ABSTRACT

This undergraduate dissertation discusses the implementation of three key management and security servers, the Ticket Granting Server, the Record Keeping Server and the SYSLOG Server, in the Packet Cable Network. A detail picture of the Packet Cable project also includes a general overview of how it is able to achieve transfer of voice over the HFC network and communicate with the PSTN network. This document also encompasses the pros and cons of how the authors implemented the Kerberos protocol with PKINIT extension, the Internet Key Exchange protocol implementation, the RADIUS protocol, the SYSLOG server.

The research and subsequent development of the key management and security servers was carried out by Mubashir Hayat, Hammad Amjad, Arslan Javed and Farid Anwar, under the supervision of Dr. Muhammad Riaz and Lt. Col. Muhammad Tufail.

ACKNOWLEDGEMENTS

We would like to thank Dr. Muhammad Riaz and Lt. Col. Muhammad Tufail, our project supervisors, for their continuous support and motivation. We are extremely grateful to Dr. Sohail Naqvi for his unlimited support and guidance which helped us successfully complete the project. We would also like to thank Mr. Ali Khayam and Miss Sania for guiding us in the implementation of the project, especially the encryption and digest algorithms, Mr. Umair for helping us understand networking issues and Mr. Khalid Jameel for extending all the possible technical support.

We would also like to thank our families for supporting us and understanding our commitments throughout the tenure of the project.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. PacketCable Overview

The PacketCable project is aimed at defining interface specifications that can be used to develop interoperable equipment capable of providing packet-based voice, video and other high-speed multimedia services over hybrid fiber coax (HFC) cable systems utilizing the DOCSIS protocol.

PacketCable utilizes a network superstructure that overlays the two-way data-ready broadband cable access network. The PacketCable project is aimed at defining interface specifications that can be used to develop interoperable equipment capable of providing packet-based voice, video and other high-speed multimedia services over hybrid fiber coax (HFC) cable systems utilizing the DOCSIS protocol.

The initial PacketCable offering comprising of packet-based voice communications will soon be extended to encompass a large suite of packet-based capabilities.

1.1.1. PacketCable Architecture Framework

The PacketCable architecture consists of the following three main networks;

- The DOCSIS HFC Access Network;
- The Managed IP Network;
- The PSTN.

The Cable Modem Termination System (CMTS) provides connectivity between the DOCSIS HFC Access Network and the Managed IP Network. Both the Signaling Gateway (SG) and the Media Gateway (MG) provide connectivity between the Managed IP Network and the PSTN.

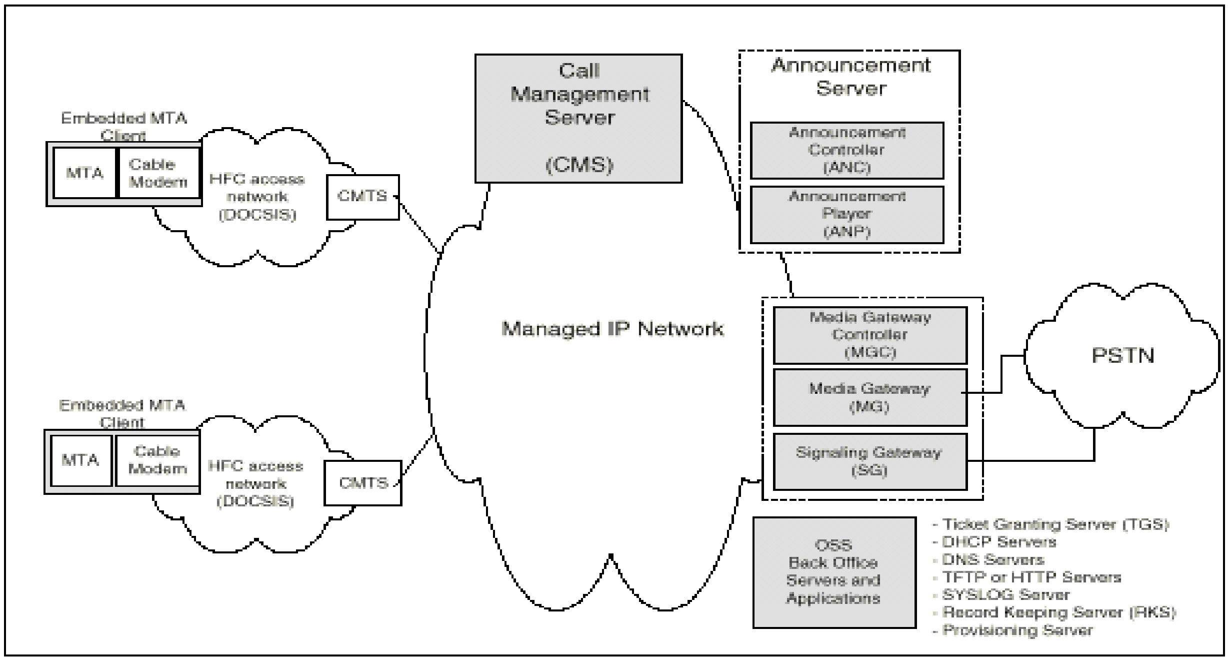

Embedded MTA Client
MTA
Cable Modem
HFC access network (DOCSIS)
CMTS
Embedded MTA Client
MTA
Cable Modem
HFC access network (DOCSIS)
CMTS
Call Management Server (CMS)
Announcement Server
Announcement Controller (ANC)
Announcement Player (ANP)
Managed IP Network
Media Gateway Controller (MGC)
Media Gateway (MG)
Signaling Gateway (SG)
PSTN
OSS Back Office Servers and Applications
- Ticket Granting Server (TGS)
- DHCP Servers
- DNS Servers
- TFTP or HTTP Servers
- SYSLOG Server
- Record Keeping Server (RKS)
- Provisioning Server

Figure 1.1 – PacketCable Architecture Framework

The DOCSIS HFC access network provides high-speed, reliable, and secure transport between the customer premise and the cable head end. This access network may provide all DOCSIS 1.1 capabilities including Quality of Service. The DOCSIS HFC access network includes the following functional components:

- Cable Modem (CM);
- Multi-media Terminal Adapter (MTA);
- Cable Modem Termination System (CMTS).

The Managed IP network serves several functions. First, it provides interconnection between the basic PacketCable functional components responsible for signaling, media, provisioning, and quality of service establishment. In addition, the managed IP network provides long-haul IP connectivity between other Managed IP and DOCSIS HFC networks. The Managed IP network includes the following functional components:

- Call Management Server (CMS);
- Announcement Server (ANS);
- Operational Support System (OSS) back-office servers;
- Signaling Gateway (SG);
- Media Gateway (MG);
- Media Gateway Controller (MGC)

1.1.2. PacketCable Functional Components

Each of the functional components in the packet cable architecture, has a precise role to play in the entire scenario. Figure 1.2 helps illustrate the functionality of each of these components.

The main components with respect to the Packet Cable framework are:

- Multimedia Terminal Adapter (MTA);
- Cable Modem Termination System (CMTS);
- Operation Support System (OSS);
- Call Management Server (CMS);
- Media Servers;
- PSTN Gateways;

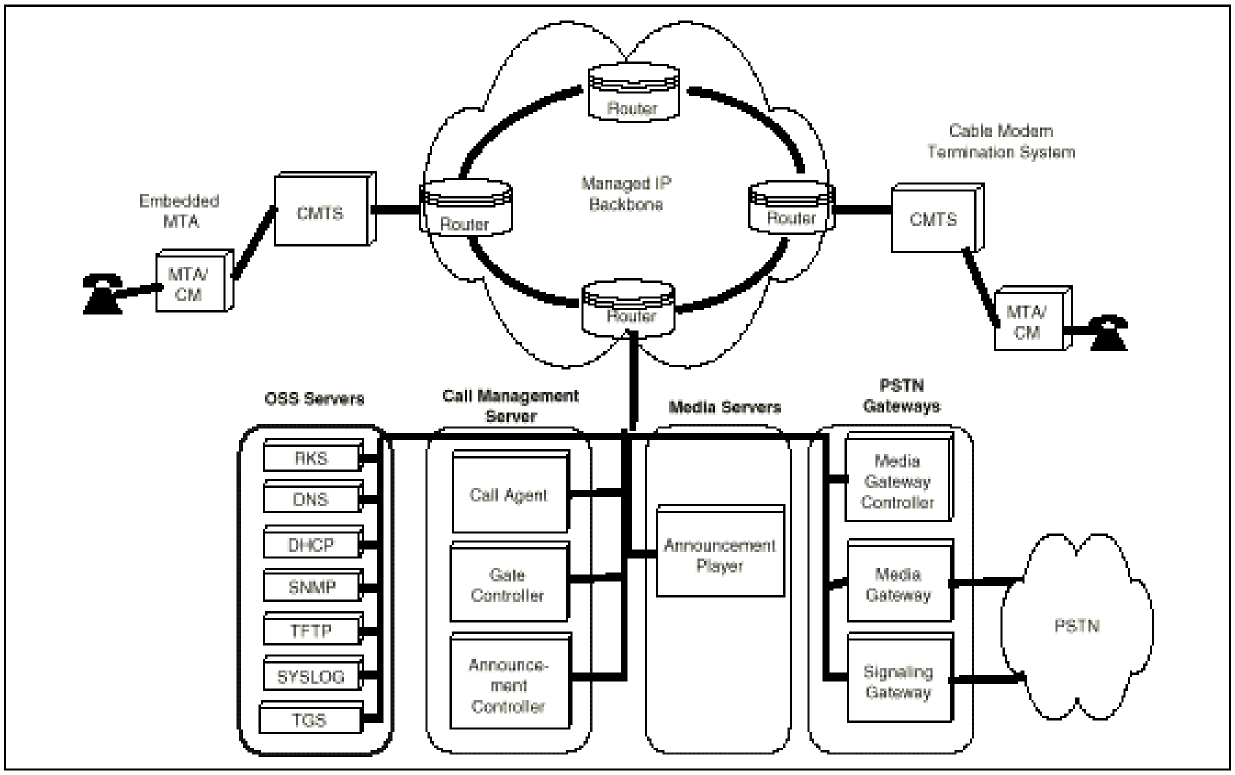


Figure 1.2 – PacketCable Reference Model

1.1.3. PacketCable Security Features

The Packet Cable protocol interfaces are subject to threats that could pose security risks to both the subscriber and service provider. The PacketCable architecture addresses these threats by specifying, for each defined protocol interface, the underlying security mechanisms that provide the protocol interface with the security services it requires for example, authentication, integrity, confidentiality.

For example, the media stream path may traverse a large number of potentially unknown Internet service and backbone service providers' wires.

As a result, the media stream may be vulnerable to malicious eavesdropping, resulting in a loss of communications privacy. Packet Cable core security services include a mechanism for providing end-to-end encryption of RTP media streams, thus substantially reducing the threat to privacy.

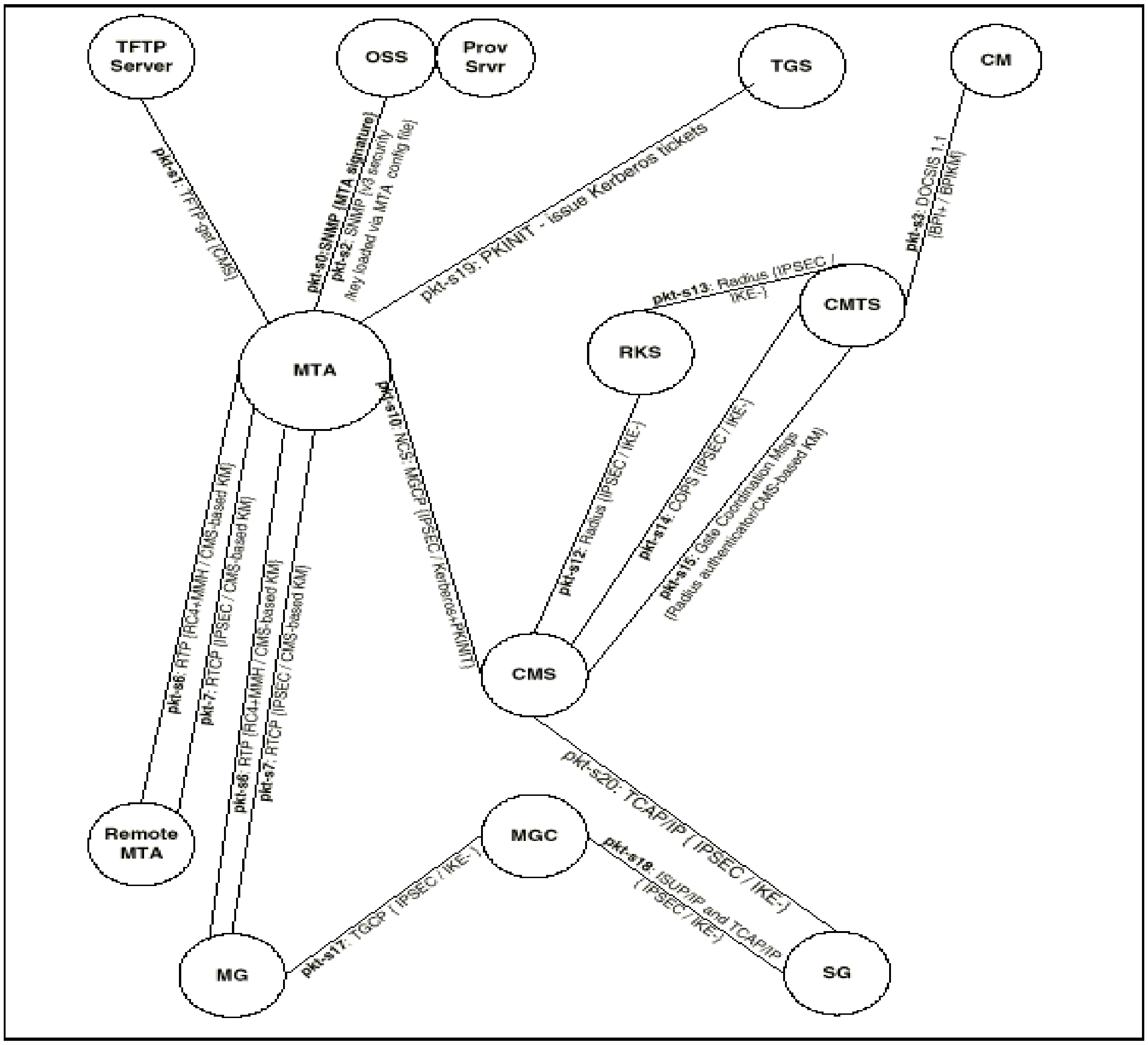
The security services available through Packet Cable's core service layer are authentication, access control, integrity, confidentiality and non-repudiation. A PacketCable protocol interface may employ zero or more of these services to address its particular security requirements.

PacketCable security addresses the security requirements of each constituent protocol interface by:

- Identifying the threat model specific to each constituent protocol interface;
- Identifying the security services (authentication, authorization, confidentiality, integrity, and non-repudiation) required to address the identified threats;
- Specifying the particular security mechanism providing the required security services.

The security mechanisms include both the security protocol, for example IPSec, RTP-layer security, and SNMPv3 security and the supporting key management protocol, for example IKE, PKINIT/Kerberos.

The various security interfaces in the entire scenario are illustrated in the following Figure 1.3.



TFTP Server
OSS
Prov Srvr
TGS
CM
CMTS
RKS
MTA
CMS
Remote MTA
MGC
MG
SG
pkt-s1: TFTP-get [CMS]
pkt-s0:SNMP [MTA signature]
pkt-s2: SNMP [v3 security key loaded via MTA config file]
pkt-s19: PKINIT - issue Kerberos tickets
pkt-s3: DOCSIS 1.1 [BPI+ / BPIKM]
pkt-s13: Radius [IPSEC / IKE-]
pkt-s10: NCS MGCP [IPSEC Kerberos+PKINIT]
pkt-s12: Radius [IPSEC / IKE-]
pkt-s14: COPS [IPSEC / IKE-]
pkt-s15: Gate Coordination Msgs (Radius authenticator/CMS-based KM)
pkt-s6: RTP [RC4+MMH / CMS-based KM]
pkt-s7: RTCP [IPSEC / CMS-based KM]
pkt-s20: TCAP/IP (IPSEC / IKE-)
pkt-s18: ISUP/IP and TCAP/IP (IPSEC / IKE-)
pkt-s17: TGCP (IPSEC / IKE-)

__**Figure 1.3 – PacketCable Security Interfaces**__

These interfaces illustrate the importance of the security features in the entire scenario and thus, the significance of the security and key management protocols implemented to protect these interfaces.

1.2. OSS Back Office Components

The Operation Support System (OSS) back office contains business, service, and network management components supporting the core business processes. The main functional areas for Operation Support System are fault management, performance management, security management, accounting management, and configuration management.

The various servers comprising these back office components are:

- Ticket Granting Server (TGS);
- Record Keeping Server (RKS);
- SYSLOG Server;
- Dynamic Host Configuration Protocol (DHCP) Server;
- Domain Name System (DNS) Server;
- Trivial File Transfer Protocol (TFTP) Server or Hyper Text Transfer Protocol (HTTP) Server;
- Simple Network Management Protocol (SNMP) Server.

The following three servers have been implemented in this project. These mainly deal with key management and security features related to the Packet Cable scenario.

1.2.1. Ticket Granting Server

The Ticket Granting Server is actually a Key Distribution Center (KDC), which is employed for key management on the Multimedia Terminal Adapter and Call

Management Server interface. This is an implementation of the Kerberos protocol with the public key PKINIT extension.

The Ticket Granting Server is responsible for granting Kerberos tickets to a Multimedia Terminal Adapter. A ticket contains information used to set up authentication, privacy, integrity and access control for the call signaling between the Multimedia Terminal Adapter and the Call Management Server.

The Multimedia Terminal Adapter requests tickets from the Ticket Granting Server in case of device provisioning, so that it can then establish a secure session with the Cal Management Server using this ticket. Each of these tickets remain saved with the client device until expiry, after which the client device should request for another ticket by executing PKINIT.

The Ticket Granting Server thus prevents the provisioning of any alien client, and hence, the use of any unauthorized services by any unidentified device on the network.

1.2.2. Record Keeping Server

The Record Keeping Server is a trusted network element component that receives Packet Cable Event Messages from other trusted Packet Cable network elements such as the Call Management Server, Cable Management Termination Server, and Media Gateway Controller. The Record Keeping Server also, at a minimum, is a short-term repository for Packet Cable Event Messages. The Record Keeping Server may assemble the Event Messages into coherent sets or Call Detail Records (CDR), which are then made available to other back office systems such as billing, fraud detection, and other systems.

The RKS acts as a database and stores each event as sent by the Cable Modem Termination System. The Record Keeping Server stores the messages by attaching received time and network element information. The Record Keeping Server has to have sufficient interface and/or processing power to allow additional processing to be done.

The Record Keeping Server is basically an implementation of a number of protocols, in which the key management is achieved via the Internet Key exchange Protocol, and further generation and receipt of event messages is achieved under the protection of the security associations negotiated in Internet Key Exchange.

1.2.3. SYSLOG Server

The SYSLOG Server is responsible for capturing any kind of traps generated by the client device, that is the Multimedia Terminal Adapter, and then maintain a log of these traps. This server is mainly linked with the network management aspects, and helps administrate the network with minimal errors and traps on the client end.

1.3. Future of IP over Cable

Transmitting IP over a cable network has come a long in a relatively short duration of time but it still has a ling way to go. Throughout the world, the cable network is spread over far and wide and is accessible to the mass population. Transferring IP packets over the cable back-bone not only reduces cost but also provides much wider bandwidth and resources to the user so that many diverse services can be provided to the user.

The use of cable back-bone for transferring IP data is still being researched upon and already some service providers have started employing this technology.

CHAPTER 2
Project Specifications

2.1 Statement

The key management and security servers allow the various entities in the Packet Cable framework to communicate securely over the network. In case of the Ticket Granting Server, the client will have to obtain a Kerberos ticket from the server by executing PKINIT to further communicate with the Call Management Server. In case of the Record Keeping Server, the communicating entities need to negotiate security associations using Internet Key Exchange and then further exchange event messages using RADIUS protocol. Finally, whenever the client may generate traps and errors, which will be captured by the SYSLOG server.

The project is designed following proper software engineering methodologies. For this purpose the Unified Modeling Language (UML) has been used as the primary design and modeling language. The project will be developed on the Linux platform, due to its improved networking capabilities and to develop a better understanding of the environment.

The Ticket Granting Server is based on the Kerberos protocol with PKINIT extension, which will enable the client to negotiate with the server for a valid key to further initiate the communication with the Call Management Server and use the available services.

The Record Keeping Server is based on the RADIUS protocol, which achieves the exchange of messages, and is preceded by the Internet

Key Exchange protocol, which enables the exchange of security associations over an open network.

The SYSLOG server is based on the generation of traps and errors at the client end, which are captured and recorded by the server and help in network management issues.

2.2 Development Environments

The following environments were utilized in the implementation of this project:

- K-Developer Linux environment for C/C++
- Visual C++ for testing and debugging individual modules
- Rational Rose for UML development
- Microsoft Project for Gantt Charts
- Qt libraries for GUI development in Linux
- Qt designer
- T-make for makefile generation
- OSS Nokalva ASN.1 Compiler and associated libraries
- MIB Browser
- MIB Builder
- MIB Compiler

The programming language used for implementation purposes is C++.

2.3 Platform Supported

The implementation is consistent on following platforms:

- Red Hat Linux Zoot

- Red Hat Linux Guinness

2.4 System Modeling/Design

Unified Modeling Language (UML) was employed in the development of the project. All the phases of UML designing were followed. The environment used was Rational Rose Enterprise.

<u>CHAPTER 3</u>

ENCRPYTION ALGORITHMS

3.1 Introduction

The PacketCable specifications list a number of encryption algorithms that can be incorporated in the framework. Among these, three DESCBC is mandatory while others like RSA are optional.

3.2 Data Encryption Algorithm

This implementation utilizes 3DESCBC. The Data Encryption Standard Algorithm is designed to encipher and decipher blocks of data consisting of 64 bits under control of a 64-bit key . Deciphering must be accomplished by using the same key as for enciphering, but with the schedule of addressing the key bits altered so that the deciphering process is the reverse of the enciphering process.

3.2.1 Data Encryption Algorithm Description

A block to be enciphered is subjected to an initial permutation ***IP***, then to a complex key-dependent computation and finally to a permutation which is the inverse of the initial permutation ***IP*** **-1** . The key-dependent computation can be simply defined in terms of a function ***f***, called the cipher function, and a function ***KS***, called the key schedule. The cipher function ***f*** can be defined in terms of primitive functions which are called the selection functions ***Si*** and the permutation function ***P***.

The following diagram (on the next page) illustrates the enciphering procedure, where **L** and **R** are two blocks of bits and **LR** denotes a block with bits of **L** followed by bits of **R**.

