getDeviceID(), receiving.getClientIP(),
receiving.getClientPort());
                sending.setField(T_STATISTICS_RESPONSE, message.getBytes());

            }
            else if(receiving.getMsgType() == FAULT_UPDATE_REQUEST)
            {

                sending.initMsg(receiving.getDeviceCategory(), gui.isRMPMode(), AF,
FAULT_UPDATE_RESPONSE, receiving.getDeviceID(), receiving.getClientIP(), receiving.
getClientPort());
                sending.setField(T_FAULT_CODE, receiving.getField(T_FAULT_CODE));
                sending.setField(T_TIME_STAMP, receiving.getField(T_TIME_STAMP));
            }
            else if(receiving.getMsgType() == REGISTRATION_REQUEST)
            {

                String message = "SUCCESS";
                sending.initMsg(receiving.getDeviceCategory(), gui.isRMPMode(), AF,
REGISTRATION_RESPONSE, receiving.getDeviceID(), receiving.getClientIP(), receiving.
getClientPort());
```

```
                sending.setField(T_REGISTRATION_RESPONSE, message.getBytes());
            }
            else if(receiving.getMsgType() == CONFIGURATION_REQUEST)
            {

                //if action flag is set, send new RMS-IP for redirections
                if(AF)
                {
                    //init an configuration message
                    sending.initMsg(receiving.getDeviceCategory(), gui.isRMPMode(),
AF, CONFIGURATION, receiving.getDeviceID(), receiving.getClientIP(), receiving.
getClientPort());
                    //set the configuration field
                    sending.setField(T_RMS_ADDRESS, gui.getIp());
                }
                //if some other than action initiated configuration request arrives,
send just something in testing purposes
                else
                {
                    String message = "It is today";
                    sending.initMsg(receiving.getDeviceCategory(), gui.isRMPMode(),
AF, CONFIGURATION, receiving.getDeviceID(), receiving.getClientIP(), receiving.
getClientPort());
                    sending.setField(T_DATE_TIME, message.getBytes());
                }
            }
            else if(receiving.getMsgType() == BOM_REQUEST)
            {
                gui.setAF(false);
                AF = false;
                String message = "SUCCESS";
                sending.initMsg(receiving.getDeviceCategory(), gui.isRMPMode(), AF,
BOM_RESPONSE, receiving.getDeviceID(), receiving.getClientIP(), receiving.
getClientPort());
                sending.setField(T_BOM_RESPONSE, message.getBytes());
            }
            else if(receiving.getMsgType() == ACTION_REQUEST)
            {

                sending.initMsg(receiving.getDeviceCategory(), gui.isRMPMode(), AF,
ACTION_RESPONSE, receiving.getDeviceID(), receiving.getClientIP(), receiving.
getClientPort());
                sending.setField(T_ACTION, gui.getMsg().getBytes());
            }
            gui.cipherText1.append("\nServer: about to send");
            //do nothing if the configuration response was received
            if((receiving.getMsgType()&1) == 1)
            {
                gui.cipherText1.append("\nServer: no response required");
                //if configuration response was received set the actions off
                if(receiving.getMsgType() == CONFIGURATION_RESPONSE)
                {
```

```
                    gui.setAF(false);
                    AF = false;
                }
                continue;

            }


            gui.cipherText1.append("\nServer: sending message with AF: " + sending.
getAF());
            //send response
            if(0>rmpSrv.sendMsg(sending))
                gui.cipherText1.append("\nServer: sending error");
            sending = new ApplicationMessage();