The permutation **IP** takes a 64-bit input and produces a 64-bit output depending on the permutation function. This 64-bit output is then undergoes a complicated key dependent computation. Thus, the 64-bit output is first divided into two 32-bit blocks, **L(i)** and **R(i)**. Then the following operations are performed 16 times, that is from i equals to zero to i equals to 15

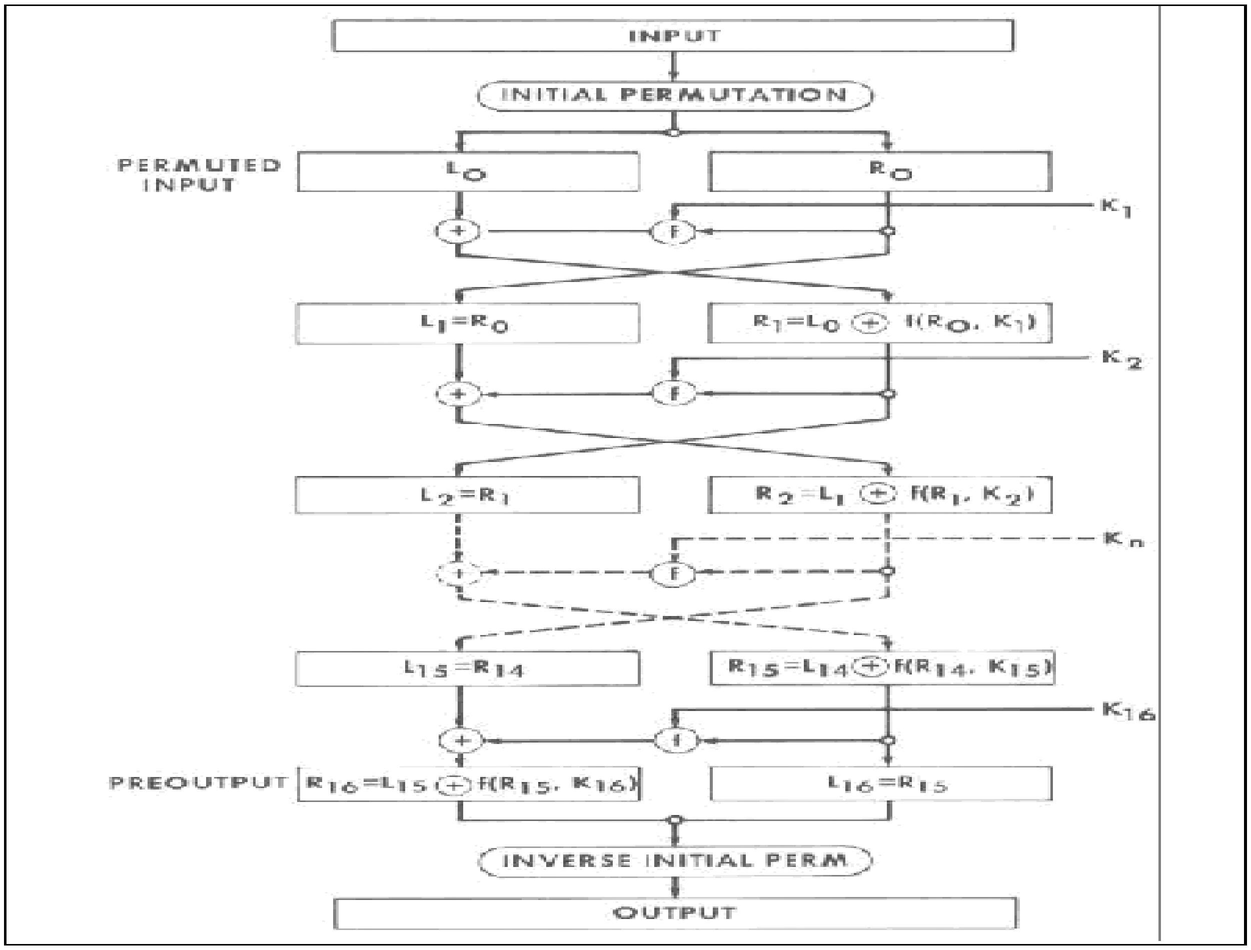


Figure 3.1 – Data Encryption Standard

***L*(i+1) = *R*(i)**
R(i+1)* = *L*(i+1) *xor f(R,K)

Here, the key, **K**, is a 48-bit block chosen from a 64-bit block depending upon the number of iteration. This key is generated using a Key Schedule, **KS**, which is illustrated in the following diagram.

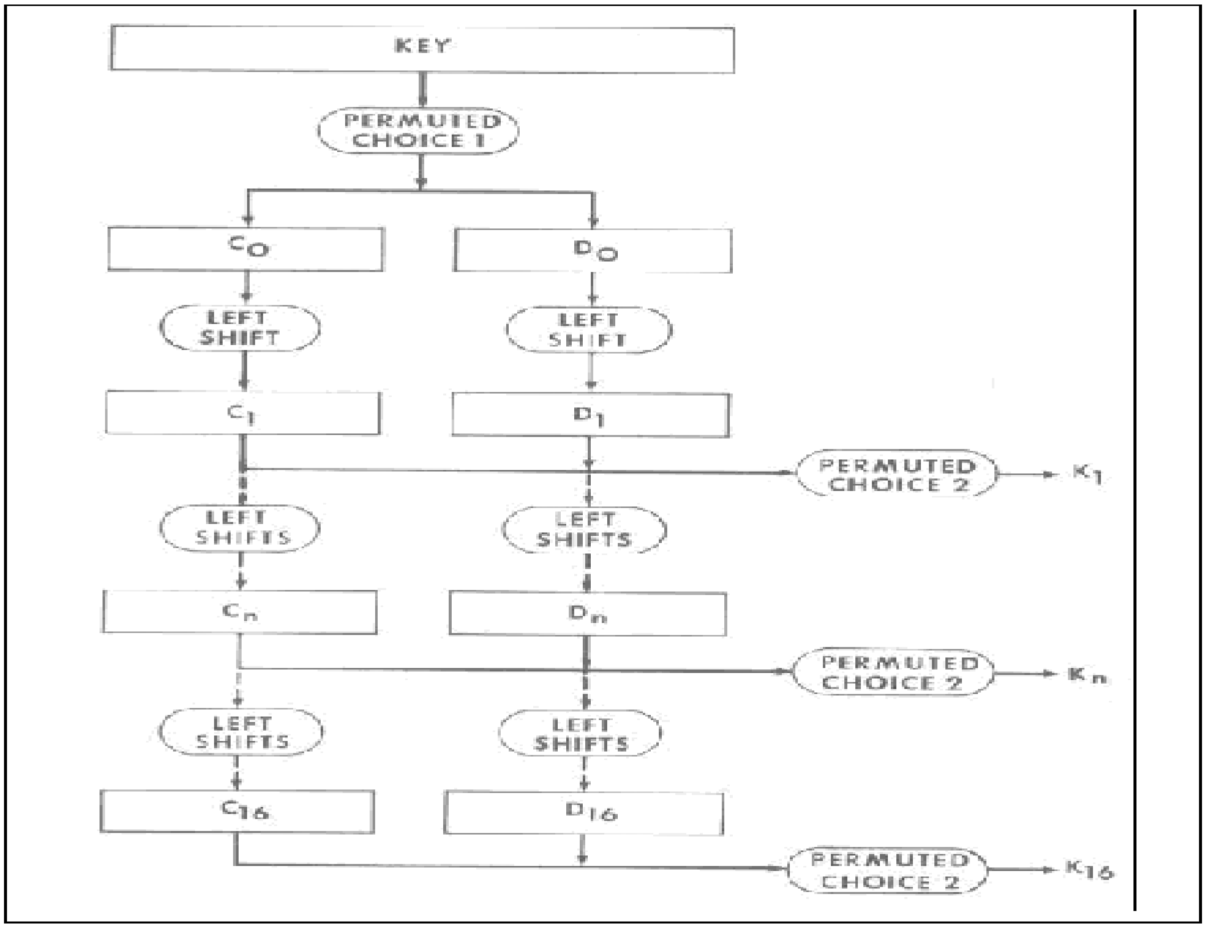


Figure 3.2 – DES Key Generation

Thus, here the key is first fed to the permutation function **PC1** and then predetermined number of shifts are applied to two blocks of 28-bits each, as 8-bits from the 64-bit key are utilized for error detection in key generation, distribution and storage.

The cipher function, **f**, is applied on the **R** bits and the **KS** output **K**. This is a combination of the permutation functions **E** and **P** and a set of selection functions **S(i)**. This is illustrated in the following diagram.

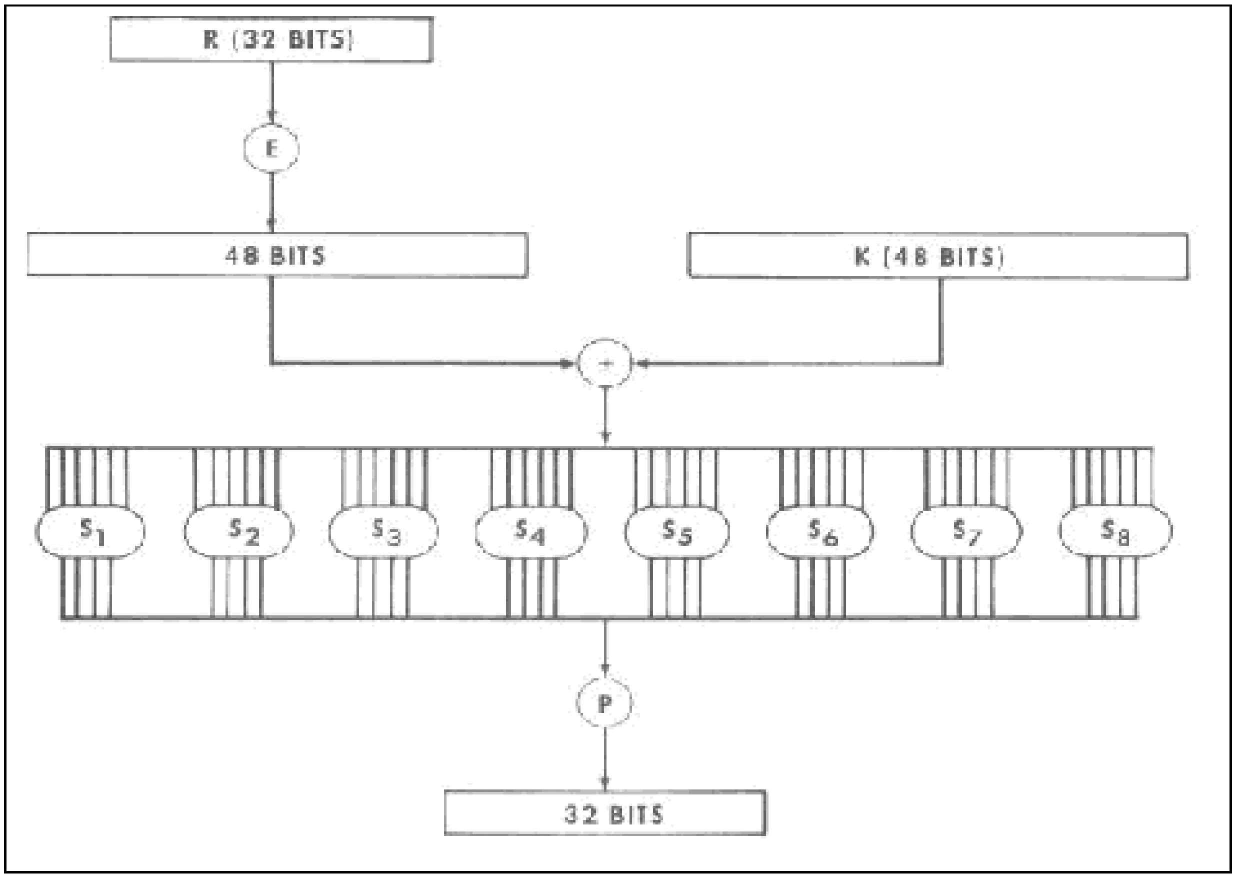


Figure 3.3 – DES Selection and Permutation Function

Thus, here the 32-bits are fed to the permutation function **E** which yields a 48-bit output. This is then XORed with the 48-bit **K** and then fed to the selection functions **S(i)** which produce a 32-bit output. These 32-bits are then fed to the permutation function **P** thus obtaining a 32-bit output.

Finally, when the 16 iterations are complete the final 32-bit blocks are swapped and fed to the permutation function **IP-1** which yields the final enciphered output.

To decipher the output the reverse procedure is employed and the input message is obtained if the keys are properly handled.

3.2.2 Triple Data Encryption Algorithm (TDEA)

Triple Data Encryption Algorithm (TDEA) is achieved by first enciphering an input block with the given key. Then this enciphered block is deciphered using a key different from the first one used for enciphering. Finally, this deciphered block is again enciphered using the same key, which was used for the first enciphering. This is as shown in the following diagram.

$$\begin{tikzcd} I \arrow[r] & \boxed{\text{DES } E_{K1}} \arrow[r] & \boxed{\text{DES } D_{K2}} \arrow[r] & \boxed{\text{DES } E_{K3}} \arrow[r] & O \end{tikzcd}$$

Figure 3.4 – Three DES

Here the keys K1 and K3 are normally kept the same to avoid storage and generation overheads.

3.2.3 Cipher Block Chaining (CBC)

Cipher Block Chaining (CBC) is employed for further security by XORing the output of the TDEA for the first 64-bit input with an initialization vector, **IV**, and then consequently XORing the output of the TDEA for the next 64-bit blocks with the output for the previous block.

CHAPTER 4
DIGEST ALGORITHMS

4.1 Introduction

The PacketCable specifications list a number of digest algorithms that can be incorporated in the framework. Among these, MD5 is mandatory while others like SHA1 are optional.

4.2 Message Digest MD5

The MD5 message-digest algorithm is an extension of its predecessor, the MD4 message digest algorithm, and incorporates enhanced security features. The MD5 message-digest algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest.

The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

The MD5 algorithm is designed to be quite fast on 32-bit machines. In addition, the MD5 algorithm does not require any large substitution tables and can be coded quite compactly.

4.2.1 MD5 ALGORITHM DESCRIPTION

Supposing that the input to the MD5 message digest algorithm is a b-bit message, where b is an arbitrary nonnegative integer. Thus, b may be zero, it need not be a multiple of eight, and it may be arbitrarily large. The bits of the message can thus be written down as follows:

m_0 m_1 ... m_{b-1}

Now, the MD5 message digest algorithm comprises of five main steps which perform various computations on the given input data and finally produce as output a 128-bit message digest.

However, before discussing the five steps of the digest algorithm some terminologies and notations used in the following discussion are as follows;

- "word" is a 32-bit quantity and a "byte" is an 8-bit quantity. Moreover, a sequence of bits can be interpreted in a natural manner as a sequence of bytes, where each consecutive group of eight bits is interpreted as a byte with the high-order (most significant) bit of each byte listed first. Similarly, a sequence of bytes can be interpreted as a sequence of 32-bit words, where each consecutive group of four bytes is interpreted as a word with the low-order (least significant) byte given first.
- Let x_i denote "x sub i". If the subscript is an expression, surround it in braces, as in x_{i+1}. Similarly, x^i denotes x to the i-th power.

- Let the symbol "+" denote addition of words (i.e., modulo-2^32 addition).
- Let X <<< s denote the 32-bit value obtained by circularly shifting (rotating) X left by s bit positions.
- Let not(X) denote the bit-wise complement of X, and let X v Y denote the bit-wise OR of X and Y. Let X xor Y denote the bit-wise XOR of X and Y, and let XY denote the bit-wise AND of X and Y.

The five steps for computing the MD5 message digest of a given input are as follows:

4.2.2 Append Padding Bits

The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512. Padding is performed as follows:

a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.

4.2.3 Append Length

A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^64, then only the low-order 64 bits of b are used. (These bits are appended as two 32-

bit words and appended low-order word first in accordance with the previous conventions.) At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits.

Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let M[0 ... N-1] denote the words of the resulting message, where N is a multiple of 16.

4.2.4 Initialize MD Buffer

A four-word buffer (A,B,C,D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal, low-order bytes first:

```
word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10
```

4.2.5 Process Message in 16-Word Blocks

We first define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

```
F(X,Y,Z) = XY v not(X) Z
G(X,Y,Z) = XZ v Y not(Z)
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X v not(Z))
```

This step uses a 64-element table T[1 ... 64] constructed from the sine function. Let T[i] denote the i-th element of the table, which is equal to the integer part of 4294967296 times abs(sin(i)), where i is in radians.

Now, perform the following computations N/16 times and sequentially process each 16-word block of M.

First of all, copy the 16-word block to be processed into X and save the initial values of the registers A, B, C and D into temporary registers AA, BB, CC and DD.

Next, perform the computations specified by the following four rounds.

Round 1: Here [abcd k s i] denotes the operation

$$a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s)$$

Do the following operations;

[ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
[ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]
[ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
[ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]

Round 2: Here [abcd k s i] denotes the operation

$$a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s)$$

Do the following operations;

[ABCD 1 5 17] [DABC 6 18] [CDAB 11 14 19] [BCDA 0 20 20]
[ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]
[ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
[ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]

Round 3: Here [abcd k s t] denotes the operation

a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s)

Do the following 16 operations;

[ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
[ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
[ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
[ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]

Round 4: Here [abcd k s t] denotes the operation

a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s)

Do the following 16 operations.

```
[ABCD  0  6 49]  [DABC  7 10 50]  [CDAB 14 15 51]  [BCDA  5 21 52]
[ABCD 12  6 53]  [DABC  3 10 54]  [CDAB 10 15 55]  [BCDA  1 21 56]
[ABCD  8  6 57]  [DABC 15 10 58]  [CDAB  6 15 59]  [BCDA 13 21 60]
[ABCD  4  6 61]  [DABC 11 10 62]  [CDAB  2 15 63]  [BCDA  9 21 64]
```

Finally, perform the following additions. (That is increment each of the four registers by the value it had before this block was started.)

A = A + AA

B = B + BB

C = C + CC

D = D + DD

4.2.6 Output

The message digest produced as output is A, B, C and D, that is, starting with the low-order byte of A, and end with the high-order byte of D.

CHAPTER 5
ABSTRACT SYNTAX NOTATION

5.1 Introduction to ASN1

ASN.1 is an abstract notation for data values and structures. It is very much like a type declaration in C or C++. In other words, it can be described as a formal notation used for describing data transmitted by telecommunications protocols, regardless of language implementation and physical representation of these data, whatever the application, whether complex or very simple.

The notation provides a certain number of pre-defined basic types such as:

- Integers (INTEGER)
- Booleans (BOOLEAN)
- Character strings (IA5String, UniversalString etc)
- Bit strings (BIT STRING), etc.

Moreover, it also makes it possible to define constructed types such as:

- Structures (SEQUENCE)
- Lists (SEQUENCE OF)
- Choice between types (CHOICE), etc.

Sub-typing constraints can be also applied on any ASN.1 type in order to restrict its set of values. Moreover, ASN.1 only covers the structural aspects of information and there are no operators to handle the values once these are defined or to make calculations with. Therefore it is not a programming language.

ASN.1 is also associated with several standardized encoding rules such as the BER (*Basic Encoding Rules*), the DER (*Distinguished Encoding Rules*) or more recently the PER (*Packed Encoding Rules*), which prove useful for applications that undergo restrictions in terms of bandwidth. These encoding rules describe how the values defined in ASN.1 can be translated into the bytes 'over the wire' and reverse.

ASN.1 defines four main kind of types, where types can be defined as a set of values. These are:

- Simple types (which are ‘atomic’ and have no components)
- Structured types (which have components)
- Tagged types (which are derived from other types)
- Other types (which include the CHOICE and ANY types)

Each of these types can also be assigned names in ASN.1 by employing the assignment operator (:=) and these names can further be employed for other types and values.

Every ASN.1 type, except CHOICE and ANY, has tags, which consist of a class and a nonnegative tag number. There are four main classes of tags in ASN.1 and these are as follows:

- Universal
- Application
- Private
- Context Specific.

Thus, it is the tag number which effects the ASN.1 type and not the name of the type. There are a certain set of rules and some standardized values associated with these tag numbers.

5.2 Encoding Rules

Encoding rules are basically a set of pre-defined rules employed for representing the ASN.1 values as an octet string. These mainly include the following three rules:

- Basic Encoding Rules (BER)
- Distinguished Encoding Rules (DER)
- Packed Encoding Rules (PER)

Among these the Basic Encoding Rules was the original set of rules employed for transforming an ASN.1 value into a sequence of bits and bytes. However, in case of Basic Encoding Rules there were a number of representations possible for a certain value in ASN.1, therefore, the Distinguished Encoding Rules was introduced. Distinguished Encoding Rules is a subset of the Basic Encoding Rules and defines only one possible representation for a certain ASN.1 value.

Furthermore, the Packed Encoding Rules was introduced which reduced the size of the representation in case of the Basic Encoding Rules by approximately four to five times.

5.3 ASN.1 Compiler

There a number of compilers capable of interpreting ASN.1 structures into some specified language constructs, like for C, C++ etc. These take as input an input file consisting of ASN.1 structures and then generate corresponding files to be used with the application, for example in case of C or C++ a ‘cpp’ and a ‘header’ file is generated.

Moreover, encoders and decoders accompany the compiler for encoding and decoding the application data. Thus, the encoder takes as input the values assigned to the C structures and convert them to a sequence of bits employing any one of the encoding rules and these can then be transmitted over a communication line. Similarly, the decoder takes as input a string of bits and interprets them into values and assigns them to the specified C structure.

There are mainly two types of encoders and decoders and these are:

- Space Optimized Encoders And Decoders
- Time Optimized Encoders And Decoders

CHAPTER 6

TICKET GRANTING SERVER

6.1 Introduction

Ticket Granting Server (TGS) is an essential component of the OSS (Operation Support System) back office components. The ticket-granting server is used to authenticate the MTA (Multimedia Terminal Adapter / Client) and issue a ticket so that the client can establish a session with the CMS (Call Management Server). The TGS maintains a database containing the client name, server name, secret key of CMS, key version number and maximum lifetime for tickets. Kerberos protocol is used for the implementation of Ticket Granting Server. An extension PKINIT of the Kerberos protocol has been used to improve security.

6.2 Kerberos

Kerberos provides means of verifying the identities of principals,(e.g., a workstation user or a network server) on an open (unprotected) network. This is accomplished without relying on authentication by the host operating system, without basing trust on host addresses, without requiring physical security of all the hosts on the network, and under the assumption that packets traveling along the network can be read, modified, and inserted at will. Kerberos performs authentication as a trusted third-party authentication service by using conventional cryptography, i.e., shared secret key. Thus, in public key cryptosystems, one has a public and a private key.

6.2.1 Authentication Process

The authentication process proceeds as follows:
A client sends a request to the authentication server (AS) requesting "credentials" for a given server. The AS responds with these credentials, encrypted in the client's key. The credentials consist of:

- a "ticket" for the server
- a temporary encryption key (often called a "session key").

The client transmits the ticket (which contains the client's identity and a copy of the session key, all encrypted in the server's key) to the server. The session key (now shared by the client and server) is used to authenticate the client, and may optionally be used to authenticate the server. It may also be used to encrypt further communication between the two parties or to exchange a separate sub-session key to be used to encrypt further communication.

The implementation consists of one or more authentication servers running on physically secure hosts. The authentication servers maintain a database of principals (i.e., users and servers) and their secret keys. In order to add authentication to its transactions, a typical network application adds one or two calls to the Kerberos library, which results in the transmission of the necessary messages to achieve authentication.

6.2.2 Message Processing

There are two methods by which a client can ask a Kerberos server for credentials. In the first approach, the client sends a clear text request for a ticket for the desired server to the AS. The reply is sent encrypted in the client's secret key. Usually this request is for a ticket-granting ticket (TGT), which can later be used with the ticket-granting server (TGS). In the second method, the client sends a request to the TGS. The client sends the TGT to the TGS in the same manner as if it were contacting any other application server, which requires Kerberos credentials. The reply is encrypted in the session key from the TGT.

Once obtained, credentials may be used to verify the identity of the principals in a transaction, to ensure the integrity of messages exchanged between them, or to preserve privacy of the messages. The application is free to choose whatever protection may be necessary. To verify the identities of the principals in a transaction, the client transmits the ticket to the server. Since the ticket is sent "in the clear" (parts of it are encrypted, but this encryption does not thwart replay) and might be intercepted and reused by an attacker, additional information is sent to prove that the message was originated by the principal to whom the ticket was issued. This information (called the authenticator) is encrypted in the session key, and includes a timestamp. The timestamp proves that the message was recently generated and is not a replay. Encrypting the authenticator in the session key proves that a party possessing the session key generated it. Since no one except the requesting principal and the server know the session key (it is never sent over the network in the clear) this guarantees the identity of the client.

The integrity of the messages exchanged between principals can also be guaranteed using the session key (passed in the ticket and contained in the credentials). This approach provides detection of both replay attacks and message stream modification attacks. It is accomplished by generating and transmitting a collision-proof checksum (elsewhere called a hash or digest function) of the client's message, keyed with the session key. Privacy and integrity of the messages exchanged between principals can be secured by encrypting the data to be passed using the session key passed in the ticket, and contained in the credentials.

The authentication exchanges mentioned above require read-only access to the Kerberos database. Sometimes, however, the entries in the database must be modified, such as when adding new principals or changing a principal's key.

6.2.3. TGS-Exchange

Message direction	Message type
Client to Kerberos	KRB_TGS_REQ
Kerberos to client	KRB_TGS_REP

The TGS exchange between a client and the Kerberos Ticket-Granting Server is initiated by a client when it wishes to obtain authentication credentials for a given server (which might be registered in a remote realm). In the first case, the client must already have acquired a ticket for the Ticket-Granting Service using the AS exchange (the ticket-granting ticket is usually obtained when a client initially authenticates to the system, such as when a user logs in). The message format for the TGS exchange is almost identical to that for the AS exchange. The primary difference is that encryption and decryption in the TGS exchange does not take place under the client's key. Instead, the Session key from the ticket-granting ticket is used.