            }//end of main loop

    }//end of main

    private static void printReceivedmsg(ServerAppGUI gui, ApplicationMessageAPI
receiving)
    {
        //init tag switch
        ApplicationMessageTags getName;
        gui.cipherText2.append("\nReceived a message type of " +receiving.getMsgType
() + " from the client ID: " + receiving.getDeviceID());
        gui.cipherText2.append("\nThe client is in address: " +receiving.getClientIP
() + " And uses the UDP port of: " + receiving.getClientPort());
        gui.cipherText2.append("\nMessage length is: " +receiving.getAppMsg().length
);
        int nextTag;
        //Start parsing the tags
        while((nextTag = receiving.getNextTag()) != -1)
        {

            try
            {
            getName = new ApplicationMessageTags(nextTag);
            if(nextTag == 16)
            {
                InputOutput in = new InputOutput(receiving.getField(nextTag));
                try
                {

                    gui.cipherText2.append("\nTag: " + getName.getTagName() + "\
nValue: " + in.din.readLong());
                }
                catch (IOException e)
                {
                    // TODO Auto-generated catch block
```

```
                    e.printStackTrace();
                }
                in.closeInputStream();
                continue;
            }

            gui.cipherText2.append("\nTag: " + getName.getTagName() + "\nValue: " +
new String(receiving.getField(nextTag)));
            }
            catch(NullPointerException e)
            {
                break;
            }
        }
        gui.cipherText2.append("\n***************\n");

    }

}
```

```
package rmp;


import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.InetAddress;
/**
 * @author Antti Siiril&#228; <br>0602137<br>Turku University of Applied Sciences<br>
antti.siirila&#64;mbnet.fi<br>+358 45 1396013
 * This is the user interface of the RMP Server Demo
 *
 * GUI shows the received messages and allows the user to set some actions for the
clients.
 *
 */

public class ServerAppGUI extends JFrame {


       //RMP parameters
       private int deviceCategory;
       private int deviceID;
       private int PMTU;
       private InetAddress RMS_IP;
       private InetAddress AS_IP;
       private byte[] ip;
       private int RMS_PORT;
       private int AS_PORT;
       private int LOCAL_PORT;
       private int TX_delay;
       private int reTX_T;
       private int udpBufferLength;
       private int reTransmisssionRetryCount;
       private int reassemblingTimeout;

       private int State;
       private int msgType;
       private String msg;
       private boolean RMPMode;
       private boolean AF;


    private static final int DISABLED = 1;
    private static final int AUTHENTICATING = 2;
    private static final int REGISTERING = 3;
    private static final int IDLE = 4;
    private static final int SENDING = 5;
    private static final int RECEIVING = 6;
    private static final int CONFIGURING = 7;
    private static final int GO_TO_IDLE = 8;
```

```
private static final int RECEIVED = 9;
private static final int RECEIVE = 10;
private static final int ACTION = 11;
private static final int SEND = 12;


//Application request message types
private static final int REGISTRATION_REQUEST = 6;
private static final int CONFIGURATION_REQUEST = 8;
private static final int FAULT_OCCURRED_REQUEST = 12;
private static final int FAULT_UPDATE_REQUEST = 14;
private static final int STATISTICS_REPORT_REQUEST = 16;
private static final int ACTION_REQUEST = 18;
private static final int BOM_REQUEST = 20;
//Application response message types
private static final int REGISTRATION_RESPONSE = 7;
private static final int CONFIGURATION = 10;
private static final int CONFIGURATION_RESPONSE = 11;
private static final int FAULT_OCCURRED_RESPONSE = 13;
private static final int FAULT_UPDATE_RESPONSE = 15;
private static final int STATISTICS_REPORT_RESPONSE = 17;
private static final int ACTION_RESPONSE = 19;
private static final int BOM_RESPONSE = 21;



    public JTextArea cipherText1;
    public JTextArea cipherText2;
    private JPanel buttonPanel;    // To hold the buttons
    private JPanel TextArea;    // To hold the buttons
   private JButton[] buttonPanelB;    // To calculate the cost
   private JButton[] configurationButton;    // To exit the application
   private JButton[] set;    // To exit the application
   private JScrollPane scrollingArea1;
   private JScrollPane scrollingArea2;
   private JTextField[] parameterInput;
   private JTextField actionPanelInput[];
   private JRadioButton[] actionTypes;
   private ButtonGroup actions;
   private JCheckBox RACK;
   private JLabel[] parameters;
   private JPanel configurationPanel;
   private JTabbedPane systems;
   private JPanel actionPanel;
   private EventListener clientListener;

   /**
    * Constructs the client GUI
    */
   public ServerAppGUI()
   {
```

```
            //set the server to disabled state
            setState(DISABLED);
            //set action to false
            setAF(false);
            this.clientListener = new EventListener();//Initiate the EventListener
object
          // Display a title.
           setTitle("RMP Server Demo");

           // Specify an action for the close button.
           setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

           // Create a BorderLayout manager.
           setLayout(new BorderLayout());

           // Create the custom panels.
           buildTextArea();
          // Create the button panel.
           buildButtonPanel();
           buildTabPane();
           // Add the components to the content pane.

           add(TextArea, BorderLayout.CENTER);
           add(buttonPanel, BorderLayout.SOUTH);
           add(systems, BorderLayout.EAST);

           // Pack the contents of the window and display it.
           pack();
           setVisible(true);
        }

        /**
         * build the Tab pane and add the panel into it
         */
        private void buildTabPane()
        {
           systems = new JTabbedPane();
           buildConfigurationPanel();
           buildActionPanel();
           systems.addTab("Configuration",null, configurationPanel,"The parameters for
configuring the RMP client");
           systems.setMnemonicAt(0, KeyEvent.VK_1);

           systems.addTab("Actions", null, actionPanel, "Action Options");
           systems.setMnemonicAt(1, KeyEvent.VK_2);


     }

        /**
         * Build the server configuration panel with required input fields
         */
```

```
        private void buildConfigurationPanel()
        {
            configurationPanel = new JPanel(new GridBagLayout());

            GridBagConstraints c = new GridBagConstraints();

            c.gridx = 0;
            c.gridy = 0;
            c.gridwidth = 16;
            c.gridheight = 24;

            //initiate the configure button
            configurationButton = new JButton[1];
            configurationButton[0] = new JButton("Configure");
            //initiate the input fields for the parameters
            parameterInput = new JTextField[7];
            //initiate the parameter labels
            parameters = new JLabel[7];
            //initiate the labels
            String[] labelText = {"Path MTU","Transmission Delay","Retransmission
Delay", "Retransmission Count","Size of the Receiving UDP Buffer","Reassembling
Timeout", "Local Port"};
            String[] defaultConfiguration = {"1278","2000","10000", "3","9092","30000"
, "9046"};
            c.gridy = 0;
            c.gridwidth = 16;
            c.gridheight = 24;

            for(int i = 0; i < parameterInput.length; i++)
            {

                c.gridx = 0;
                //initiate the JLabel with the corresponding string
                parameters[i] = new JLabel(labelText[i]);
                //initiate the input field for IP
                parameterInput[i] = new JTextField(defaultConfiguration[i], 16);

                //add the label
                configurationPanel.add(parameters[i], c);
                //increment the x-constrain to next column
                c.gridx = 16;
                //add the input.
                configurationPanel.add(parameterInput[i], c);
                c.gridy += 24;
            }

            c.gridwidth = 32;
            c.gridx = 0;
            //add action listener for the button
            configurationButton[0].addActionListener(this.clientListener);
            // add configure button
            configurationPanel.add(configurationButton[0], c);
```

```
        }

        /**
         * build the Server Action Panel
         */
        private void buildActionPanel()
        {
            //initiate the sending panel with GridBackLayout
            this.actionPanel = new JPanel(new GridBagLayout());
            //initiate the constrains for the GridBackLayout
            GridBagConstraints c = new GridBagConstraints();
            //Set the constrains
            c.gridx = 0;
            c.gridy = 0;
            c.gridwidth = 12;
            c.gridheight = 24;

            //Initiate the input fields
            this.actionPanelInput = new JTextField[5];
            //initiate a panel array for inner panels
            JPanel innerPanels[] = new JPanel[2];
            innerPanels[0] = new JPanel(new GridLayout(4, 1));
            innerPanels[1] = new JPanel(new GridLayout(4, 1));
            //set borders around the panels
            innerPanels[0].setBorder(BorderFactory.createTitledBorder("Client Actions"
));
            innerPanels[1].setBorder(BorderFactory.createTitledBorder("RMP Mode"));