The TGS exchange consists of two messages:

- A request (KRB_TGS_REQ) from the client to the Kerberos Ticket-Granting Server.
- A reply (KRB_TGS_REP or KRB_ERROR).

The KRB_TGS_REQ message includes information authenticating the client plus a request for credentials. The authentication information consists of the authentication header (KRB_AP_REQ).

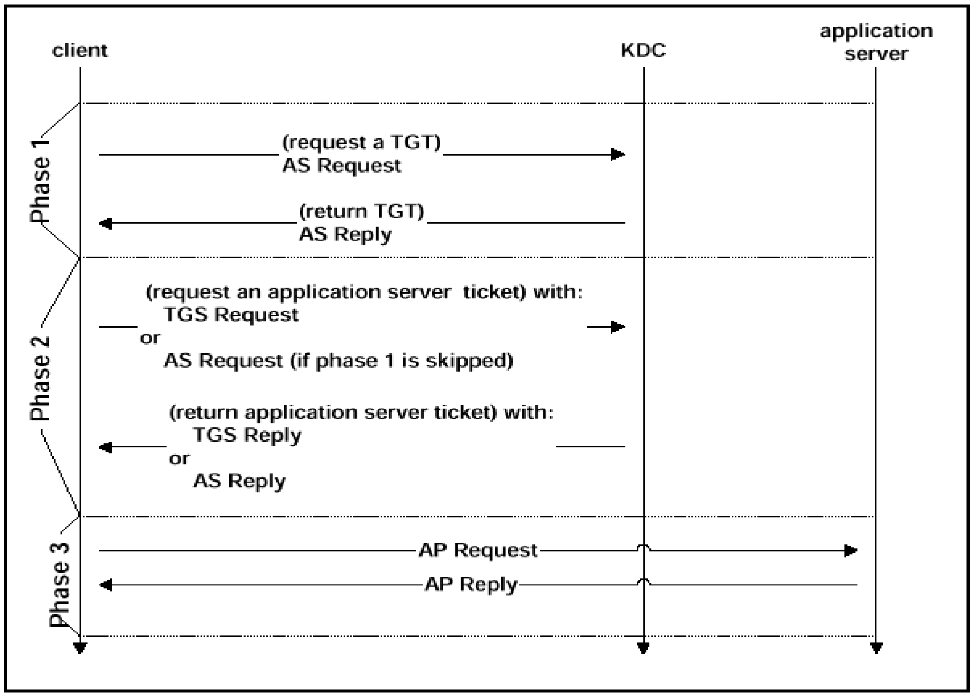


Figure 6.1 – TGS Exchanges

The TGS reply (KRB_TGS_REP) contains the requested credentials, encrypted in the session key from the ticket-granting ticket. The KRB_ERROR message contains an error code and text explaining what went wrong. The KRB_ERROR message is not encrypted. The KRB_TGS_REP message contains information, which can be used to detect replays, and to associate it with the message to which it replies. The KRB_ERROR message also contains information which can be used to associate it with the message to which it replies, but the lack of encryption in the KRB_ERROR message precludes the ability to detect replays or fabrications of such messages.

Generation of KRB_TGS_REQ Message

Before sending a request to the ticket-granting service, the client must determine in which realm the application server is registered. It might be known beforehand (since the realm is part of the principal identifier). Presently, however, this information is obtained from a configuration file. If the client does not already possess a ticket-granting ticket for the appropriate realm, then one must be obtained. This is first attempted by requesting a ticket-granting ticket for the destination realm from the local Kerberos server (using the KRB_TGS_REQ message recursively). The Kerberos server may return a TGT for the desired realm in which case one can proceed. Alternatively, the Kerberos server may return a TGT for a realm which is "closer" to the desired realm (further along the standard hierarchical path), in which case this step must be repeated with a Kerberos server in the realm specified in the returned TGT. If neither are returned, then the request must be retried with a Kerberos server for a realm higher in the hierarchy. This request will itself require a ticket-granting ticket for the higher realm which must be obtained by recursively applying these directions.

Once the client obtains a ticket-granting ticket for the appropriate realm, it determines which Kerberos servers serve that realm, and contacts one. The list might be obtained through a configuration file or network service; as long as the secret keys exchanged by realms are kept secret, only denial of service results from a false Kerberos server.

As in the AS exchange, the client may specify a number of options in the KRB_TGS_REQ message. The client prepares the KRB_TGS_REQ message, providing an authentication header as an element of the padata field, and including the same fields as used in the KRB_AS_REQ message along with several optional fields: the enc-authorization-data field for application server use and additional tickets required by some options.

Once prepared, the message is sent to a Kerberos server for the destination realm.

Receipt of KRB_TGS_REQ Message

The KRB_TGS_REQ message is processed in a manner similar to the KRB_AS_REQ message, but there are many additional checks to be performed. First, the Kerberos server must determine which server the accompanying ticket is for and it must select the appropriate key to decrypt it. For a normal KRB_TGS_REQ message, it will be for the ticket granting service, and the TGS's key will be used. If no ticket can be found in the padata field, the KDC_ERR_PADATA_TYPE_NOSUPP error is returned.

Once the accompanying ticket has been decrypted, the user-supplied checksum in the Authenticator must be verified against the contents of the request, and the message rejected if the checksums do not match (with an error code of KRB_AP_ERR_MODIFIED) or if the checksum is not keyed or not collision-proof (with an error code of KRB_AP_ERR_INAPP_CKSUM). If the checksum type is not supported, the KDC_ERR_SUMTYPE_NOSUPP error is returned. If any of the decryptions indicate failed integrity checks, the KRB_AP_ERR_BAD_INTEGRITY error is returned.

Generation of KRB_TGS_REP Message

The KRB_TGS_REP message shares its format with the KRB_AS_REP (KRB_KDC_REP), but with its type field set to KRB_TGS_REP. The response will include a ticket for the requested server. The Kerberos database is queried to retrieve the record for the requested server (including the key with which the ticket will be encrypted). If the request is for a ticket granting ticket for a remote realm, and if no key is shared with the requested realm, then the Kerberos server will select the realm "closest" to the requested realm with which it does share a key, and use that realm instead. This is the only case where the response from the KDC will be for a different server than that requested by the client.

By default, the address field, the client's name and realm, the list of transited realms, the time of initial authentication, the expiration time, and the authorization data of the newly-issued ticket will be copied from the ticket-granting ticket (TGT). If the transited field needs to be updated, but the transited type is not supported, the KDC_ERR_TRTYPE_NOSUPP error is returned.

If the request specifies an endtime, then the endtime of the new ticket is set to the minimum of

- that request
- the endtime from the TGT
- the starttime of the TGT plus the minimum of the maximum life for the application server and the maximum life for the local realm (the maximum life for the requesting principal was already applied when the TGT was issued).

If the ENC-TKT-IN-SKEY option has been specified and an additional ticket has been included in the request, the KDC will decrypt the additional ticket using the key for the server to which the additional ticket was issued and verify that it is a ticket-granting ticket. If the name of the requested server is missing from the request, the name of the client in the additional ticket will be used. Otherwise the name of the requested server will be compared to the name of the client in the additional ticket and if different, the request will be rejected. If the request succeeds, the session key from the additional ticket will be used to encrypt the new ticket that is issued instead of using the key of the server for which the new ticket will be used (This allows easy implementation of user-to-user authentication, which uses ticket-granting ticket session keys in lieu of secret server keys in situations where such secret keys could be easily compromised).

Whenever a request is made to the ticket-granting server, the presented ticket(s) is(are) checked against a hot-list of tickets which have been canceled. This hot-list might be implemented by storing a range of issue dates for "suspect tickets"; if a presented ticket had an authtime in that range, it would be rejected. In this way, a stolen ticket-granting ticket cannot be used to gain additional tickets (renewals or otherwise) once the theft has been reported. Any normal ticket obtained before it was reported stolen will still be valid (because they require no interaction with the KDC), but only until their normal expiration time.

The ciphertext part of the response in the KRB_TGS_REP message is encrypted in the session key from the ticket-granting ticket. It is not encrypted using the client's secret key. Furthermore, the client’s key’s expiration date and the key version number fields are left out since these values are stored along with the client's database record, and that record is not needed to satisfy a request based on a ticket-granting ticket.

Receipt of KRB_TGS_REP Message

When the KRB_TGS_REP is received by the client, it is processed in the same manner as the KRB_AS_REP processing described above. The primary difference is that the ciphertext part of the response must be decrypted using the session key from the ticket-granting ticket rather than the client's secret key.

6.3. Kerberos/PKINIT

The popularity of public key cryptography has produced a desire for its support in Kerberos. The advantages provided by public key cryptography include simplified key management (from the Kerberos perspective) and the ability to leverage existing and developing public key certification infrastructures. Public key cryptography can be integrated into Kerberos in a number of ways. One is to associate a key pair with each realm. Another way is to allow users with public key certificates to use them in initial authentication.

PKINIT utilizes ephemeral-ephemeral Diffie-Hellman keys in combination with digital signature keys as the primary, required mechanism. It also allows for the use of RSA keys and/or (static) Diffie-Hellman certificates. In particular PKINIT supports the use of separate signature and encryption keys.

PKINIT enables access to Kerberos-secured services based on initial authentication utilizing public key cryptography. PKINIT utilizes standard public key signature and encryption data formats within the standard Kerberos messages. The basic mechanism is as follows:

The user sends an AS-REQ message to the KDC as before, except that if that user is to use public key cryptography in the initial authentication step, his certificate and a signature accompany the initial request in the pre-authentication fields. Upon receipt of this request, the KDC verifies the certificate and issues a ticket granting ticket (TGT) as before, except that the encPart from the AS-REP message carrying the TGT is now encrypted utilizing either a Diffie-Hellman derived key or the user's public key. This message is authenticated utilizing the public key signature of the KDC.

Note that PKINIT does not require the use of certificates. A KDC may store the public key of a principal as part of that principal's record. In this scenario, the KDC is the trusted party that vouches for the principal (as in a standard, non-cross realm, Kerberos environment). Thus, for any principal, the KDC may maintain a secret key, a public key, or both.

6.3.1. PKINIT Exchange in Packet Cable

The PKINIT Request is carried as a Kerberos pre-authenticator field inside an AS_Request and the PKINIT Reply is a pre-authenticator inside the AS Reply. The PKINIT client is referred to as an MTA, as it is currently the only Packet Cable element that authenticates itself to the KDC with the PKINIT protocol.

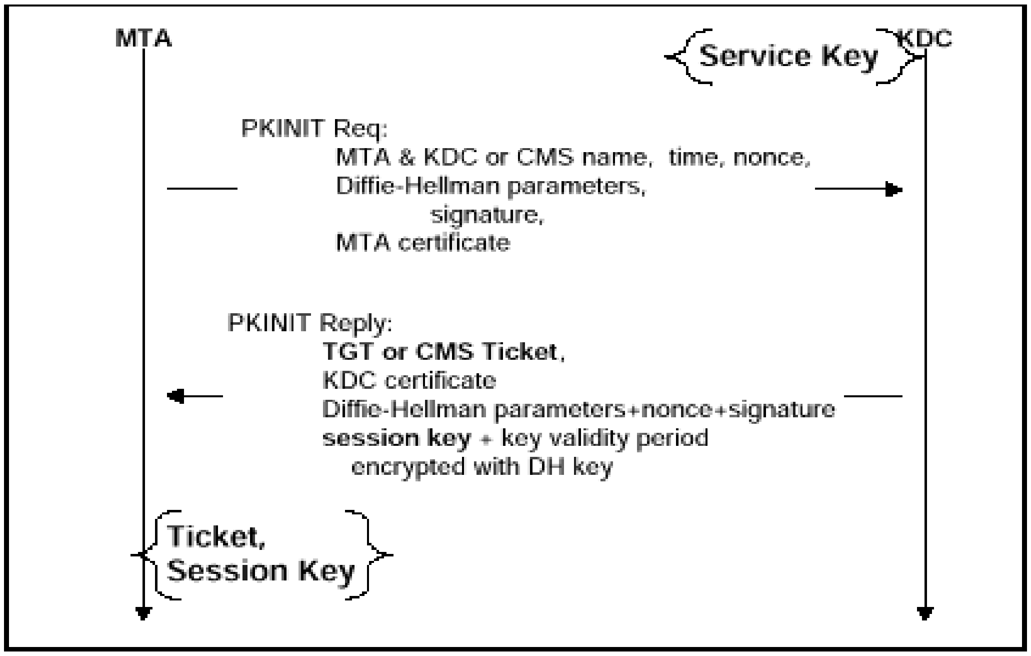


Figure 6.2 – PKINIT Exchange

The above diagram lists several important parameters in the PKINIT Request and Reply messages. These parameters are:

6.3.2. PKINIT Request

MTA (Kerberos principal) name **–** found in the KDC-REQ-BODY Kerberos structure. Its format is based on the MTA's X.500name in the certificate.

KDC or Application Server (Kerberos principal) name – found in the KDC-REQ-BODY Kerberos structure.

Time – found in the PKAuthenticator structure, specified by PKINIT

Nonce - found in the PKAuthenticator structure, specified by PKINIT (There is also a 2 nd nonce in the KDC-REQ-BODY Kerberos structure).

Diffie-Hellman parameters, signature and MTA certificate – these are all specified by PKINIT.

6.3.3. PKINIT Reply

Application Server Ticket – found in the KDC-REP Kerberos structure.

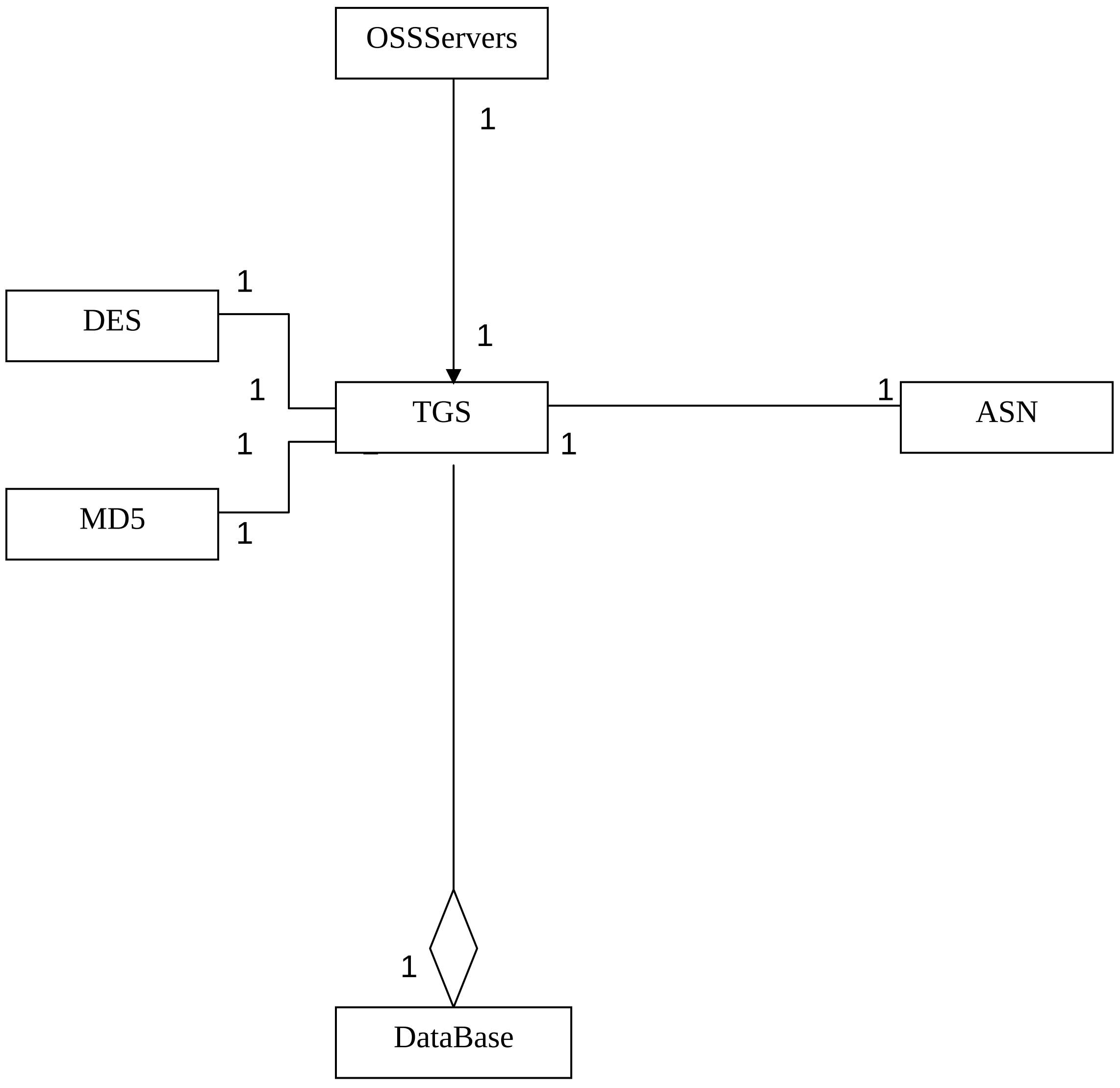
KDC Certificate, Diffie-Hellman parameters, signature – these are all specified by PKINIT.

Nonce – found in the KdcDHKeyInfo structure. This nonce must be the same as the one found in the PKAuthenticator structure of the PKINIT Request. There is another nonce in EncKDCRepPart Kerberos structure This nonce must be the same as the one found in the KDC-REQ-BODY of the PKINIT Request.

Session key, key validity period – found in the EncKDCRepPart Kerberos structure

6.4 UML Design

6.4.1. Class Diagram



6.4.2. Classes

6.4.2.1. OSSServers

This is the main parent class from which all the server classes are derived.

6.4.2.2. TGS

This class represents the Ticket Granting Server that is a child of the OSS Servers class.

6.4.2.3. Database

This class represents the Database maintained by the Ticket Granting Server to keep record of a variety of information.

6.4.2.4. MD5

This class is used to produce a hash using the MD5 digest algorithm on a given data.

6.4.2.5. 3DESCBC

This class is used to encrypt a message using a given key.

6.4.3. Sequence Diagram

The sequence diagram (attached in APPENDIX A)can be divided into two main portions.

- Request processing
- Reply processing

6.4.3.1. Request Processing

The client sends a encoded request to the Ticket Granting Server when it wants to establish a session with the call management server.

The request, which is sent to the Ticket Granting Server is composed of two things.

- Req-Body
- Pre-authentication Data

6.4.3.1.1 Req-Body

The request body consists of time and nonce to check for replay attacks, along with the client and server names which have to be looked up in the database for authentication.

6.4.3.1.2. Preauthentication Data

The pre-authentication field in the request mainly consists of the Signature and the Certificate.

When the Ticket Granting Server receives the request it first sends the request to the ASN complier to be decoded .The name of the client is looked up in the database to check if the client is a valid member of our system .If the client is not a valid member of our system then an error is returned to the client indicating that the principal or the client is unknown to the ticket granting server. However if the result of the search is true or the client is a valid member then we check the name of the server, which the client is trying to access .If the server is not a valid part of the database then we return an error to the client that the server is unknown to the Ticket Granting Server. The time has to be matched to ensure that it is not a replay.

The Ticket Granting Server has to now pass through the pre-authentication phase after it has successfully authenticated the client and the server. First the ticket granting server checks the pre-authentication data type .If the pre-authentication type is null then an error is returned to the client indicating that pre-authentication is required. The pre-authentication data value contains the encoded signed data or the PA_PK_AS_REQ so this is sent to the ASN compiler to be decoded .The Ticket Granting Server checks for the digest algorithm, if the digest algorithm is not sha-1 then an error is returned to the client indicating that the digest is invalid and if the digest is valid then it compares the client name that is sent to the Ticket Granting Server in the request body and the client name that is obtained from the certificate, if they mismatch then an error is returned to the client indicating a client name mismatch.

The Ticket Granting Server has to now verify the signature .The certificate in the signed data structure contains the encoded value of the public key .The Ticket Granting Server sends the RSA-public-key to the ASN complier so that it can be decoded. Now the value of the public key is obtained from the certificate and the value of the signature is obtained from the signerinfo (PA_PK_AS_REQ). These two parameters are passed to the RSA encryption algorithm which returns the sha-1 digest that has been sent to the Ticket Granting Server by the client.

The eContent value in the signed data (PA_PK_AS_REQ) contains the encoded value of the auth pack so the Ticket Granting Server sends the value of the authpack to the ASN compiler to be decoded. SHA-1 digest algorithm is now applied to the pkAuthenticator structure in the authpack. The digest that is obtained is compared with the one that was obtained by applying the RSA algorithm over the public key and the signature value. If the two values mismatch then the pre-authentication has failed and an error is returned to the client indicating that the pre-authentication has not been successful .The authpack contains the encoded value of the dh parameters .The dh parameters in the authpack are checked, if the range of parameters is not valid then an error is returned to the client indicating that the range is violated. Similarly the Ticket Granting Server also checks the strength of the key, if the key is weak an error is returned to the client indicating that the key is weak.

6.4.3.2. Reply Processing

After the Ticket Granting Server has finished the processing of the request it prepares to send data to the client so that the client can establish a session with the Call Management Server. The data, which has to be sent to the client, is mainly composed of the following:

- Pre-authentication data
- Ticket
- Client Data

6.4.3.2.1. Pre-authentication Data

The DH public key value is first encoded since it is part of the kdc-dhkeyinfo structure of the pre-authentication data. The Ticket Granting server passes the encoded kdc-dhkeyinfo as parameter to the digest algorithm sha-1. Applying the message digest algorithm (SHA-1) on the message (kdc-dhkeyinfo) returns the digest to the Ticket Granting Server. The Ticket Granting Server obtains the private key of the server from the database. Then the message digest is encrypted with the private key of Ticket Granting Server by applying rsa encryption algorithm. This process prepares the signature that is to be sent to the client. A certificate is also sent to the client which carries the public key that will be used at the client end for pre-authentication.

6.4.3.2.2. Ticket

The ticket is a record or structure that the Ticket Granting Server generates for the Call Management Server. Ticket Granting Server generates a session key, which is to be sent as part of the ticket. The ticket is sent encrypted to the client in the Call Management Server secret key, which is obtained from the database. The ticket is encrypted with the encryption type set to des3-cbc-md5. The following data must be concatenated and processed in the following sequence before being encrypted with 3-DES CBC, IV=0:

- 8-byte random byte sequence, called a confounder
- MD5 checksum over the ticket
- Ticket that is to be encrypted
- Random padding up to a multiple of 8

6.4.3.2.3. Client Data

The data that is sent to the client is encrypted in the DH key that is derived from the Deffie-Hellman parameters sent to the Ticket Granting Server as part of the certificate in the pre-authentication data. The encryption is done with the same encryption as was done for the ticket i.e. DES algorithm in CBC mode with the MD5 message digest.

All these fields pre-authentication, ticket, client data in addition to some other fields are first encoded using the ASN.1 compiler and then sent to the client which uses this data to establish a session with the Call Management Server.

6.5. ASN Structures for Ticket Granting Server

The ASN.1 structures employed in Ticket Granting Server are as attached in the Appendix A. These files were fed as input to the ASN.1 compiler and the output files generated are as attached in the Appendix A.

The C++ structures generated by the ASN.1 compiler are discussed in the following paragraphs with respect to the hierarchical relation between these structures.

First of all, there are **PDU**(s) (Protocol Data Unit(s)) defined for each of the independent structures and are used for the encoding and decoding of values of those structures.

Next, the first main structure is the **KDC_REQ**, which basically consists of the values sent as request by the client to the server. This consists of the following:

<u>Bit_mask :</u> This is an unsigned character (8 bits) and indicates whether the

KDC_REQ_padata : This is optional. The presence is indicated by the bit sequence equivalent to 0x80.

<u>Pvno :</u> This is an integer and consists of the version number.

<u>msg_type :</u> This an integer and indicates the message type.

_seqof3 : This is a structure and consists of the pointer to the next _seqof3 structure in sequence as well as the value of type PA_DATA (which is discussed below).

req_body : This is of type KDC_REQ_BODY (which is discussed below).

The KDC_REQ further refers to two main structures, which are the PA_DATA and the KDC_REQ_BODY. First of all, the PA_DATA structure consists of the following:

padata_type : This is an integer value corresponding to the padata type

padata_value : This is a structure which further consists of the encoded value of the PA_PK_AS_REQ and the length of this encoding.

In this case, the PA_PK_AS_REQ consists of:

SignedAuthPack : This is of type SignedData

SignedData : further consists of the following information:

bit_mask : This is an unsigned character and consists of a sequence of bits which represent whether or not the ‘certificates’ and/or the ‘crls’ are present or not.

Version : This is of type CMSVersion, which basically indicates the version number of the CMS indicated by a pre-defined integer value.

DigestAlgorithms : This is of type DigestAlgorithmIdentifiers, which is mainly a structure referring to the DigestAlgorithmIdentifier. The DigestAlgorithmIdentifier then further refers to the AlgorithmIdentifier, which is supposed to have the ObjectID corresponding to the algorithm employed, which in this case would be SHA-1.

EncapContentInfo : This is of type EncapsulatedContentInfo, which in this case, is supposed to have the encoding of the AuthPack in the 'value' field of the eContentType structure. The main fileds in the AuthPack are the 'pkAuthenticator' and the 'clientPublicValue'. The pkAuthenticator is supposed to have the general information like the kdcname, kdcRealm, cusec, ctime and nonce. On the other hand, the clientPublicValue, which is of type SubjectPublicKeyInfo, has the AlgorithmIdentifier, which is RSA in this case, and the _bit1, which has the encoded value and length of the DomainParameters. The DomainParameters are basically the DH-parameters.

Certificates : This is of type CertificateSet, which further refers the TBSCertificate via the CertificateChoices and the Certficate. It is here that the encoded value of the RSA public key is stored in the SubjectPublicKeyInfo's '_bit1' field.

SignerInfos : This is of type SignerInfo, which further refers to the SignatureValue, where the encoded value of the digest of the pkAuthenticator is placed. This value is encrypted before being encoded.

Next is the KDC_REQ_BODY which consists of the following fields:

bit_mask : This is an unsigned character, which indicates whether the sname, cname and from fields exist or not.

Cname : This is of type PrincipalName, which mainly consists of info regarding the client.

Realm : This is of type Realm, which offers information regarding the realm.

Sname : This is of type PrincipalName, which this time consists of information regarding the server.

From : This is of type KerberosTime, and consists of the requested starting time of the session.

Till : This is of type KerberosTime, and consists of the requested termination time of the session.

Nonce : This is basically a random number generated at the client end and is retransmitted in the reply so that the client can be assured that the reply is from the true server.

This concludes the request part of the Ticket Granting Server. Next is the reply part that is the reply to be sent from the server to the client. The reply part has the main structure **KDC_REP**. This consists of the following main fields:

bit_mask : This is of type unsigned character, and indicates whether the padata is present or not.

Padata : This is a structure of type _seqof4 and consists of the PA_DATA value. This PA_data value is now supposed to have the encoding of the PA_PK_AS_REP, which consists of the pre-authentication information for the client.

Crealm : This is of type Realm, which consists of the information of the client's realm.

Cname : This is of type PrincipalName, which consists of the information regarding the client.

Ticket : This is of type Ticket, which is supposed to have general information regarding the ticket and an enc_part, which is of type EncryptedData. The main information in the EncryptedData is the encoded value of the encrypted value of the EncTicketPart, confounder and the checksum of the EncTicketPart. This is placed in the value field of the cipher structure. The EncTicketPart is supposed to have the 'session key' for the server and plus other general information regarding the client.

enc_part : This is of type EncryptedData, which is now supposed to have the encoded value of the EncKDCRepPart in the value field of the cipher structure. The EncKDCRepPart is supposed to have the session key for the client and plus other general information regarding the server.

The third important structure is the **KRB_ERROR**, which is to be employed whenever some kind of an error is generated. The main field here is the 'error_code', which indicates the type of error that has occurred. Other fields provide general information time, realm, name etc.

6.6. Database for the Ticket Granting Server

The Data Base in the Ticket Granting Server has the following contents:

Field	**Value**
name	Principal's identifier
key	Principal's secret key
p_kvno	Principal's key version
max_life	Maximum lifetime for Tickets

The purpose of the above mentioned fields are briefly explained as follows

- The name field is an unsigned character array and contains the principal's identifier that is either the client or the server name.
- The key field is an unsigned character array and contains an encryption key. This key is the principal's secret key.
- The p_kvno field is an integer value corresponding to the key version number of the principal's secret key.
- The max_life field is an integer value and contains the maximum allowable lifetime for any ticket issued for this principal by the server.

The Data Base is employed by the Ticket Granting Server (TGS) for authenticating principles (that is, either clients or servers). Initially, the records of the Data Base are stored in a file. These entries are sorted according to the principle name and in case of entry of a new record, which is achieved by employing the insert() function, the sorting algorithm is applied again, by employing the compare() and sortfile() functions. These functions employ the quick sort algorithm for achieving the required sorting.

Now, in order to allow the TGS to access the records in the Data Base, these records are read into an array by using the readfile() function. Thus, in order to perform any operation on these records the TGS accesses this array and can accordingly perform the search for a specific principle by employing the binsearch() function call. This call results in a binary search over the elements of the array and in case of success returns the index of the corresponding record in the array. Otherwise, -1 is returned in case of failure.

6.7. Encryption Algorithm Employed

The encryption algorithm used is threeDESCBC.

6.8. Digest Algorithm Employed

The digest algorithm used is MD5.

CHAPTER 7
RECORD KEEPING SERVER

7.1. Introduction

The Record Keeping Server is responsible for collecting billing events and reporting the back to the billing system. The Record Keeping Server (RKS) is a trusted network element function. The RKS is the mediation layer between the call signaling and transport layer and the back-office applications. The RKS is expected to pre-process the data from the Call Signaling and Transport layer and present it to the back-office applications in the format and within the time constraints deemed necessary by the MSO.

The Record Keeping Server also, at a minimum, is a short-term repository for PacketCable Event Messages. It receives Event Messages from various trusted PacketCable network elements. The RKS assembles the Event Messages into coherent sets, which are then made available to a usage-processing platform and potentially to several other back office systems. It acts as the demarcation point between the PacketCable network and the back office applications.

7.2. Integrity of Record Keeping Server

Security of the Record Keeping Server is of vital importance. A compromised Record Keeping Server may result in

- Free or reduced service;
- Billing to a wrong account;
- Billing customers for communications that were never made;

- Unauthorized disclosure of the customer identities and personal information.

7.3. Responsibilities

The Record Keeping Server performs the following functions:

- Receives Event Messages.
- Correlates all Event Messages related to an individual call
- Assembles events and determine completeness. This includes the capability to distinguish Event Messages, and recognize when a complete set, representing a coherent set of billing data is available for transport to the back office system.
- Can store the Event Messages for at least one week or until sent to the other back office systems and successful receipt is acknowledged from those systems.

The Record Keeping Server receives messages from a Call Management Server, Cable Modem Termination System and also a Media Gateway Controller. Either of these initiators that want to communicate with the Record Keeping Server does so by first generating a mutually agreed upon key through Internet Key Exchange and then sending the required information encrypted in with this key using the RADIUS protocol.

7.4. Record Keeping Server Interfaces

7.4.1. CMS-RKS Interface

CMS and RKS will negotiate a shared secret (CMS-RKS Secret) using IKE. IKE uses one of the modes with pre-shared keys for this interface. IKE runs asynchronous to the billing event generation and will guarantee that there is always a valid, non-expired CMS-RKS Secret. This shared secret is unique to this particular CMS and RKS.

7.4.2. CMTS-RKS Interface

CMTS and RKS negotiate a shared secret (CMTS-RKS Secret) using IKE. IKE uses one of the modes with pre-shared keys.
IKE runs asynchronous to the billing event generation and will guarantee that there is always a valid, non-expired CMTS-RKS Secret. This shared secret is unique to this particular CMTS and RKS.
The RKS communicates with other entities in such a manner that first a secret key is established using the Internet Key Exchange and then this key is used to provide security to RADIUS event messages that are passed from initiator to responder and vice-versa.

7.5. ISAKMP And IKE

7.5.1. Introduction

The IPSec protocol suite is used to provide privacy and authentication services at the IP layer. However, secure internet sessions need uni-directional Security Associations (SA) between the communicating parties. A Security association describes what operations should be applied to a packet. The information that security association specifies include:

- An authentication method;
- An encryption algorithm;
- Encryption and authentication keys;
- A lifetime of the encryption key;

- A lifetime of the security association;
- A sequence number for replay prevention.

Secure associations can be set manually or automatically. The automatic secure association management is required to make possible deployment of IPSec or when on-demand creation of security association is needed.

Internet Key Exchange (IKE) is an automated protocol for establishing, negotiating, modifying, and deleting security associations between two hosts in a network. Security associations contain information to establish a secure connection between the parties on pre-defined manners.

The IKE is a combination of the Internet Security Association and Key Management Protocol (ISAKMP), Oakley, SKEME. ISAKMP is a key exchange independent framework for authentication, security association management, and establishment. Oakley defines series of key exchanges and services provided by each of them. SKEME defines a key exchange which provides anonymity, repudiability, and fast key refreshment.

7.5.2. Internet Security Association and Key Management Protocol (ISAKMP)

ISAKMP defines the procedures for authentication of communicating peers, creation and management of security associations, key generation techniques, and threat mitigation. It provides a framework for the Internet Key Exchange (IKE), which can be identified as one implementation of ISAKMP to be used with IPSec.

The ISAKMP negotiation is divided into two separate phases.

(i) In the first phase, ISAKMP Security Association is established between two entities to protect further negotiation traffic.

(ii) In the second phase, the security association for some security protocol is negotiated and established. One ISAKMP security association can be used to establish many security associations for other protocols.

Two phase approach is chosen to allow establishing many security associations without the need to start over for each communication and thus reducing the cost of ISAKMP management by reducing the need to go through costly re-authentication.

7.5.2.1. ISAKMP Exchanges

ISAKMP mainly comprises of the following five main types of exchanges:

7.5.2.1.1. Base Exchange

The Base Exchange is designed to allow the key exchange and authentication related information to be transmitted together. This reduces the number of round-trips at the expense of identity protection. Identity protection is not provided in this exchange because identities are established before a common shared secret has been established.

7.5.2.1.2. Identity Protection Exchange

The Identity Protection Exchange is designed to separate the Key Exchange information from the identity and authentication related information. A common share secret can be established before identification and authentication related information is exchanged and encryption provides protection of the communicating identities. This identity protection is achieved at the expense of two additional messages as compared to the base exchange.

7.5.2.1.3. Authentication Only Exchange

The Authentication Only Exchange allows only the authentication related information to be transmitted. The benefit of this exchange is the ability to perform only authentication without the computational expense of computing keys.

7.5.2.1.4. Aggressive Exchange

The Aggressive Exchange allows the security association, key exchange and authentication related payloads to be transmitted together. Combining all this information into one message reduces the number of round-trips at the expense of not providing identity protection.

7.5.2.1.5. Informational Exchanges

The Informational Exchange is one-way transmittal that can be used for security association management. If the Informational Exchange occurs to the keying material exchange during an ISAKMP phase one then there will be no protection for the Informational Exchange.

7.5.3. Internet Key Exchange (IKE)

Internet Key Exchange is an implementation of the ISAKMP and is currently being used with IPSec, though it can be employed with a number of protocols in the future. IKE mainly achieves the automatic configuration of keying material and security associations, which not only helps improve the security issues but furthermore relieves the complications of manual configuration. Moreover, it also supports the refreshing of keys and obtaining of multiple security associations for the protocol, based on the initially established security associations of the IKE.

IKE can be employed in the negotiation of virtual private networks (VPN) and also for providing a remote user from a remote site, with an unknown IP address, to access a secure host or network. Moreover, it also incorporates features which provide Perfect Forward Secrecy (PFS), that is, the compromise of one key will not effect the transmission of data encrypted with subsequent keys. This is achieved by preventing the usage of a key to derive subsequent keys.

IKE negotiations must be protected, so each IKE negotiation begins by each peer agreeing on a common (shared) IKE policy. This policy states which security parameters will be used to protect subsequent IKE negotiations. After the two peers agree upon a policy, the security parameters of the policy are identified by a security association established at each peer, and these security associations apply to all subsequent IKE traffic during the negotiation.

When the IKE negotiation begins, IKE looks for an IKE policy that is the same on both peers. The peer that initiates the negotiation will send all its policies to the remote peer, and the remote peer will try to find a match. The remote peer looks for a match by comparing its own highest priority policy against the other peer's received policies. The remote peer checks each of its policies in order of its priority (highest priority first) until a match is found.

A match is made when both policies from the two peers contain the same encryption, hash, authentication, and Diffie-Hellman parameter values, and when the remote peer's policy specifies a lifetime less than or equal to the lifetime in the policy being compared. (If the lifetimes are not identical, the shorter lifetime--from the remote peer's policy--will be used).

If no acceptable match is found, IKE refuses negotiation and IPSec will not be established. If a match is found, IKE will complete negotiation, and IPSec security associations will be created.

Multiple IKE policies can be created, each with a different combination of parameter values. For each policy that is created, a unique priority (1 through 10,000, with 1 being the highest priority) is assigned.

Multiple policies can be configured on each peer--but at least one of these policies must contain exactly the same encryption, hash, authentication, and Diffie-Hellman parameter values as one of the policies on the remote peer.

7.5.3.1. IKE Modes

IKE comprises of the following four main modes. These are as follows:

7.5.3.1.1. Main Mode

The Main Mode is an exchange in the first phase of IKE/ISAKMP, and falls under the ISAKMP Identity Protection Exchange. The first two messages are used for negotiating the security policy for the exchange, the next two messages are used for the Diffie-Hellman keying material exchange and the last two messages are used for authenticating the peers with signatures or hashes and optional certificates. Last two authentication messages are encrypted with the previously negotiated key, thus protecting the identities of the parties from eavesdroppers.

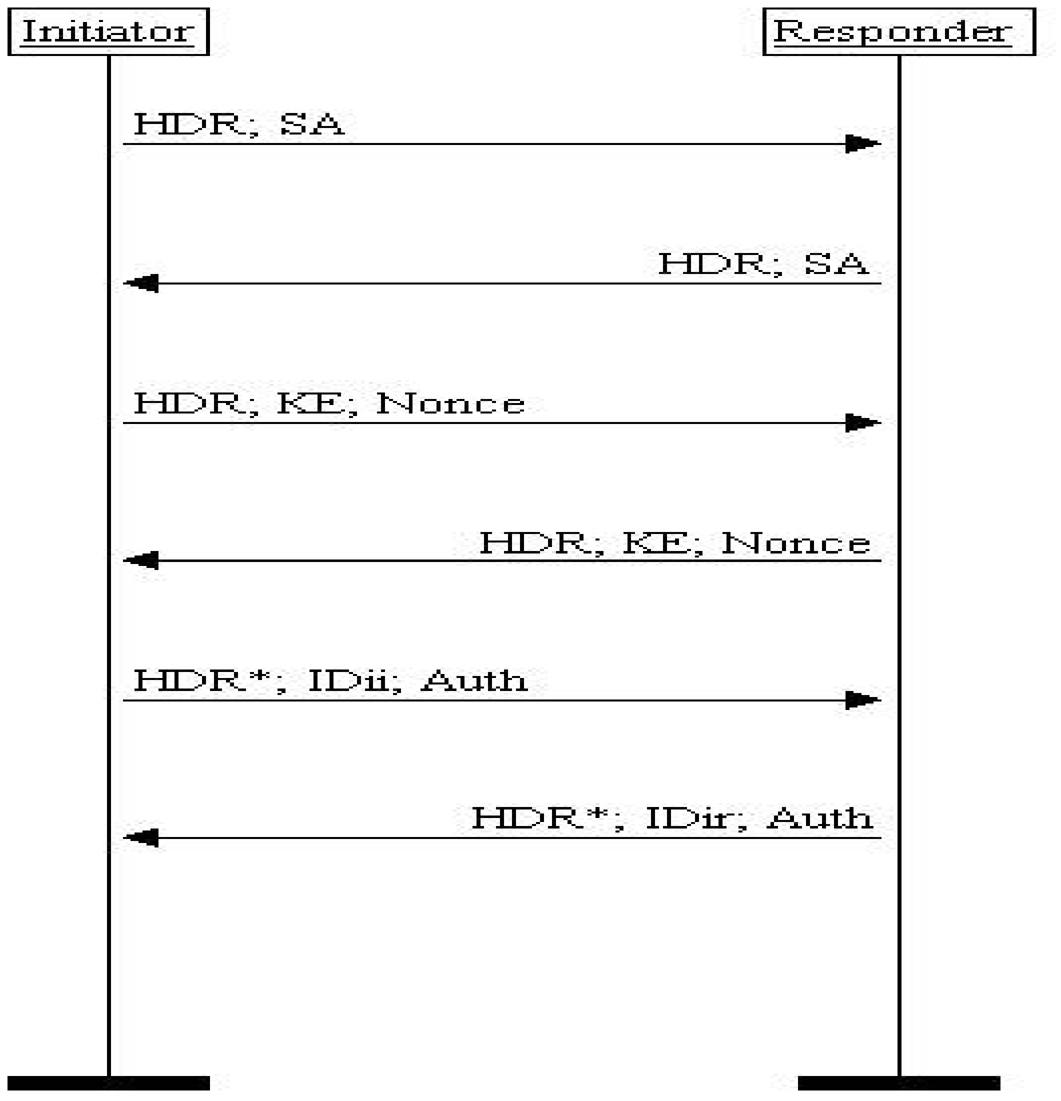


Figure 7.1 – Main Mode Exchanges

7.5.3.1.2. Aggressive Mode

The Aggressive Mode is an exchange in the first phase of IKE/ISAKMP and falls under the ISAKMP Aggressive Exchange. The first message proposes the policy, and passes data for key-exchange, the nonce and some information for identification. The second message is a response which authenticates the responder and concludes the policy and key-exchange. At this point all the information for encryption key for the ISAKMP SA is exchanged and last the message could be encrypted, however, this is not mandatory. The last message is used for authenticating the initiator and provides a proof of participation in the exchange. The identity of the responder could not be protected, but by encrypting the last message the identity of the initiator is protected.

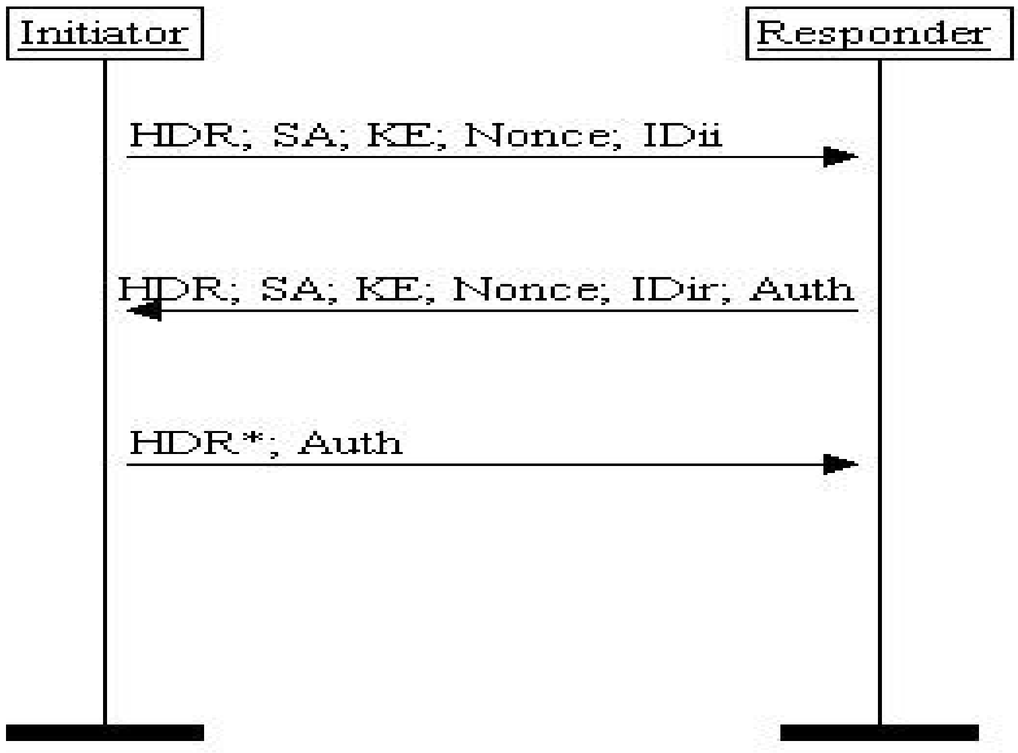


Figure 7.2 – Aggressive Mode Exchanges

7.5.3.1.3. Quick Mode

The Quick Mode is used for exchange in the second phase of IKE. An IKE security association is established in the first phase to protect the second phase exchange by previously described Main or Aggressive Mode. The Quick Mode is used for negotiating security association and generating new keying material. All the payloads except ISAKMP header are encrypted. A Diffie-Hellman key exchange may be done to achieve perfect forward secrecy. Many security associations can be negotiated during one Quick Mode exchange. Either one of the parties might initiate the quick mode exchange, independent of the fact regarding who initiated the first phase.

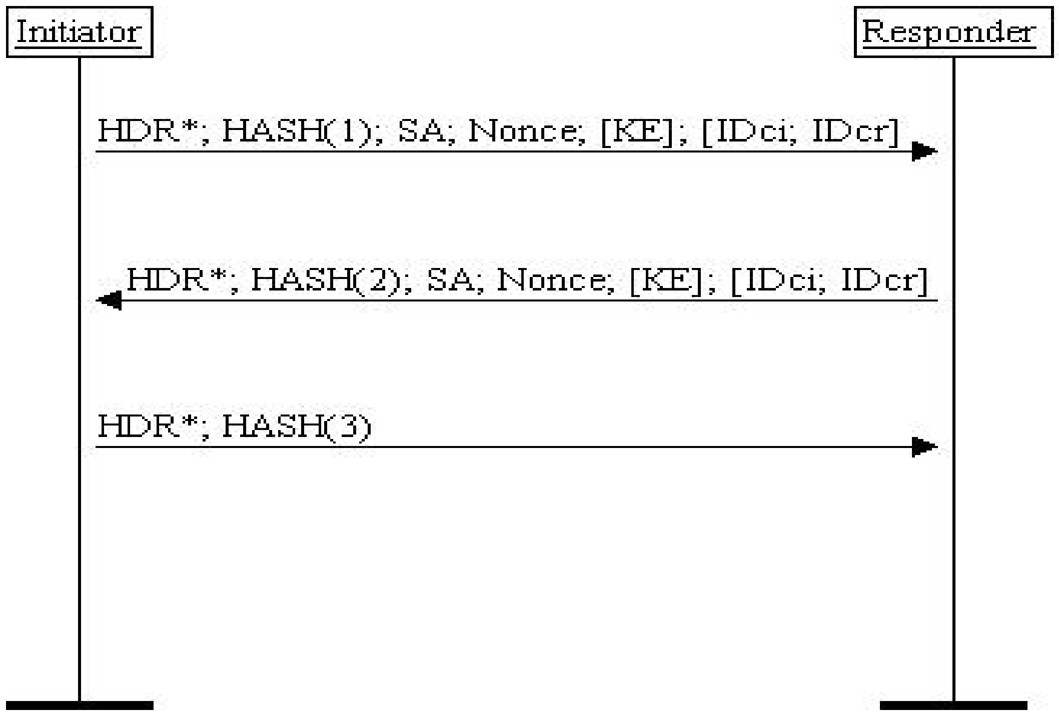

Initiator
Responder
HDR*; HASH(1); SA; Nonce; [KE]; [IDci; IDcr]
HDR*; HASH(2); SA; Nonce; [KE]; [IDci; IDcr]
HDR*; HASH(3)

Figure 7.3 – Quick Mode Exchanges

7.5.3.1.4. New Group Mode

The New Group Mode is used for negotiating a new group where to do Diffie-Hellman exchange. The MODP defines group characteristics where to calculate Diffie-Hellman. Even if New Group Mode exchange is not phase two exchange it must follow phase one exchange

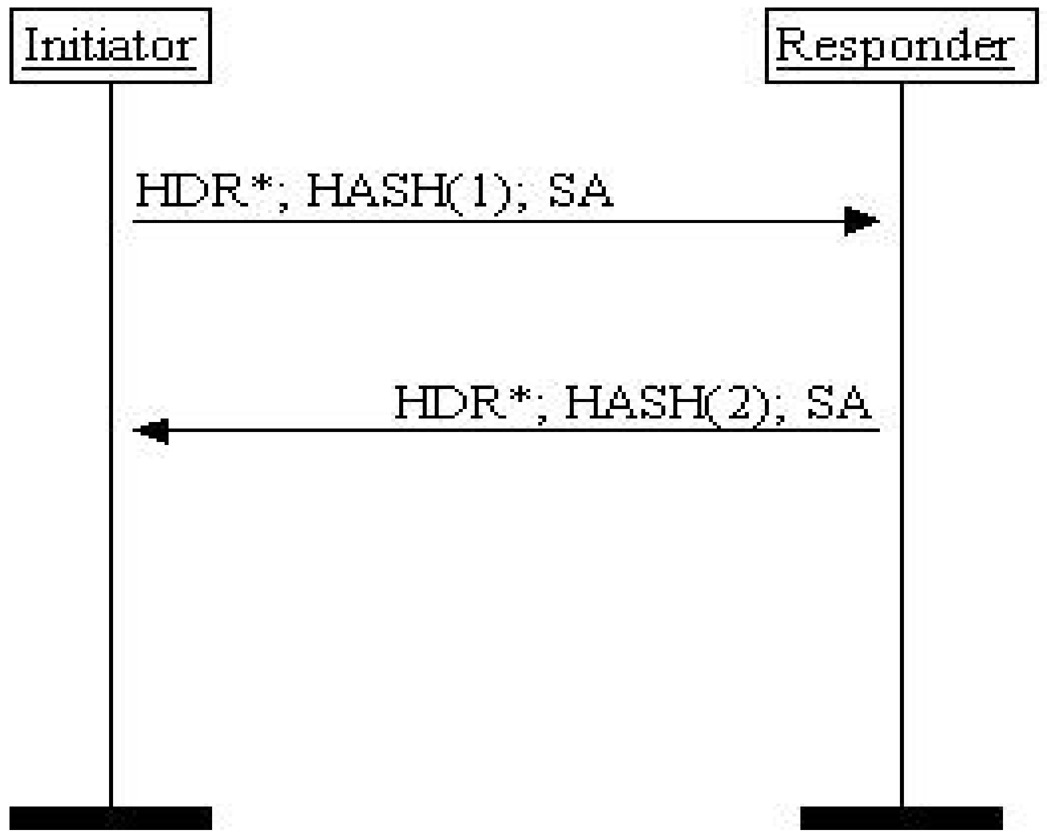


Figure 7.4 – New Group Mode Exchanges

7.5.3.2. IKE Authentication Methods

Four different authentication methods are allowed with the Main Mode and Aggressive Mode. These include the following:

7.5.3.2.1. Authentication With Digital Signatures

The exchange is authenticated by applying a negotiated signature algorithm to hashes, which are available only to the negotiating parties. A hashing algorithm is applied to almost all of the exchanged parameters. Certificates might also be provided within the exchange.