            //initiate the JRadioButton array for the application messages
            this.actionTypes = new JRadioButton[2];
            //initiate the radio buttons
            this.actionTypes[0] = new JRadioButton("New Config", true);
            this.actionTypes[1] = new JRadioButton("Send BOM");

            //initiate the messages ButtonGroup
            this.actions = new ButtonGroup();

            //add the buttons into the ButtonGroup and into the actionPanel
            for(int i = 0; i < this.actionTypes.length; i++)
            {
                this.actions.add(this.actionTypes[i]);
                innerPanels[0].add(this.actionTypes[i]);
            }

            //initiate a checkBox for RMP mode bit
             this.RACK = new JCheckBox("RMP Acknowledged Mode");
            //initiate a panel for the check box
            innerPanels[1].add(this.RACK);

            //add the Rbuttons panel into the sending panel
            this.actionPanel.add(innerPanels[0], c);
```

```
        //set the x-constrain
        c.gridx = 12;
        //add the check box panel into sending panel
        this.actionPanel.add(innerPanels[1], c);

        //set the constrains
        c.gridx = 0;
        c.gridy += 24;
        //add an informative label into the panel
        this.actionPanel.add(new JLabel("New Server IP for the client:"),c);
        c.gridwidth = 3;
        c.gridx = 12;
        //add input fields
        for(int i = 0; i < 4; i++)
        {
            this.actionPanelInput[i] = new JTextField(3);
            this.actionPanel.add(this.actionPanelInput[i], c);
            c.gridx += 3;
        }
        //initiates the send button array
        this.set = new JButton[1];
        //initiate the input field for the application message
        this.actionPanelInput[4] = new JTextField(12);
        //change row, reset x, and return the grid width
        c.gridy += 24;
        c.gridx = 0;
        c.gridwidth = 12;
        //add an informative label into the panel
        this.actionPanel.add(new JLabel("Device ID:"),c);
        //set the x-constrain
        c.gridx = 12;
        //add the message input into the panel
        this.actionPanel.add(this.actionPanelInput[4], c);
        //initiate the send button
        this.set[0] = new JButton("Set Action Flag");
        //add actionlistener for the button
        this.set[0].addActionListener(this.clientListener);
        //set the constrains
        c.gridwidth = 24;
        c.gridx = 0;
        c.gridy += 24;
        //add the button into the panel
        this.actionPanel.add(set[0], c);



    }



    /**
     * Build the text areas
```

```
 */
private void buildTextArea()
{
       TextArea = new JPanel();
       cipherText1 = new JTextArea(30,20);
       cipherText1.setLineWrap(true);
       cipherText1.setWrapStyleWord(true);
       cipherText2 = new JTextArea(30,20);
       cipherText2.setLineWrap(true);
       cipherText2.setWrapStyleWord(true);

       scrollingArea1 = new JScrollPane(cipherText1);
       scrollingArea2 = new JScrollPane(cipherText2);
       TextArea.add(scrollingArea1);
       TextArea.add(scrollingArea2);
}
/**
   The buildButtonPanel method builds the button panel.
*/

private void buildButtonPanel()
{
   // Create a panel for the buttons.
   buttonPanel = new JPanel();

   // Create the buttons.
   buttonPanelB = new JButton[1];
   buttonPanelB[0] = new JButton("Exit");

   // Register the action listeners.
   buttonPanelB[0].addActionListener(this.clientListener);

   // Add the buttons to the button panel.
   buttonPanel.add(buttonPanelB[0]);

}


/**
   Private inner class that handles the event when
   the user clicks the Calculate button.
*/

public class EventListener implements ActionListener
{

    private boolean[] notBeenHere;


    /**
     * Constructs an EventListener object. Sets the client state to disabled
     */
```