7.5.3.2.2. Authentication With Public Key Encryption

The exchange is authenticated by encrypting the identities and nonce with the other party's public key and then examining the hash sent by the other party. A right hash value proves that the other party can decrypt a data encrypted with its public key. The public keys must be provided somehow before hand.

The usage of public key encryption adds security to the key exchange, since an attacker would have to break the Diffie-Hellman exchange and RSA encryption. An identity is protected also with Aggressive Mode.

The authentication with public key encryption is computing wise relatively expensive, two public key encryption and decryption keys needed by each party.

Then each party can construct both sides of the exchange so there is no proof that the conversation ever took place.

7.5.3.2.3. Authentication With a Revised Mode of Public Key Encryption

The idea of revised mode of public key encryption is to take significant advantages of the previously described authentication and replace some of the costly public key encryptions with symmetric encryptions. Certificates might be provided also within exchange

7.5.3.2.4. Authentication With a Pre-shared Key

A key shared by secure out-of-band mechanism may also be used to authenticate parties. This authentication limits identifying methods in the Main Mode to an IP address. The authentication with a pre-shared key is the only authentication method, which is mandatory in IKE.

7.5.3.3. IKE Security Considerations

7.5.3.3.1. Protection From Attacks

ISAKMP sets requirements for its key exchange components and authentication. These requirements guard against protocol targeted attacks.

Man In The Middle

Man-In-The-Middle attack is a situation where a bad guy sits between communicating parties (A and B) on the network and intercepts traffic. The man in the middle acts as B to A and as A to B and relays traffic between them. The man in the middle could also modify, delete or insert traffic.The linking of ISAKMP exchanges is designed to prevent insertion of messages. The deletion of messages will cancel the creation of security association so partial security association will not be created. Strong authentication of the parties prevents the risk of establishing a SA with other than intended party.

Denial Of Service

Denial of Service attack is where an user can set the system unusable for legitimate users by using the system's resources. Computers on a public network are vulnerable to denial of service attacks. A cookie pair at the ISAKMP header is used to protect computing resources without spending a lot of own resources to drop bogus messages before computing intensive public key operations. Also aggressive garbage state collection should be implemented to discard protocol state information, which are created for started bogus exchanges. Absolute denial of service protection is impossible to create, but the design of the ISAKMP makes situation easier to handle.

Replay / Rejection

Replay or Reflection attack is a situation when a third party records network traffic and replays it. ISAKMP sets requirement for cookies to include a time variable material which eases detection of replay.

Connection Hijacking

The connection hijacking is an attack where a third party jumps in the middle of transaction and steals the connection. IKE is protected from the connection hijacking by linking the authentication, key exchange, and security association exchanges. The linking of exchanges prevents a third party attacker to jump in after authentication and act as one of the authenticated party during key exchange or security association exchange.

7.5.4. Perfect Forward Secrecy

Perfect Forward Secrecy (PFS) refers to the notion that compromise of a single key will permit access to only data protected by a single key. For PFS to exist the key used to protect transmission of data must not be used to derive any additional keys, and if the key used to protect transmission of data was derived from some other keying material, that material must not be used to derive any more keys.

To provide Perfect Forward Secrecy both parties must:

- Use the Main Mode to protect identities, when establishing ISAKMP security association.
- Use the Quick Mode to negotiate security associations.
- Delete ISAKMP security associations after each Quick Mode exchange pairs to force a creation of the new ISAKMP security association.

When the ISAKMP security associations are deleted a new Diffie-Hellman key generation from the new keying material will be done and bindings to the old keys are totally lost, thus, preserving Perfect Forward Secrecy.

7.6. Remote Authentication Dial In User Service (RADIUS)

7.6.1. Introduction

The RADIUS Accounting protocol is a client/server protocol that consists of two message types:

- Accounting–Request
- Accounting-Response

PacketCable network elements that generate Event Messages are RADIUS clients that send Accounting-Request messages to the RKS. The RKS is a RADIUS server that sends Accounting-Response messages back to the PacketCable network elements indicating that it has successfully received and stored the Event Message. Although

PacketCable 1.0 specifies RADIUS as the transport protocol, alternate transport protocols may be supported in future PacketCable releases. The RADIUS messages are transported over UDP, which does not guarantee reliable delivery of messages, hence the request/response nature of the protocol.

7.6.2. Event Message Structure

An Event Message contains a header followed by attributes. The header is required on every Event Message. The attributes will vary based on the type of service the Event Message is describing. Example information contained in the header includes:

- version of Event
- message structure
- timestamp indicating when the trigger event occurred
- Billing Correlation ID used to associate multiple Event Messages with a single service.

Example information contained in attributes includes:

- Called Party Number
- Calling Party Number
- Trunk Group ID

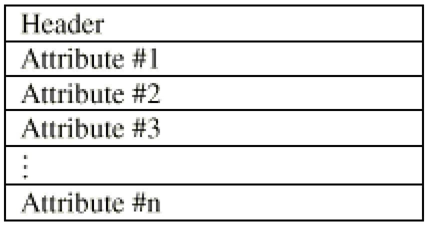


Figure 7.5 – RADIUS Packet

Each RADIUS message starts with the standard RADIUS header

Field Name	Semantics	Field Length

Code	Accounting-Request = 4
	Accounting-Response = 5
		1 byte

Identifier	Used to match accounting-request and accounting-response messages.	1 byte
Length	Total length of RADIUS message. min value = 20, max value = 4096	2 bytes
Authenticator	value = 0 as per PacketCable Security Specification [5].	16 bytes

Figure 7.6 – RADIUS Header

The standard RADIUS Acct_Status_Type attribute follows the RADIUS Message Header in every Accounting-Request message. This attribute indicates the type of this RADIUS Accounting-Request and is specific to the use of RADIUS as the transport protocol.

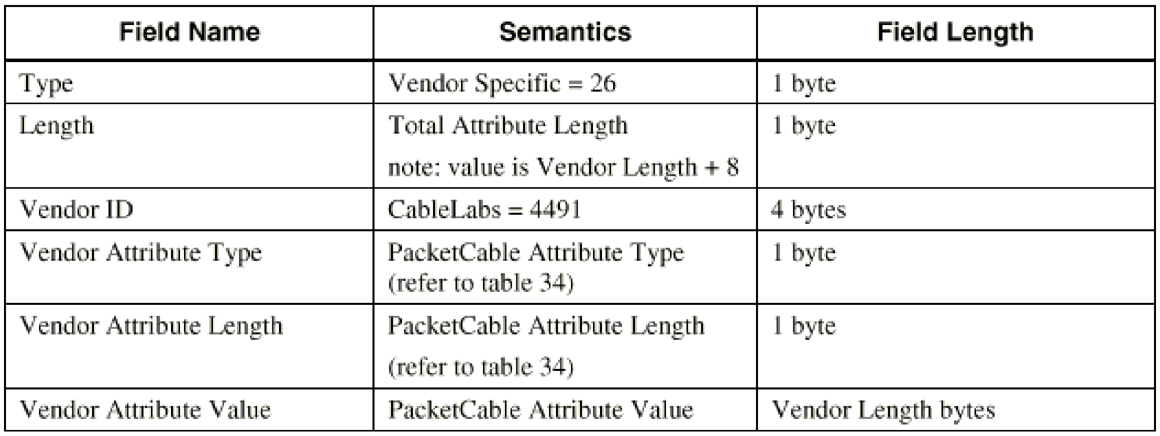
An Acct-Status-Type value of Interim-Update is used to represent PacketCable Event Messages. This improves interoperability with existing RADIUS server implementation. The Acct_Status_Type attribute is the only standard RADIUS attribute used by PacketCable.

Type	Length	Value
40	6 bytes	Interim-Update = 3

Figure 7.7 – RADIUS Accounting Status Type

PacketCable attributes are encoded in the RADIUS Vendor Specific Attributes (VSA) structure. The Vendor-Specific attribute includes a field to identify the vendor and the Internet Assigned Number Authority (IANA) has assigned PacketCable an SMI Network Management Private Enterprise Number of 4491 for the encoding of these attributes.

The RKS server ignores Event Messages where the PacketCable "Event Message type" is unidentified. The RKS server also ignores PacketCable event attributes where the event attribute type is unidentified.



Field Name	Semantics	Field Length
Type	Vendor Specific = 26	1 byte
Length	Total Attribute Length note: value is Vendor Length + 8	1 byte
Vendor ID	CableLabs = 4491	4 bytes
Vendor Attribute Type	PacketCable Attribute Type (refer to table 34)	1 byte
Vendor Attribute Length	PacketCable Attribute Length (refer to table 34)	1 byte
Vendor Attribute Value	PacketCable Attribute Value	Vendor Length bytes

Figure 7.8 – RADIUS Vendor Specific Attributes Structure

7.7. Internet Key Exchange in Record Keeping Server

The communication between the RKS and other entities is first secured using the Internet Key Exchange. There are two phases in IKE through which a secure session is established.

The Record Keeping Server (RKS) utilizes the following modes of the Internet Key Exchange (IKE):

- Phase1 – Aggressive Mode;
- Phase2 – Quick Mode.

Each of this mode requires an exchange of a certain set of messages resulting in the negotiation of a certain set of security associations, cookies and keys to protect further exchange of data. The implementation details of each of the modes is as follows:

7.7.1. Aggressive Exchange

Aggressive Exchange consists of three massages. The exchange is initiated by the Call Management Server (CMS), the Media Gateway Controller (MGC) or the Cable Modem TermiNation System (CMTS). The first message is supposed to consist of a generic header, security associations, initiator key exchanges, initiator nonce and the initiator's identification.

The security associations consist of a number of proposals, which further consist of a number of transforms. These proposals present a number of options for the security association to be negotiated with the Record Keeping Server. These security associations determine the type of the encryption algorithm, digest algorithm, the key lifetime, type of the key lifetime and various other attributes needed to protect the Phase 2 exchanges.

The Record Keeping Server is supposed to save all the initiator information and select one of the most suitable proposal from the available choices. The currently implemented version supports DES and MD5 as the available encryption and digest algorithms, respectively. The other attributes can be changed to allow various permutations.

The Record Keeping Server receives this first message and parses it. It retrieves the security associations passed into the message and compares them with its own capabilities. If no match is found, it simply send a PHASE1_NO_MATCH_FOUND error and terminates the session. If, on the other hand, a match is found, then the Record Keeping Server saves the nonce, the matched security association and the client ID. The Record Keeping Server generates its own nonce and KE values. It maintains a database log of all the trusted clients that can communicate with it. For each trusted client, the Record Keeping Server maintains a pre-shared secret key in this database. Once the Record Keeping Server receives the first message of phase one, it retrieves the key for the client. It generates the key SKEYID:

SKEYID=prf(pre-shared key, Initiator Nonce | Responder Nonce)

The prf is a function that takes as input a key and data to be encrypted. It then calculates digest on that data using MD5 and then encrypts it with the given key to generate a signed digest.

The Record Keeping Server then generates Hash_r:

Hash_r=prf(SKEY_ID,dh values | Responder Cookie | Initiator Cookie | Security Assocation | Identity of Responder)

The second message sent by the Record Keeping Server includes the generic header, the security associations with the selected proposal, the server’s key exchange, the server’s nonce, the server’s identification and the signature generated by the server. The receiving entity then parses the message, computes a corresponding signature and compares it with the received signature. After the authentication is successful, it compiles the third message and sends it to the Record Keeping server.

The third message consists of the generic header and the other entity’s signature. In this case the payload, that is everything else than the generic header, is encrypted according the algorithm agreed upon in the first two messages. The used to encrypt this message is the pre-shared key between the client and the Record Keeping Server.

The Record Keeping Server on receiving the third message authenticates it. The Record Keeping Server decrypts this message and retreives the Hash_i generated by the client. The Hash_i is given by the follwing formula:

> Hash_I=prf(SKEYID,dh values | Initiator Cookie | Responder Cookie | Security Association | Identity of Initiator)

The Record Keeping Server generates the same Hash_I using the values in the third message of phase onr. It then compares the two signatures to ensure that the third message received is correct and without error. If an error is found, it simply returns an error message to the client and terminates the session.

After this each side computes a set of keys from the exchanged information, which are to be employed in the second phase. The keys generated are given below:

> SKEYID_d=prf(SKEYID, dh values | Initiator Cookie | Responder Cookie | 0)

> SKEYID_a=prf(SKEYID,SKEYID_d | dh values | Initiator Cookie | Responder Cookie | 1)

> SKEYID_e=prf(SKEYID,SKEYID_a | dh values | Initiator Cookie | Responder Cookie | 1)

7.7.2. Quick Mode

Quick mode follows the Aggressive mode and is the only supported mode for the second phase. It consists of three messages which are employed to negotiate the security associations for a protocol following the IKE exchanges, like IPSec.

Either the Call Management Server (CMS), or the Cable Modem Termination System (CMTS) or the Media Gateway Controller (MGC) sends the first message. This message consists of the generic header along with a Hash value computed using the information exchanged in the first phase, the security associations with a number of proposals, the initiator's nonce and the identification information.

The Record Keeping Server again performs an authentication check and compares the sent Hash with its own computed Hash value.

The Hash1 sent in first message of phase two is computes as follows:

> HASH(1) = prf(SKEYID_a, message ID | Security Association | Initiator Nonce)

After successful comparison of Hash1 generated on both sides, the Record Keeping Server then generates the second message of phase two. The second message consists of the generic header, nonce, security associations selected from the offered set of choices and the second HASH value computed from a different set of the first phase values. Hash2 can be computed by the following formula:

> HASH(2) = prf(SKEYID_a, message ID | Security Association | Responder Nonce)

This is then verified on the other end, which generates the third message on successful authentication.

The third message received by the record keeping server consists of the third Hash (Hash3) value which completes the phase 2. The attributes and information negotiated in these exchanges are to be used to protect further communication between the record keeping server and the other entity.

Again, the Record Keeping Server retrieves Hash3 from the third message of phase2 and calculates its own Hash3 with the required values. Hash3 is calculated by the following formula:

> HASH(3) = prf(SKEYID_a, 0 | message ID | Initiator Nonce | Responder Nonce)

Finally, a Master Key is generated for that particular session with which further communication regarding that session will take place. This Key is given by the following formula:

> Final Key=prf(SKEYID_d, Protocol | SPISize | Initiator Nonce | Responder Nonce)

7.8. RADIUS Messages in Record Keeping Server

The client transmits a radius event message to the RKS. This message is authenticated using a signed Hash. The signature is computed using the Final Key generated above. The RKS logs that event message and accordingly sends an acknowledgement to the client.

When an RKS receives and successfully records all PacketCable Event Messages in a RADIUS Accounting-Request message, it sends an Accounting-Response message to the client. The PacketCable network element continues resending the Accounting-Request until it receives an acknowledgement from an RKS or the message expires from its cache. The RADIUS server does not transmit any Accounting-Response reply if it fails to successfully record the Event Message.

The PacketCable event messages supported in this implementation are listed below:

- Signaling_Start
- Signaling_Stop
- Call_Answer
- Call_Disconnect

7.8.1. Signaling_Start

This Event Message indicates the time at which signaling starts. The originating CMS or MGC issues this Event Message for any given call. The originating CMS or MGC that issues this Event Message must issue the corresponding Signaling_Stop Event Message. The terminating CMS or MGC may issue this Event Message. If the terminating CMS or MGC issues this Event Message, then that terminating CMS or MGC MUST also issue the corresponding Signaling_Stop Event Message. The CMS or MGC must timestamp this message.

Attribute Name	Required or Optional	Comment
[Event Message Header] (see Table 32)	R	none.
Direction_indicator	R	none.
MTA_Endpoint_Name	R	This attribute if required when the CMS generates this message. This attribute is NOT required when the MGC generates this message.
Calling_Party_Number	R	none.
Called_Party_Number	R	none.
Carrier_Identification_Code	O	This attribute MUST be included when the MGC generates this message.
Trunk_Group_ID	O	This attribute MUST be included when the MGC generates this message.

Figure 7.9 – RADIUS Signaling Start Event Message

7.8.2. Signaling_Stop

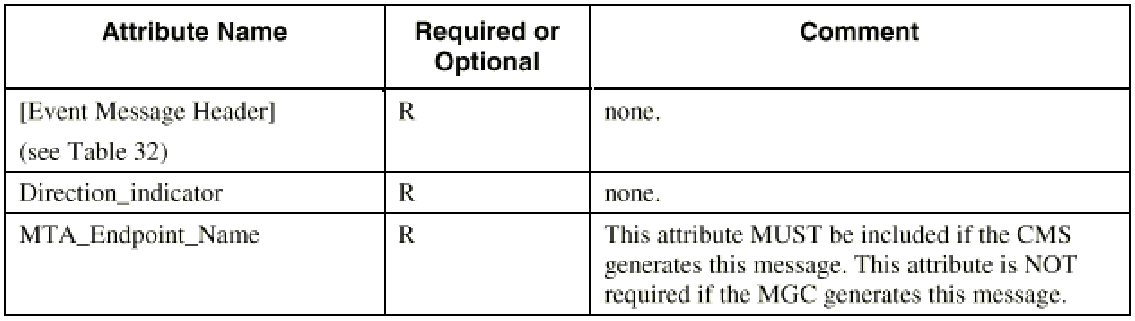
This Event Message indicates the time at which signaling terminates. The originating CMS or MGC that issues the corresponding Signaling_Start Event Message issues this Signaling_Stop Event Message. If the terminating CMS or MGC issues a corresponding Signaling_Start Event Message, then that terminating CMS or MGC must also issue this corresponding Signaling_Stop Event Message.The CMS must timestamp this message.

Attribute Name	Required or Optional	Comment
[Event Message Header] (see Table 32)	R	none.
Direction_indicator	R	none.
MTA_Endpoint_Name	R	This attribute MUST be included if the CMS generates this message. This attribute is NOT required if the MGC generates this message.

Figure 7.10 – RADIUS Signaling Stop Event Message

7.8.3. Call_Answer

This Event Message indicates that the media connection is open because answer has occurred. The terminating CMS or MGC generates this Event Message. The originating CMS or MGC may generate this Event Message. The CMS MUST timestamp this message.

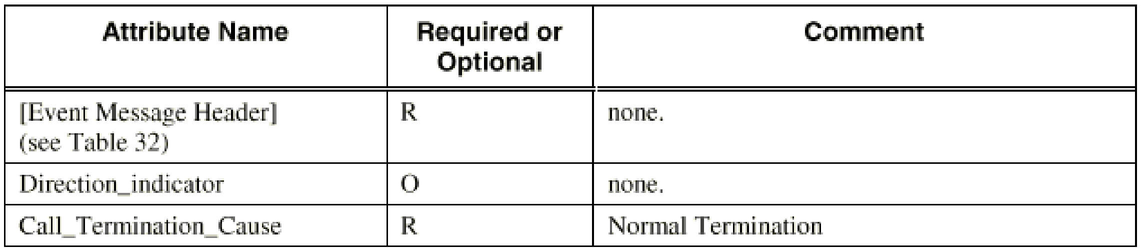


Attribute Name	Required or Optional	Comment
[Event Message Header] (see Table 32)	R	none.
Direction_indicator	R	none.
MTA_Endpoint_Name	R	This attribute MUST be included if the CMS generates this message. This attribute is NOT required if the MGC generates this message.

Figure 7.11 – RADIUS Call Answer Event Message

7.8.4. Call_Disconnect

This Event Message indicates the time at which the media connection is closed because the calling party has terminated the call by going on-hook, or that the destination party has gone on-hook and the called-party's call-continuation timer has expired. This message is be issued by the first party, either terminating or originating, to detect call termination. The CMS MUST timestamp this message.

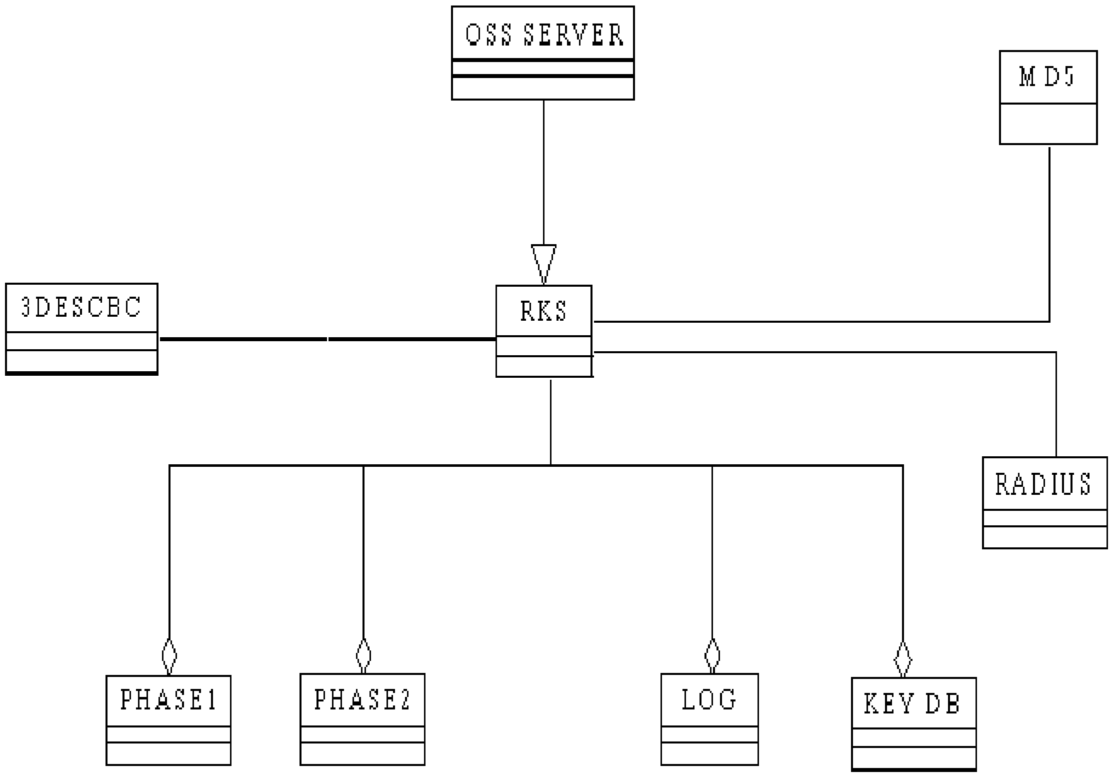


Attribute Name	Required or Optional	Comment
[Event Message Header] (see Table 32)	R	none.
Direction_indicator	O	none.
Call_Termination_Cause	R	Normal Termination

Figure 7.12 – RADIUS Call Disconnect Event Message

7.9. UML Design

7.9.1. Class Diagram



7.9.2. Classes

RKS : This is the record keeping server class

MD5 : This is the MD5 digest class which has a one to one relationship with the RKS

3DESCBC : This is the three DES CBC class which has a one to one relationship with the RKS

RADIUS : This is a RADIUS class which has a one to tone relationship with the RKS.

PHASE1 : This is the phase 1 class which is encapsulated within the RKS and consists of all the phase1 functionality
PHASE2 : This is the phase 2 class which is encapsulated within the RKS and consists of all the phase1 functionality

LOG : This is the log class for maintaining all logging functionality

KEY DB : This is the key data base that consists of the keys against the client identification information.

7.9.3. Sequence Diagram

The sequence diagram is attached in Appendix B.

CHAPTER 8
TELEPHONY SYSLOG SERVER

8.1. SYSLOG Overview

The SYSLOG server is an OSS back office network element used to collect events such as traps and errors from an Multimedia terminal Adapter. This server keeps a log of the traps that are send to it by the Multimedia Terminal Adapter so that they can be properly addressed at a later stage. The trap message is a message generated asynchronously by the MTA to notify the server of a problem aspect. Multimedia Terminal Adapter sends notification that provisioning has completed to the SYSLOG server via UDP. The only interface that the SYSLOG server has is with the Multimedia Terminal Adapter, which sends the trap messages

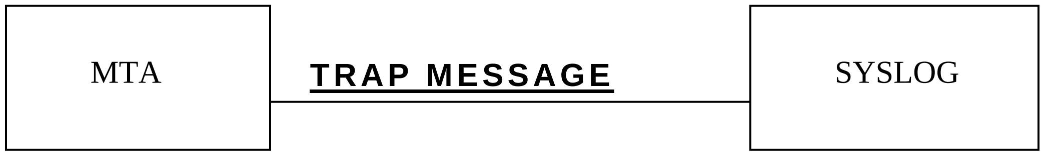


Figure 8.1 - MTA-SYSLOG Interface

8.2. Message Format

The message send by the Multimedia Terminal Adapter(MTA) to the SYSLOG server is send in the following format

<level> MTA[vendor] :<event Id> text

level – ASCII presentation of the event priority. The resulted level has the range between 128 and 135.The digit send to the SYSLOG server represents the priority of the event which has occurred.

Vendor – This field specifies the vendor name for the vendor-specific SYSLOG messages The vendor name is **PACKET CABLE** for the packet cable trap messages that are send to the SYSLOG server.

Event-Id – This number uniquely identifies the type of event, which is being sent to the SYSLOG server by the MTA. This number is the same number which is stored in the Management Information Base of the Multimedia Terminal Adapter. In the Management Information Base this number is specified in the MtaDevEvId object in pktcDevEventTable.

text – This is the textual description of the error message that is being sent to the SYSLOG server. This string MUST have the textual description as defined in the MtaDevEventText in the Management Information Base of the Multimedia Terminal Adapter.

Example: SYSLOG event for AC power failure in the MTA

<132> MTA [PACKET CABLE]: <435> AC Power Fail

The Multimedia Terminal Adapter stores information about the various elements in the Management Information Base.

8.3. Management Information Base (MIB)

Management information bases (MIBs) are a collection of definitions, which define the properties of the managed object within the device to be managed. Every managed device keeps a database of values for each of the definitions written in the MIB. The MIB can be thought of as a information ware house. MIB is simply an abstraction like "Database" which can be applied to mean all data, or any portion there of , associated with the network.

8.3.1. Features of MIB

- Objects are uniquely named.
- Abstract structure of the MIB is universal.
- Allow for private extensions
- Object must be general and not too device dependant
- Objects can not be easily derivable from their objects

8.3.2. Structure of MIB

The object set is arranged in a tree structured fashion, similar in many ways to a disk directory Structure of files. The top level branch begins with the ISO “internet" directory, which contains four main branches: The "mgmt" branch contains the standard objects usually supported (at least in part) by all network devices. The “private" branch contains those extended objects defined by network equipment vendors; the "experimental" and "directory" branches, also defined within the "internet" root directory, are usually devoid of any meaningful data or objects. The "tree" structure described above is an integral part of the standard MIBS, however the most pertinent parts of the tree are the "leaf" objects of the tree that provide actual management data regarding the device. Generally, leaf objects can be partitioned into two similar but slightly different types that reflect the organization of the tree structure.

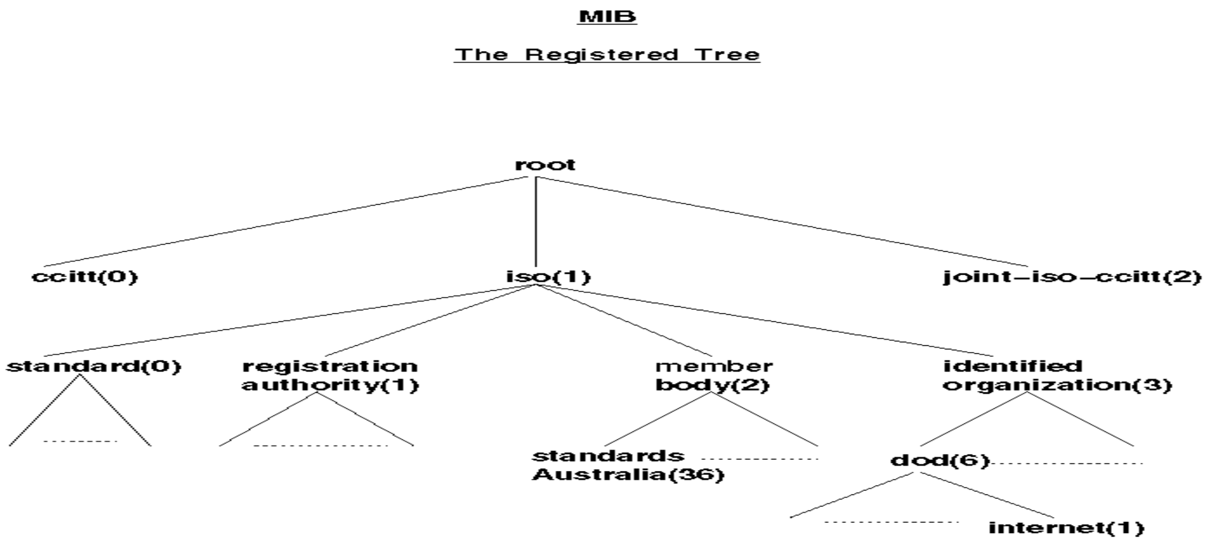


Figure 8.2 – MIB Tree

8.3.3. Categories of MIB Objects

- Discrete objects
- Table objects

8.3.3.1. Discrete Objects

Discrete objects contain one precise piece of management data. Discrete objects often represent summary values for a device, particularly useful for scanning information from the network for the purposes of comparing network device performance. These objects are often distinguished from "Table" items (below) by adding a ".0" (dot-zero) extension to their names. (If the ".0" extension is omitted from a leaf object name, it is almost always implied.

8.3.3.2. Table MIB Objects

Table objects contain multiple pieces of management data. The tables are special types of objects that allow parallel arrays or information to be supported These objects are distinguished from Discrete" items by requiring a "." (dot) extension to their names that uniquely distinguishes the particular value being referenced. The "." (dot) extension is referred to in some literature as the "instance" number of an object. In the case of "Discrete" objects, this instance number will be zero. In the case of "Table" objects, this instance number will be the index into the table.

8.3.4. MIB Object Types

There are several primitive types that can be assigned to the MIB objects. These types are classified as follows:

Text Type : This type that can contain arbitrary textual information to a maximum of 255 characters. The text must contain only printable characters.

Counter Type : This type is a numeric value that can only increase. This is the most common type of object in the standard MIB. Counters roll over at their maximum value, and can never be less than zero.

Gauge Type : This type is a numeric value that can increase or decrease.

Integer Type : The Integer type can contain positive or negative values. This value is usually supplanted by "Counter" or "Gauge" type values, but is sometimes expressed in "private" MIBs of vendor equipment.

EnumVal : This type defines an "Enumerated Value" type that associates a textual label with a numeric value. This type is quite common in the standard MIB, and includes objects, whose enumerated values are
"up(1)", "down(2)", and "testing(3)".

Time Type : This type represents an elapsed time. This time always has a resolution of one hundredth of a second, even if this resolution is not used.

Object Type : The "Object" type can contain the identifier for another object

IPAddr Type : The "IP address" type contains the IP address of a network device.

Table Type : The "Table" type is a branch object that contains table entries. This object type is always an intermediate name that contains an "Entry" directory, which in turn contains various table objects.

Branch Type : The "Branch" type is a branch object that contains additional branches, tables, or any of the discrete objects types listed above.

MIB description files are written in a particular format called ASN.1 (Abstract Syntax Notation One)

8.4. Multimedia Terminal Adapter (MTA)

This MIB provides a set of objects required for the management of DOCSIS compliant Cable Modems (CM) and Multimedia Terminal Adapter.

8.4.1. Groups in MTA MIB

This MIB is structured into seven groups:

- The docsDevBase group contains the objects needed for cable device system management.
- The docsDevNmAccessGroup provides a minimum level of access security.
- The docsDevSoftware group provides information for network-downloadable software upgrades.
- The docsDevServer group provides information about the progress of the interaction between the CM or CMTS and various provisioning servers.
- The docsDevEvent group provides control and logging for event reporting.
- The docsDevFilter group configures filters at link layer and IP layer for bridged data traffic. This group consists of a link-layer filter table, docsDevFilterLLCTable, which is used to manage the processing and forwarding of non-IP traffic; and IP packet classifier table, docsDevFilterIpTable, which is used to map classes of packets to specific policy actions; a policy table, docsDevFilterPolicyTable, which maps zero or more policy actions onto a specific packet classification, and one or more policy action tables. At this time, this MIB specifies only one policy action table, docsDevFilterTosTable, which allows the

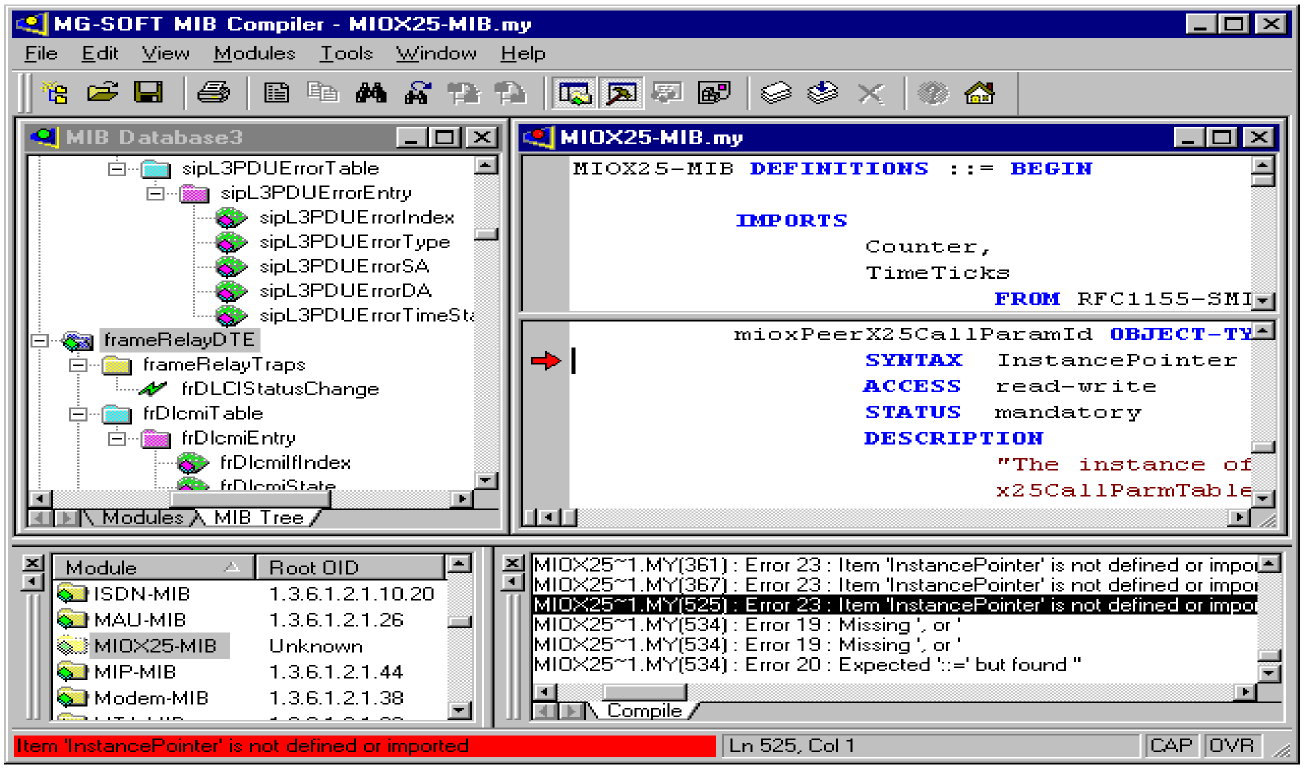
manipulation of the type of services bits in an IP packet based on matching some criteria. The working group may add additional policy types and action tables in the future.

- The docsDevCpe group provides control over which IP addresses may be used by customer premises equipment (e.g. PCs) serviced by a given cable modem. This provides anti-spoofing control at the point of origin for a large cable modem system.This group is separate from docsDevFilter primarily as this group is only implemented on the Cable Modem (CM) and MUST NOT be implemented on the Cable Modem Termination System (CMTS).

The main purpose of the syslog was concerned with the logging of events so the docsDevEvent Group was implemented. In order to obtain the tree structure of the Multimedia Terminal Adapter the following tools were used in the project.

- MIB Compiler
- MIB Builder

The Multimedia Terminal Adapter MIB file written in the Abstract Syntax Notation (ASN.1) was given as input to the MIB Compiler which checks the syntax of the file written in ASN and generates different errors according to the errors that have occurred.


MG-SOFT MIB Compiler - MIOX25-MIB.my
File Edit View Modules Tools Window Help
MIB Database3
MIOX25-MIB.my
Modules
MIB Tree
Compile
CAP
OVR

```
sipL3PDUErrorTable
  sipL3PDUErrorEntry
    sipL3PDUErrorIndex
    sipL3PDUErrorType
    sipL3PDUErrorSA
    sipL3PDUErrorDA
    sipL3PDUErrorTimeSt
frameRelayDTE
  frameRelayTraps
    frDLCIStatusChange
  frDlcmiTable
    frDlcmiEntry
      frDlcmiIfIndex
      frDlcmiState
```

```
MIOX25-MIB DEFINITIONS ::= BEGIN

        IMPORTS
                Counter,
                TimeTicks
                        FROM RFC1155-SMI
```

```
        mioxPeerX25CallParamId OBJECT-TY
                SYNTAX    InstancePointer
                ACCESS    read-write
                STATUS    mandatory
                DESCRIPTION
                          "The instance of
                          x25CallParmTable
```

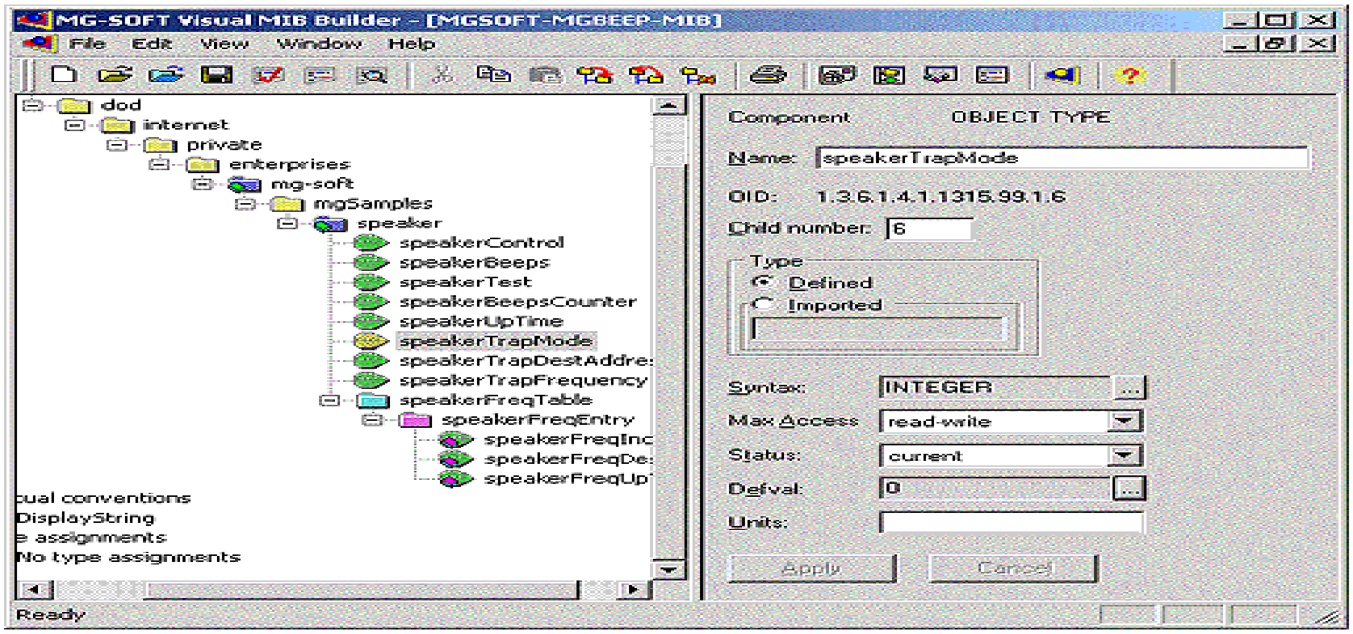
Module	Root OID
ISDN-MIB	1.3.6.1.2.1.10.20
MAU-MIB	1.3.6.1.2.1.26
MIOX25-MIB	Unknown
MIP-MIB	1.3.6.1.2.1.44
Modem-MIB	1.3.6.1.2.1.38

```
MIOX25~1.MY(361) : Error 23 : Item 'InstancePointer' is not defined or impo
MIOX25~1.MY(367) : Error 23 : Item 'InstancePointer' is not defined or impo
MIOX25~1.MY(525) : Error 23 : Item 'InstancePointer' is not defined or impo
MIOX25~1.MY(534) : Error 19 : Missing ', or '
MIOX25~1.MY(534) : Error 19 : Missing ', or '
MIOX25~1.MY(534) : Error 20 : Expected '::=' but found ''
```

```
Item 'InstancePointer' is not defined or imported    Ln 525, Col 1
```

Figure 8,3 – MIB Compiler

The other tool that was used was the MIB Builder. The file which has been compiled is given as input to the MIB builder and this tool generates the tree structure of the given MIB. The tree structure for the Multimedia Terminal Adapter generated with the help of this tool is attached in Appendix C which follows the document.

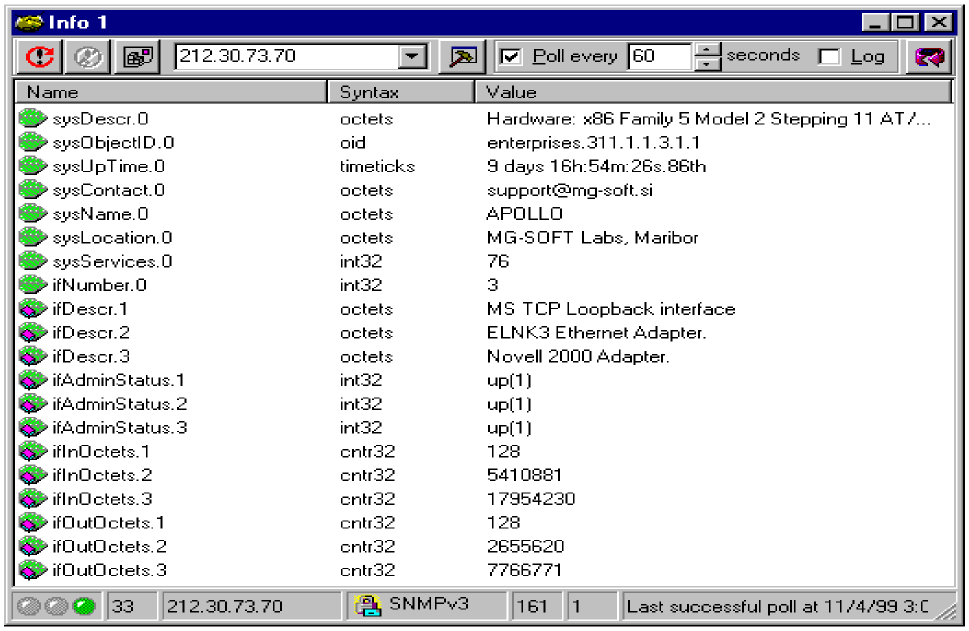


```
dod
  internet
    private
      enterprises
        mg-soft
          mgSamples
            speaker
              speakerControl
              speakerBeeps
              speakerTest
              speakerBeepsCounter
              speakerUpTime
              speakerTrapMode
              speakerTrapDestAddre
              speakerTrapFrequency
              speakerFreqTable
                speakerFreqEntry
                  speakerFreqInc
                  speakerFreqDe
                  speakerFreqUp
ual conventions
DisplayString
e assignments
No type assignments
```

Component: OBJECT TYPE
Name: speakerTrapMode
OID: 1.3.6.1.4.1.1315.99.1.6
Child number: 6
Type: (●) Defined ( ) Imported
Syntax: INTEGER
Max Access: read-write
Status: current
Defval: 0
Units:

Figure 8.4 – MIB Builder

In addition to these two tools the other tool that was used to view the static values in the MIB was the MIB Browser. With the help of this tool you can view the values in the tables of the MIB.



Name	Syntax	Value
sysDescr.0	octets	Hardware: x86 Family 5 Model 2 Stepping 11 AT/...
sysObjectID.0	oid	enterprises.311.1.1.3.1.1
sysUpTime.0	timeticks	9 days 16h:54m:26s.86th
sysContact.0	octets	support@mg-soft.si
sysName.0	octets	APOLLO
sysLocation.0	octets	MG-SOFT Labs, Maribor
sysServices.0	int32	76
ifNumber.0	int32	3
ifDescr.1	octets	MS TCP Loopback interface
ifDescr.2	octets	ELNK3 Ethernet Adapter.
ifDescr.3	octets	Novell 2000 Adapter.
ifAdminStatus.1	int32	up(1)
ifAdminStatus.2	int32	up(1)
ifAdminStatus.3	int32	up(1)
ifInOctets.1	cntr32	128
ifInOctets.2	cntr32	5410881
ifInOctets.3	cntr32	17954230
ifOutOctets.1	cntr32	128
ifOutOctets.2	cntr32	2655620
ifOutOctets.3	cntr32	7766771

33 | 212.30.73.70 | SNMPv3 | 161 | 1 | Last successful poll at 11/4/99 3:0

Figure 8.5 – MIB Browser

8.4.2. Events and Traps Group in MTA-MIB :

This group of MIB provides control facilities for reporting events through syslog, traps, and non-volatile logging.The trap messages follow a specified conventions . The definition and coding of events is vendor-specific.Trap definition in the abstract syntax notation file are defined as folows:

```
trapName NOTIFICATION-TYPE

            OBJECTS {
               ifIndex,
               eventReason,
               other useful objects
                    }
 STATUS      current

 DESCRIPTION
             "trap description"
           ::= Object Id
```

8.4.2.1. Types of Traps

There are various types of traps that can be send to the syslog server for logging. These traps are sent in response to different conditions. Some of the more generic traps are as follows:

Cold-Start: This trap is sent when the Multimedia Terminal Adapter restarts as a result of a crash or a major fault.

Warm-Start: This trap is sent when the Multimedia Terminal Adapter reinitializes itself.

Link-Down: This trap is sent to signal a failure in one of the communication links of the Multimedia Terminal Adapter.

Link-Up: This trap is sent to signal that one of the communication links has again come up.

Authentication-Failure: This signals that the authentication of Multimedia Terminal Adapter has failed.