```
            public EventListener()
            {
                setState(DISABLED);//
                this.notBeenHere = new boolean[4];
                this.notBeenHere[0] = true;
                this.notBeenHere[1] = true;
                this.notBeenHere[2] = false;
                this.notBeenHere[3] = false;
            }
            /**
             * Action performed method
             */
           public void actionPerformed(ActionEvent e)
           {
               //Configure button pressed
               if(e.getSource() == configurationButton[0])
               {
                   //read the given parameter values
                   try
                   {
                       setPMTU(Integer.valueOf(parameterInput[0].getText()));
                       setTX_delay(Integer.valueOf(parameterInput[1].getText()));
                       setReTX_T(Integer.valueOf(parameterInput[2].getText()));
                       setUdpBufferLength(Integer.valueOf(parameterInput[4].getText())
);
                       setReTransmisssionRetryCount(Integer.valueOf(parameterInput[3].
getText()));
                       setReassemblingTimeout(Integer.valueOf(parameterInput[5].
getText()));
                       setLOCAL_PORT(Integer.valueOf(parameterInput[6].getText()));
                   }
                   catch(NullPointerException e1)
                   {
                       JOptionPane.showConfirmDialog(null, "Some of the parameters
were not correctly assigned");
                   }

                   cipherText1.append("Server configure pressed");
                       //set the client state to configuring
                       setState(RECEIVING);

               }
               //exit button
               else if(e.getSource() == buttonPanelB[0])
               {

                      System.exit(0);
               }
               //action button
               else if(e.getSource() == set[0])
               {
                   //check the message type
```

```
                    if(actionTypes[0].isSelected())
                    {
                        //set the correct message type
                        setMsg("CONFIGURATION_REQUEST");
                        //save the new RMS ip into byte array
                        ip = new byte[4];
                        cipherText1.append("\n");
                        for(int i = 0; i < 4; i++)
                        {
                            ip[i] = (byte)Integer.parseInt(actionPanelInput[i].getText
());
                            cipherText1.append("ip " + i + " equals " + ip[i]);
                        }

                    }
                    else if(actionTypes[1].isSelected())
                    {
                        //set the correct message type
                        setMsg("BOM_REQUEST");


                    }

                    //set the RMPMode if required
                    if(RACK.isSelected())
                    {
                        setRMPMode(true);
                    }
                    else
                    {
                        setRMPMode(false);
                    }
                    //Get the message
                    setDeviceID(Integer.valueOf(actionPanelInput[4].getText()));
                    //set the action flag for the client
                    setAF(true);
                    cipherText1.append("\n***************\n");
                }
            }
        }



            /**
             * Synchronized method to set the client state
             * @param State state value
             */
            public synchronized void setState(int State)
            {
                this.State = State;
            }
```

```
  /**
   * Synchronized method to get the client state
   * @return
   */
  public synchronized int getState()
  {
      return this.State;
  }
/**
 * @return the deviceCategory
 */
public synchronized int getDeviceCategory() {
    return deviceCategory;
}
/**
 * @param deviceCategory the deviceCategory to set
 */
public synchronized void setDeviceCategory(int deviceCategory) {
    this.deviceCategory = deviceCategory;
}
/**
 * @return the deviceID
 */
public synchronized int getDeviceID() {
    return deviceID;
}
/**
 * @param deviceID the deviceID to set
 */
public synchronized void setDeviceID(int deviceID) {
    this.deviceID = deviceID;
}
/**
 * @return the pMTU
 */
public synchronized int getPMTU() {
    return PMTU;
}
/**
 * @param pMTU the pMTU to set
 */
public synchronized void setPMTU(int pMTU) {
    PMTU = pMTU;
}
/**
 * @return the rMS_IP
 */
public synchronized InetAddress getRMS_IP() {
    return RMS_IP;
}
/**
 * @param rMSIP the rMS_IP to set
```