These are some of the generic traps .In addition to these traps there are also some specific traps, which occur in response to different error conditions. The Syslog server keeps a record of all the traps that are sent to it by the Multimedia Terminal Adapter. The user can view the details of different errors generated in the Multimedia Terminal Adapter.

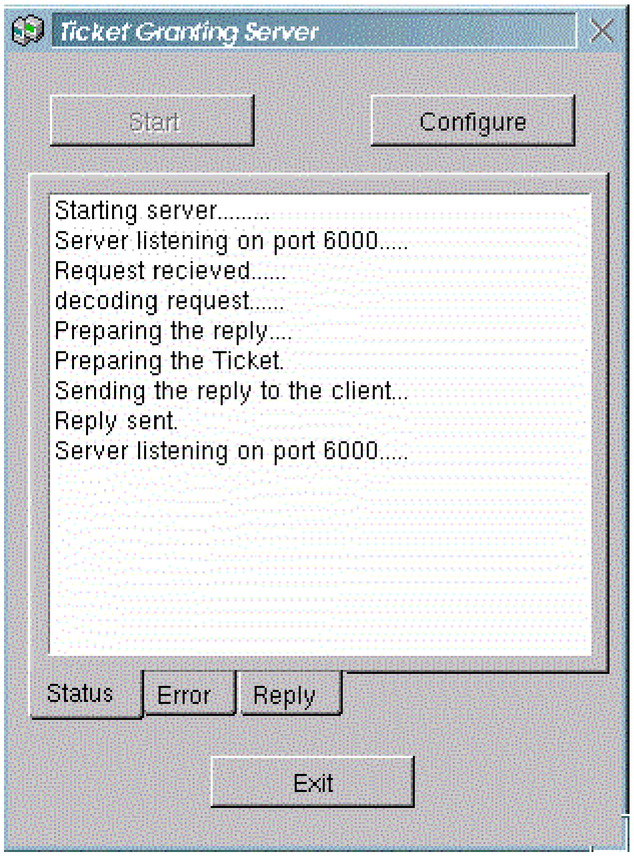
CHAPTER 9

User Manual

9.1. Ticket Granting Server

9.1.1 Server GUI

The Ticket granting server consists of a simple easy to operate interface as shown in the following figure:

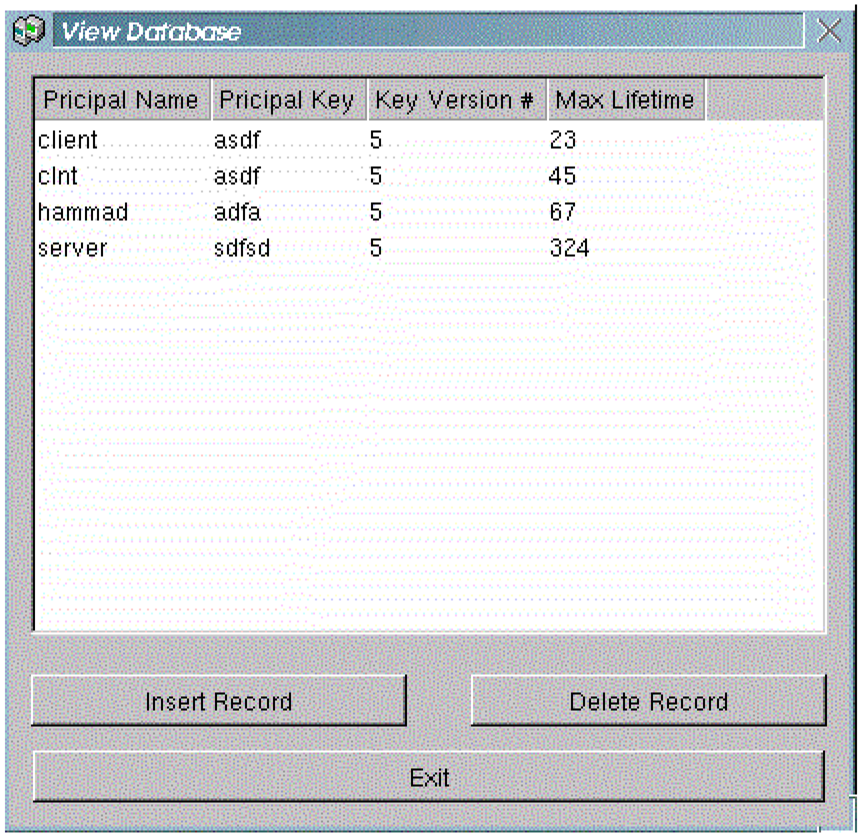


```
Starting server.........
Server listening on port 6000.....
Request recieved......
decoding request......
Preparing the reply....
Preparing the Ticket.
Sending the reply to the client...
Reply sent.
Server listening on port 6000.....
```

Figure 9.1 – TGS User Interface

The server can be initiated by clicking on the ‘Start’ button. This sends the server in the blocking call of listening to a pre-defined socket. On receipt of a message the server will display the status in the ‘Status’ tab, with the corresponding received structures and any possible errors in the ‘Reply’ tab and ‘Error’ tab, respectively.

To view the database of the ticket granting server click on the ‘Configure’ button, which will invoke the following dialog consisting of the database contents:



Pricipal Name	Pricipal Key	Key Version #	Max Lifetime
client	asdf	5	23
clnt	asdf	5	45
hammad	adfa	5	67
server	sdfsd	5	324

Figure 9.2– Database View

To insert a record in the database click on the ‘Insert Record’, which invokes the following dialog box:

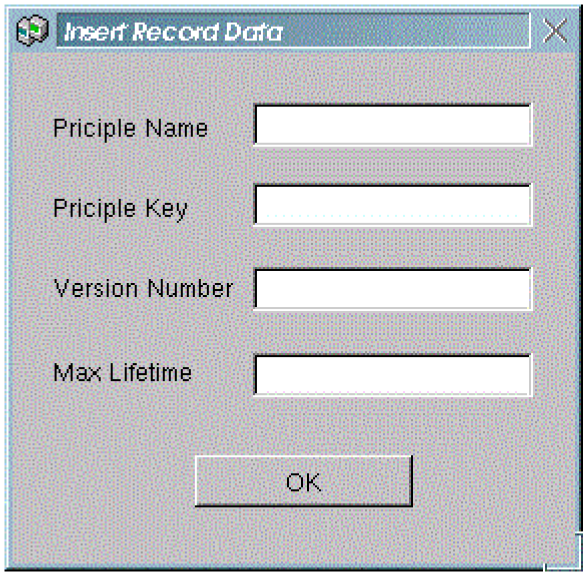


Figure 9.3– Insert Record Dialog

Here enter the attributes for the new record in the corresponding edit boxes and click ‘Ok’. All fields are required. To delete a record click on the ‘Delete Record’ button, which invokes the following dialog box:

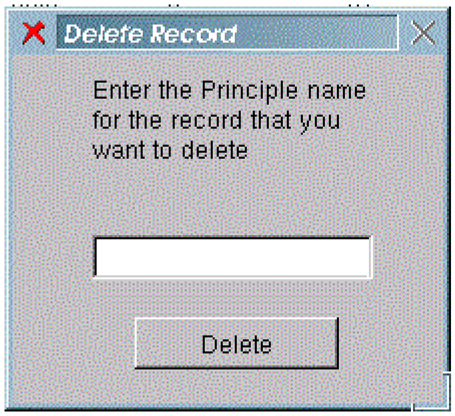
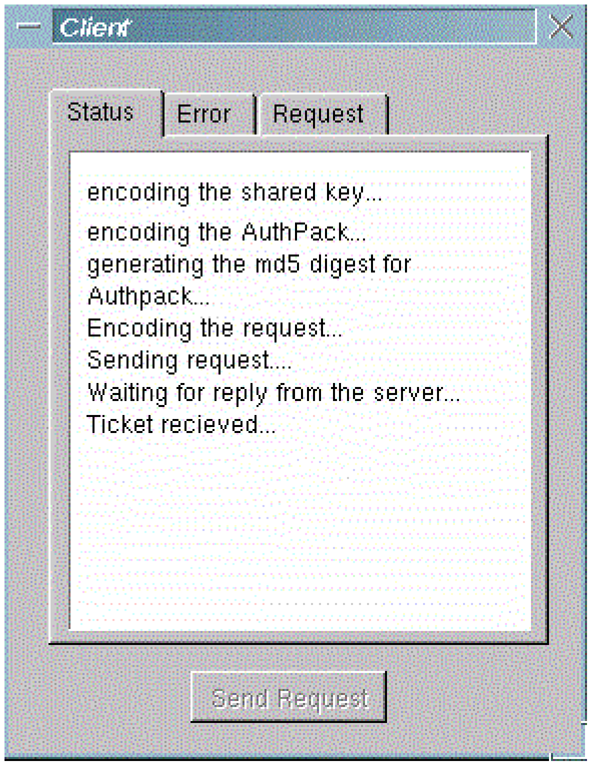

Delete Record
Enter the Principle name for the record that you want to delete
Delete

Figure 9.4– Delete Record Dialog

Here enter the principle name for the record to delete and click 'Delete'.

9.1.2. Client GUI

This consists of a simple interface, which invokes a request from the client by clicking on the 'Send Request' button, as shown in the following figure:



```
encoding the shared key...
encoding the AuthPack...
generating the md5 digest for
Authpack...
Encoding the request...
Sending request....
Waiting for reply from the server...
Ticket recieved...
```

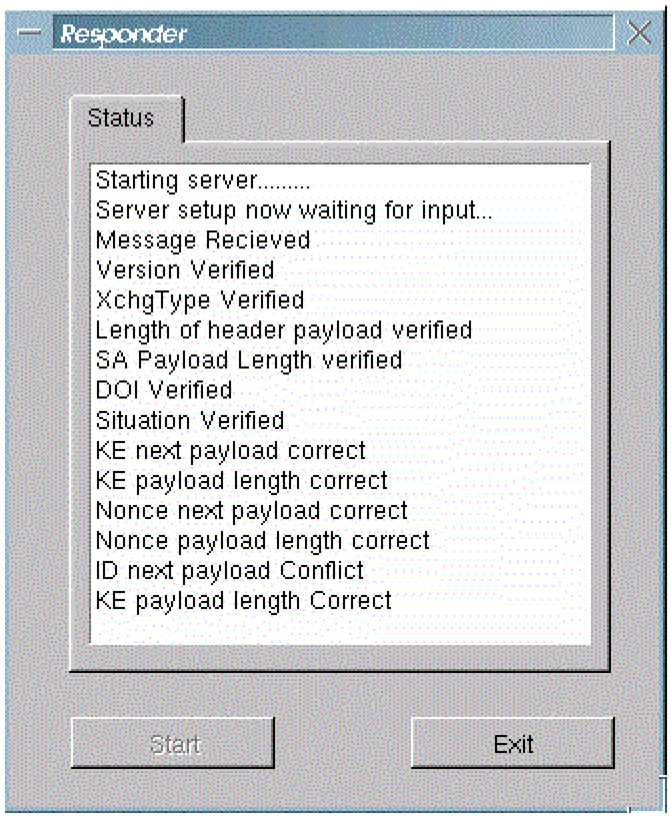
Figure 9.5– TGS Client User Interface

The status of the client is visible in the 'Status' tab, while the corresponding request and possible errors can be viewed under the 'Request' tab and the 'Error' tab, respectively.

9.2. Record Keeping Server

9.2.1. Server GUI

The record keeping server graphical user interface is as shown in the following figure:



```
Starting server.........
Server setup now waiting for input...
Message Recieved
Version Verified
XchgType Verified
Length of header payload verified
SA Payload Length verified
DOI Verified
Situation Verified
KE next payload correct
KE payload length correct
Nonce next payload correct
Nonce payload length correct
ID next payload Conflict
KE payload length Correct
```

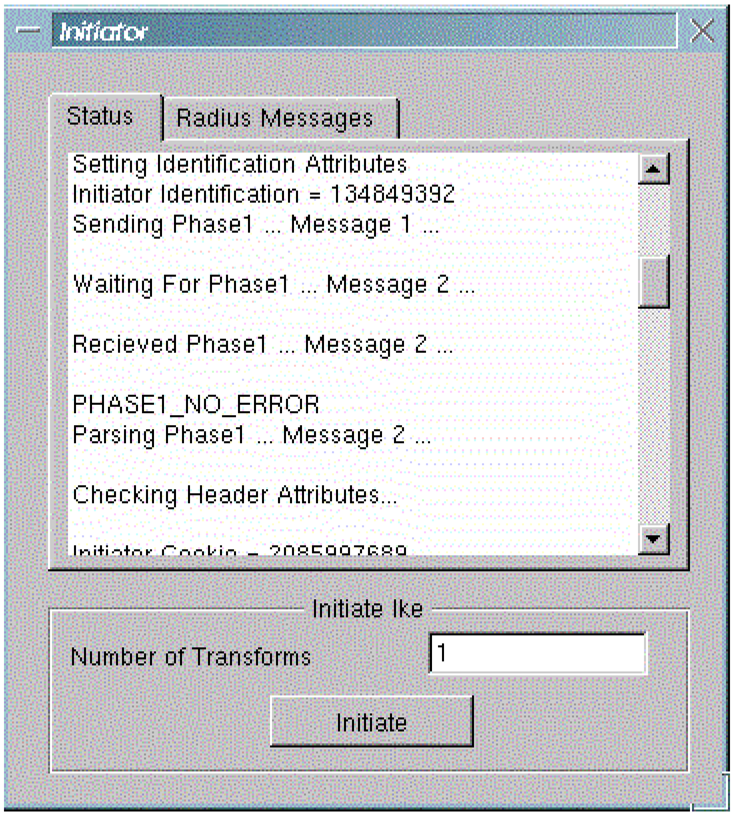
Figure 9.6 – RKS User Interface

Click on the 'Start' button to start the server, which enters the listening mode waiting for a request from the client.

The server status is visible in the 'Status' view, which displays all the messages and current status of the server.

9.2.2. Client GUI

The client end user interface is as follows:

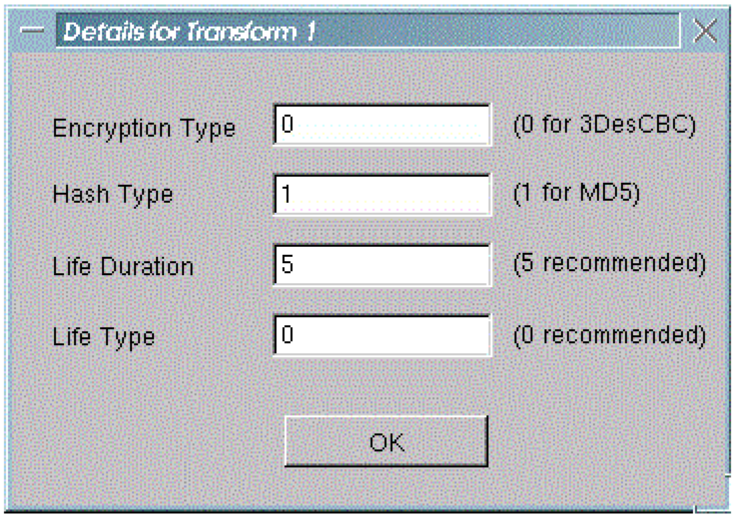


```
Setting Identification Attributes
Initiator Identification = 134849392
Sending Phase1 ... Message 1 ...

Waiting For Phase1 ... Message 2 ...

Recieved Phase1 ... Message 2 ...

PHASE1_NO_ERROR
Parsing Phase1 ... Message 2 ...

Checking Header Attributes...
```

1

Figure 9.7 – RKS Client User Interface

Enter the number of transforms to include in the first message of the internet key exchange. These number of transforms should not be less than zero and not more than five. Click on the 'Initiate' button to start an Internet Key Exchange with the server. This will invoke the following dialog.


Details for Transform 1

Encryption Type	0	(0 for 3DesCBC)
Hash Type	1	(1 for MD5)
Life Duration	5	(5 recommended)
Life Type	0	(0 recommended)

Figure 9.8 – Transform Details Dialog

Enter the attribute values in the corresponding fields and click 'Ok'. This dialog requests input depending on the number of transforms specified.

This initiates the internet key exchange, the status for which is displayed in he corresponding 'Status' tab. Click on the 'Radius' tab. This will display the following:

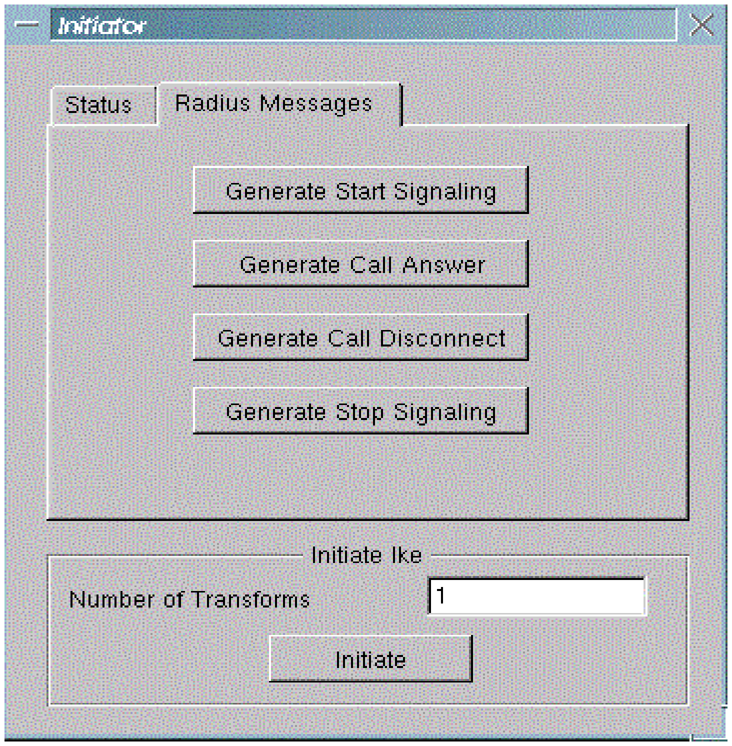


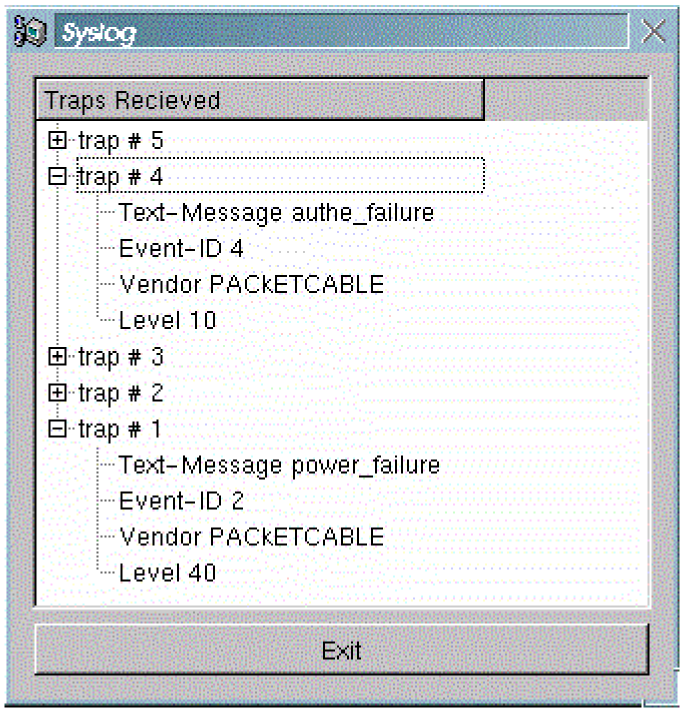
Figure 9.9 – RKS Client User Interface (RAIUS)

Click on the ‘Start Signaling’ button to send a Radius start signaling event message. Click on the ‘Call Answer’ button to send a Radius call answer event message. Click on the ‘Call Disconnect’ button to send a Radius call disconnect event message. Click on the ‘Stop Signaling’ button to send a Radius stop signaling event message.

9.3. SYSLOG Server

9.3.1. Server GUI

The server user interface is as follows:



```
trap # 5
trap # 4
    Text-Message authe_failure
    Event-ID 4
    Vendor PACkETCABLE
    Level 10
trap # 3
trap # 2
trap # 1
    Text-Message power_failure
    Event-ID 2
    Vendor PACkETCABLE
    Level 40
```

Figure 9.10 – SYSLOG User Interface

The view displays the traps captured and their details in a tree view. These can be viewed by clicking on the expand icon as shown. The traps are displayed as they are captured.

9.3.2. Client GUI

The client user interface consists of two input fields for port number and IP address of the server, and three buttons for generating and sending traps, pausing trap generation, and exiting the application. Enter the port number and IP address of the SYSLOG server before sending traps. Click ‘Send’ to invoke the trap generation. Click ‘Pause’ to pause the trap generation process. Click ‘Exit’ to quit from the client interface.

CHAPTER 10

Future Expansion

The current implementation of the key management and security servers is based on the PacketCable 1.0 specifications, which have recently been updated to incorporate a number of other features, along with enhanced security features. Thus the current implementation will provide a solid infrastructure for further development and enhancements. Moreover, the Ticket Granting Server, which currently consists of public key PKINIT, can be improved with added feature of pre-shared keys and certificates. Similarly, the basic implementation of the Record Keeping Server, which consists of all the necessary features, as prescribed in the specifications, can be improved with the incorporation of the optional features.

Conclusion

This project helped us develop a better understanding and gain practical hands-on experience of the theoretical knowledge we attained through out our under graduate course. The implementation of this project covered almost all the aspects of our degree and thus, improved our concepts of the various software courses we attended.

The software engineering principles were improved as we practiced the pure software engineering approach to design and develop the system. Unified Modeling Language was used to carry out the design process and Rational Rose was employed as the tool to achieve a sophisticated design.

The development was carried out on the Linux platform, which is comparatively less user friendly then the popular Windows, and therefore helped us develop a better understanding of the Operating Systems and helped us clear some of our doubts in this regard.

The development was carried out in C++, which helped us not only in improving our object oriented approach and implementation, but also helped us practice development of algorithms for various programming issues.

The Networking concepts were further fortified with the project, as the entire project was a networking environment with various client-server interactions involving exchange of data over the sockets using various protocols. The Abstract Syntax Notation usage helped us in obtaining a better understanding of the conceptual picture of transfer of information over the network.

The development of the graphical user interface (GUI), in Linux, for the servers was not only a new experience, but also enhanced our knowledge of graphics and the inter-relationship of the various graphical components in a user interface.

Finally, the study and development of the encryption algorithms and the digest algorithms provided us an exposure to the field of cryptology and network security, which was an outstanding experience. The study and implementation of all the protocols, including Kerberos with PKINIT extension, Internet Key Exchange and Radius, was surely an extraordinary learning experience.