```
     */
    public synchronized void setRMS_IP(InetAddress rMSIP) {
        RMS_IP = rMSIP;
    }
    /**
     * @return the aS_IP
     */
    public synchronized InetAddress getAS_IP() {
        return AS_IP;
    }
    /**
     * @param aSIP the aS_IP to set
     */
    public synchronized void setAS_IP(InetAddress aSIP) {
        AS_IP = aSIP;
    }
    /**
     * @return the rMS_PORT
     */
    public synchronized int getRMS_PORT() {
        return RMS_PORT;
    }
    /**
     * @param rMSPORT the rMS_PORT to set
     */
    public synchronized void setRMS_PORT(int rMSPORT) {
        RMS_PORT = rMSPORT;
    }
    /**
     * @return the aS_PORT
     */
    public synchronized int getAS_PORT() {
        return AS_PORT;
    }
    /**
     * @param aSPORT the aS_PORT to set
     */
    public synchronized void setAS_PORT(int aSPORT) {
        AS_PORT = aSPORT;
    }
    /**
     * @return the lOCAL_PORT
     */
    public synchronized int getLOCAL_PORT() {
        return LOCAL_PORT;
    }
    /**
     * @param lOCALPORT the lOCAL_PORT to set
     */
    public synchronized void setLOCAL_PORT(int lOCALPORT) {
        LOCAL_PORT = lOCALPORT;
    }
```

```
/**
 * @return the tX_delay
 */
public synchronized int getTX_delay() {
    return TX_delay;
}
/**
 * @param tXDelay the tX_delay to set
 */
public synchronized void setTX_delay(int tXDelay) {
    TX_delay = tXDelay;
}
/**
 * @return the reTX_T
 */
public synchronized int getReTX_T() {
    return reTX_T;
}
/**
 * @param reTXT the reTX_T to set
 */
public synchronized void setReTX_T(int reTXT) {
    reTX_T = reTXT;
}
/**
 * @return the udpBufferLength
 */
public synchronized int getUdpBufferLength() {
    return udpBufferLength;
}
/**
 * @param udpBufferLength the udpBufferLength to set
 */
public synchronized void setUdpBufferLength(int udpBufferLength) {
    this.udpBufferLength = udpBufferLength;
}
/**
 * @return the reTransmisssionRetryCount
 */
public synchronized int getReTransmisssionRetryCount() {
    return reTransmisssionRetryCount;
}
/**
 * @param reTransmisssionRetryCount the reTransmisssionRetryCount to set
 */
public synchronized void setReTransmisssionRetryCount(
        int reTransmisssionRetryCount) {
    this.reTransmisssionRetryCount = reTransmisssionRetryCount;
}
/**
 * @return the reassemblingTimeout
 */
```

```
public synchronized int getReassemblingTimeout() {
    return reassemblingTimeout;
}
/**
 * @param reassemblingTimeout the reassemblingTimeout to set
 */
public synchronized void setReassemblingTimeout(int reassemblingTimeout) {
    this.reassemblingTimeout = reassemblingTimeout;
}
/**
 * @return the msgType
 */
public synchronized int getMsgType() {
    return this.msgType;
}
/**
 * @param msgType the msgType to set
 */
public synchronized void setMsgType(int msgType) {
    this.msgType = msgType;
}
/**
 * @return the msg
 */
public synchronized String getMsg() {
    return msg;
}
/**
 * @param msg the msg to set
 */
public synchronized void setMsg(String msg) {
    this.msg = msg;
}
/**
 * @return the rMPMode
 */
public synchronized boolean isRMPMode() {
    return RMPMode;
}
/**
 * @param rMPMode the rMPMode to set
 */
public synchronized void setRMPMode(boolean rMPMode) {
    RMPMode = rMPMode;
}

/**
 * @return the aF
 */
public synchronized boolean isAF() {
    return AF;
}
```

```
        /**
         * @param aF the aF to set
         */
        public synchronized void setAF(boolean aF) {
            AF = aF;
        }

        /**
         * @return the ip
         */
        public synchronized byte[] getIp() {
            return ip;
        }

        /**
         * @param ip the ip to set
         */
        public synchronized void setIp(byte[] ip) {
            this.ip = ip;
        }

}
```