APPENDICES

APPENDIX A

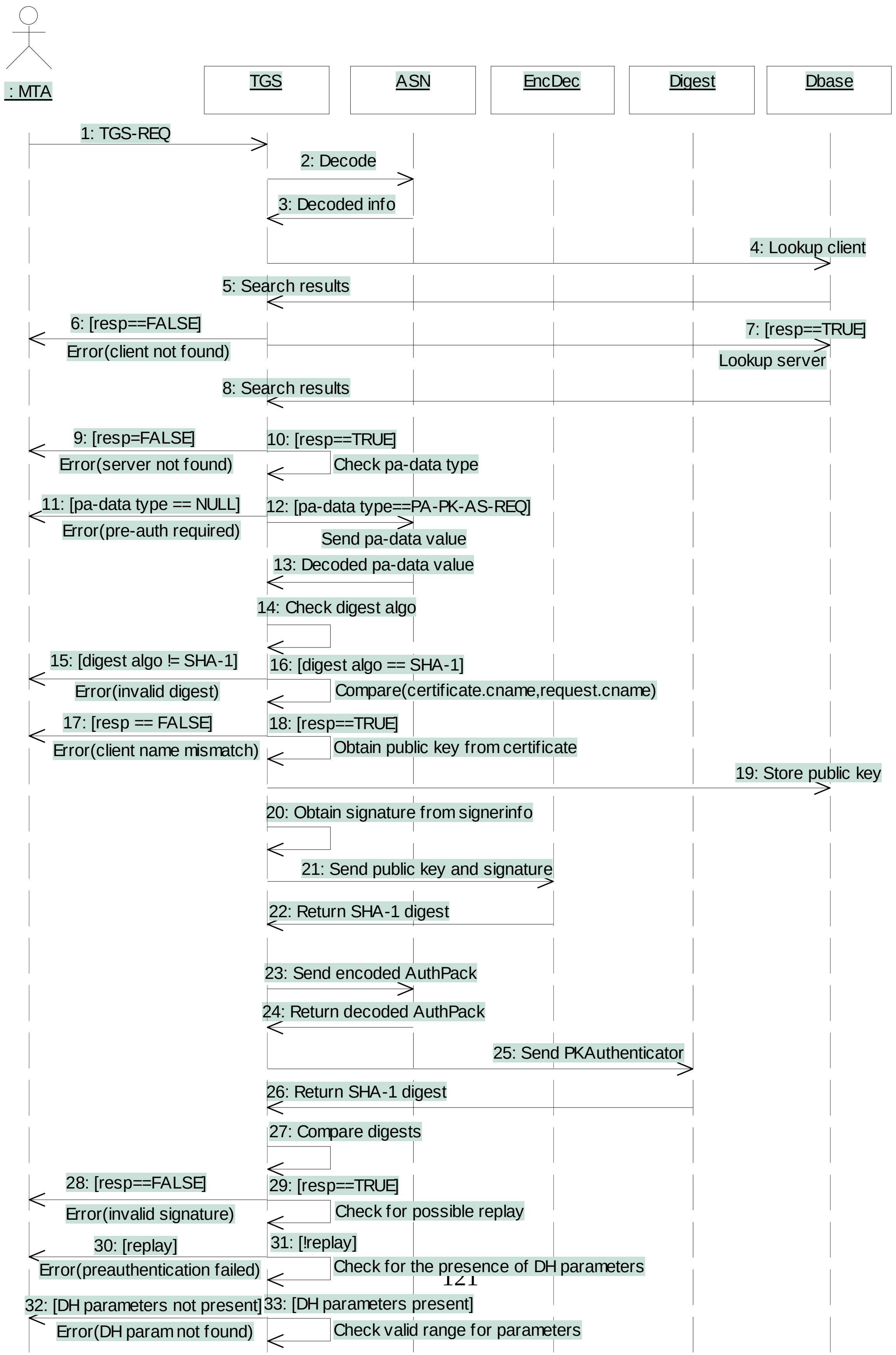
The ASN structures used for the Ticket Granting Server are listed below:

```
KDC-REQ ::= SEQUENCE {
          pvno[1] INTEGER,
          msg-type[2] INTEGER,
          padata[3] SEQUENCE OF PA-DATA OPTIONAL,
          req-body[4] KDC-REQ-BODY
          }

PA-DATA ::= SEQUENCE {
          padata-type[1] INTEGER,
          padata-value[2] OCTET STRING
          }

SignedData ::= SEQUENCE {
          digestAlgorithms DigestAlgorithmIdentifiers,
          encapContentInfo EncapsulatedContentInfo,
          certificates [0] IMPLICIT CertificateSet OPTIONAL,
          crls [1] IMPLICIT CertificateRevocationLists OPTIONAL,
          signerInfos SignerInfos
          }

DigestAlgorithmIdentifier ::= AlgorithmIdentifier

AlgorithmIdentifier ::= SEQUENCE {
          algorithm OBJECT IDENTIFIER,
          parameters ANY DEFINED BY algorithm OPTIONAL
          }

EncapsulatedContentInfo ::= SEQUENCE {
          eContentType ContentType,
          eContent [0] EXPLICIT OCTET STRING OPTIONAL
          }

AuthPack ::= SEQUENCE {
          pkAuthenticator [0] PKAuthenticator,
          clientPublicValue [1] SubjectPublicKeyInfo OPTIONAL
          }

PKAuthenticator ::= SEQUENCE {
          kdcName [0] PrincipalName,
          kdcRealm [1] Realm,
          cusec [2] INTEGER,
          ctime [3] KerberosTime,
          nonce [4] INTEGER}

SubjectPublicKeyInfo ::= SEQUENCE {
          algorithm AlgorithmIdentifier,
          subjectPublicKey BIT STRING
          -- for DH, equals public exponent (INTEGER encoded
          -- as payload of BIT STRING)
          }

SignerInfo ::= SEQUENCE {
           version CMSVersion,
           sid SignerIdentifier,
           digestAlgorithm DigestAlgorithmIdentifier,
          signatureAlgorithm SignatureAlgorithmIdentifier,
          signature SignatureValue
         }

SignerIdentifier ::= CHOICE {
          issuerAndSerialNumber IssuerAndSerialNumber,
          subjectKeyIdentifier [0] SubjectKeyIdentifier
         }

SignatureValue ::= OCTET STRING

SignatureAlgorithmIdentifier ::= AlgorithmIdentifier

KDC-REQ-BODY ::= SEQUENCE {
         cname[1] PrincipalName OPTIONAL, -- Used only in AS-REQ
         realm[2] Realm, -- Server's realm -- Also client's in AS-REQ
         sname[3] PrincipalName OPTIONAL,
         from[4] KerberosTime OPTIONAL,
         till[5] KerberosTime,
         nonce[7] INTEGER,
         etype[8] SEQUENCE OF INTEGER, -- EncryptionType
         }

KDC-REP ::= SEQUENCE {
          pvno[0] INTEGER,
          msg-type[1] INTEGER,
          padata[2] SEQUENCE OF PA-DATA OPTIONAL,
          crealm[3] Realm,
          cname[4] PrincipalName,
          ticket[5] Ticket,
          enc-part[6] EncryptedData
         }

PA-PK-AS-REP ::= CHOICE {
         dhSignedData [0] SignedData,
         encKeyPack [1] EnvelopedData,
         }

KdcDHKeyInfo ::= SEQUENCE {
         nonce [0] INTEGER,
         subjectPublicKey [2] BIT STRING
         }

Ticket ::= [APPLICATION 1] SEQUENCE {
          tkt-vno[0] INTEGER,
          realm[1] Realm,
          sname[2] PrincipalName,
          enc-part[3] EncryptedData
          }

EncTicketPart ::= [APPLICATION 3] SEQUENCE {
         key[1] EncryptionKey,
          crealm[2] Realm,
          cname[3] PrincipalName,
          transited[4] TransitedEncoding,
          authtime[5] KerberosTime,
          endtime[7] KerberosTime
         }

TransitedEncoding ::= SEQUENCE {
         tr-type[0] INTEGER, -- must be registered
         contents[1] OCTET STRING
         }

EncKDCRepPart ::= SEQUENCE {
          key[0] EncryptionKey,
          last-req[1] LastReq,
          nonce[2] INTEGER,
          authtime[5] KerberosTime,
          endtime[7] KerberosTime,
          srealm[9] Realm,
          sname[10] PrincipalName
         }

Certificate ::= SEQUENCE {
          tbsCertificate TBSCertificate,
          signatureAlgorithm AlgorithmIdentifier,
          signatureValue BIT STRING }
TBSCertificate ::= SEQUENCE {
          version [0] EXPLICIT Version DEFAULT v1,
          serialNumber CertificateSerialNumber,
          signature AlgorithmIdentifier,
         issuer Name,
         validity Validity,
         subject Name,
         subjectPublicKeyInfo SubjectPublicKeyInfo,
         issuerUniqueID [1] IMPLICIT UniqueIdentifier OPTIONAL, -- If present,
         version shall be v2 or v3
         subjectUniqueID [2] IMPLICIT UniqueIdentifier OPTIONAL, -- If
         present, version shall be v2 or v3
         extensions [3] EXPLICIT Extensions OPTIONAL -- If present, version
         shall be v3
         }
```

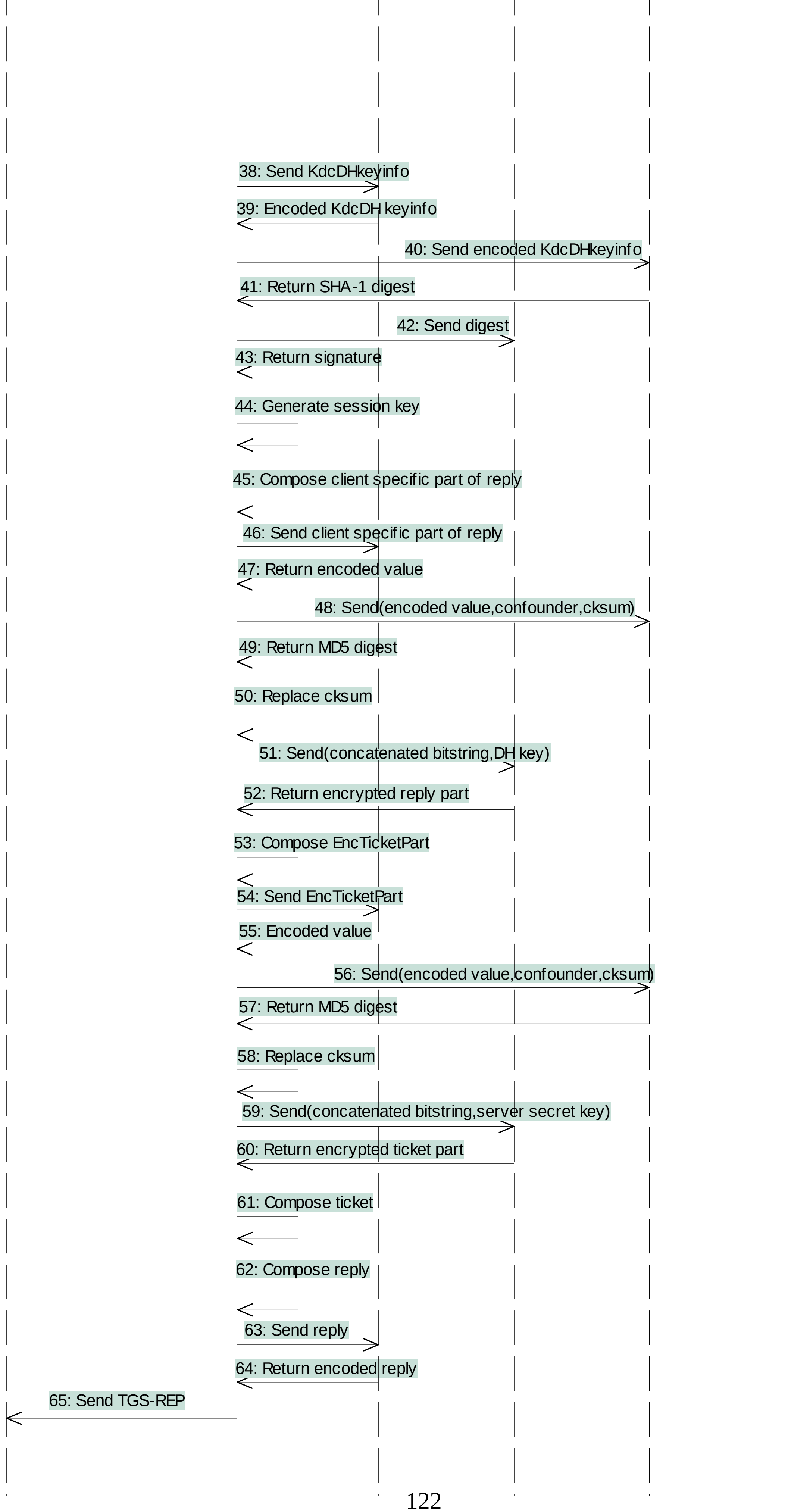
```
Version ::= INTEGER { v1(0), v2(1), v3(2) }

CertificateSerialNumber ::= INTEGER

Validity ::= SEQUENCE {
          notBefore Time,
          notAfter Time
          }

Time ::= CHOICE {
          utcTime UTCTime,
          generalTime GeneralizedTime
          }

UniqueIdentifier ::= BIT STRING

SubjectPublicKeyInfo ::= SEQUENCE {
           algorithm AlgorithmIdentifier,
           subjectPublicKey BIT STRING
          }

Extensions ::= SEQUENCE SIZE (1..MAX) OF Extension

Extension ::= SEQUENCE {
           extnID OBJECT IDENTIFIER,
           critical BOOLEAN DEFAULT FALSE,
           extnValue OCTET STRING
          }
```

APPENDIX B

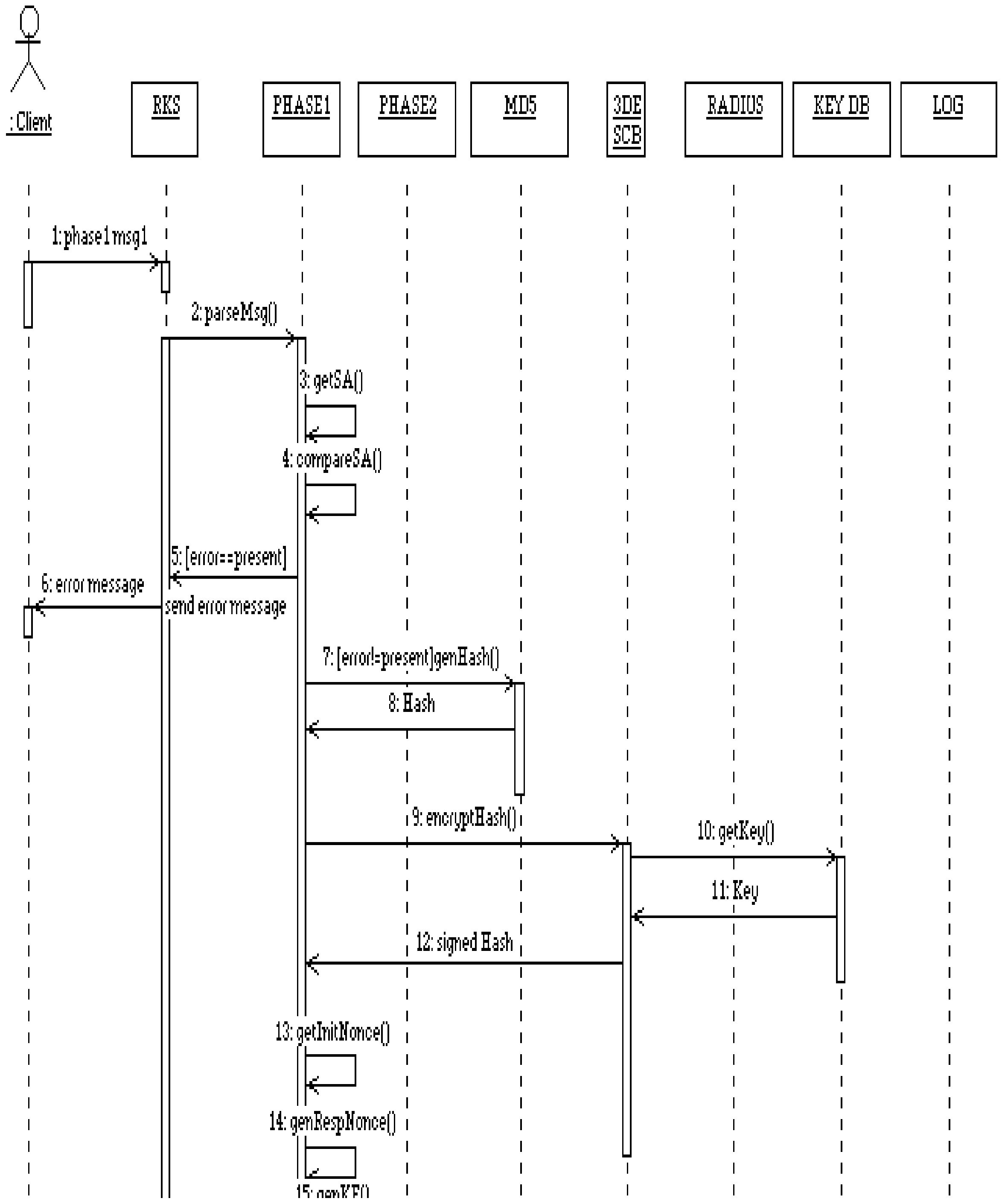
THe Sequence diagram for the TGS is shown below:

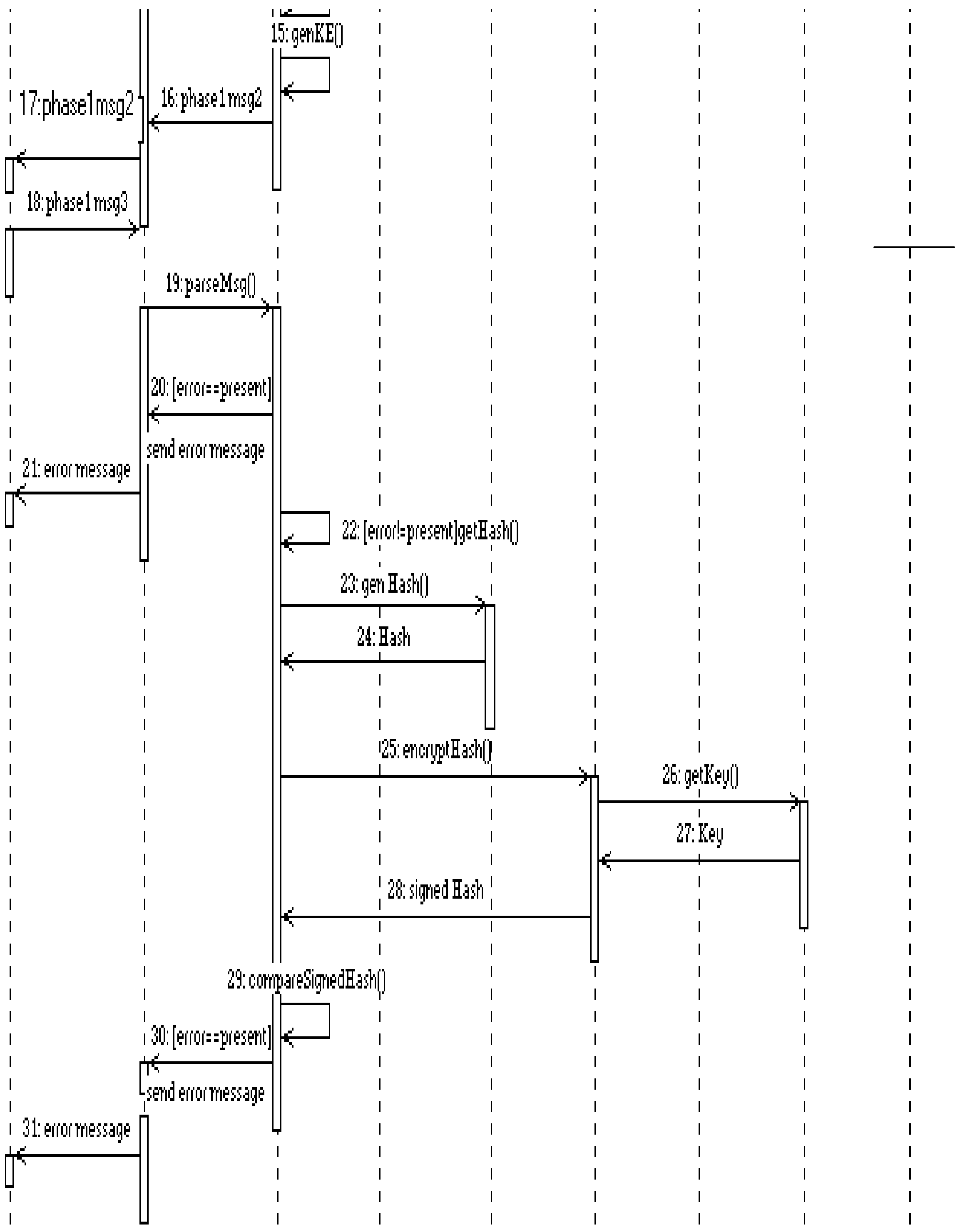


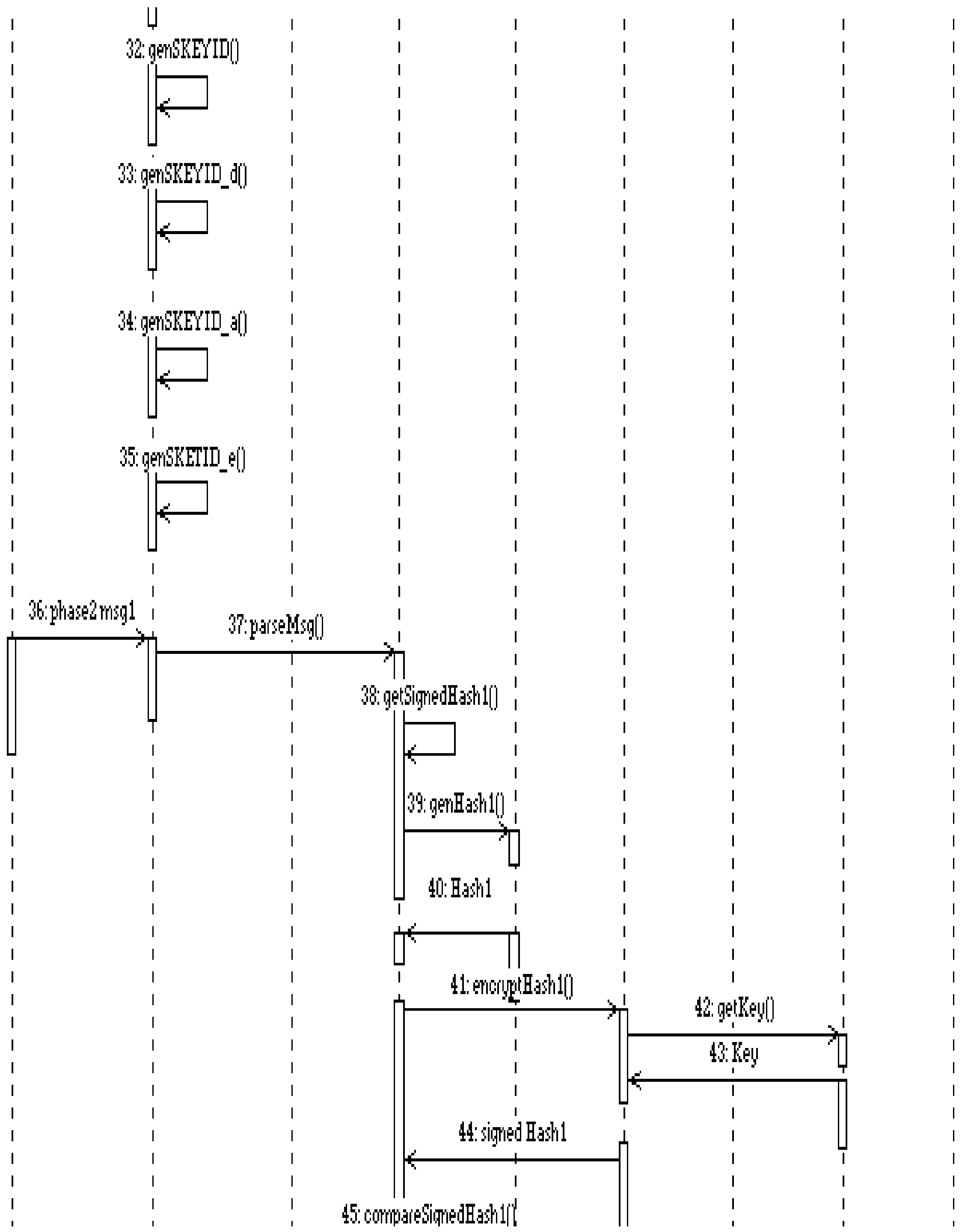
: MTA
TGS
ASN
EncDec
Digest
Dbase
1: TGS-REQ
2: Decode
3: Decoded info
4: Lookup client
5: Search results
6: [resp==FALSE]
Error(client not found)
7: [resp==TRUE]
Lookup server
8: Search results
9: [resp=FALSE]
Error(server not found)
10: [resp==TRUE]
Check pa-data type
11: [pa-data type == NULL]
Error(pre-auth required)
12: [pa-data type==PA-PK-AS-REQ]
Send pa-data value
13: Decoded pa-data value
14: Check digest algo
15: [digest algo != SHA-1]
Error(invalid digest)
16: [digest algo == SHA-1]
Compare(certificate.cname,request.cname)
17: [resp == FALSE]
Error(client name mismatch)
18: [resp==TRUE]
Obtain public key from certificate
19: Store public key
20: Obtain signature from signerinfo
21: Send public key and signature
22: Return SHA-1 digest
23: Send encoded AuthPack
24: Return decoded AuthPack
25: Send PKAuthenticator
26: Return SHA-1 digest
27: Compare digests
28: [resp==FALSE]
Error(invalid signature)
29: [resp==TRUE]
Check for possible replay
30: [replay]
Error(preauthentication failed)
31: [!replay]
Check for the presence of DH parameters
32: [DH parameters not present]
Error(DH param not found)
33: [DH parameters present]
Check valid range for parameters

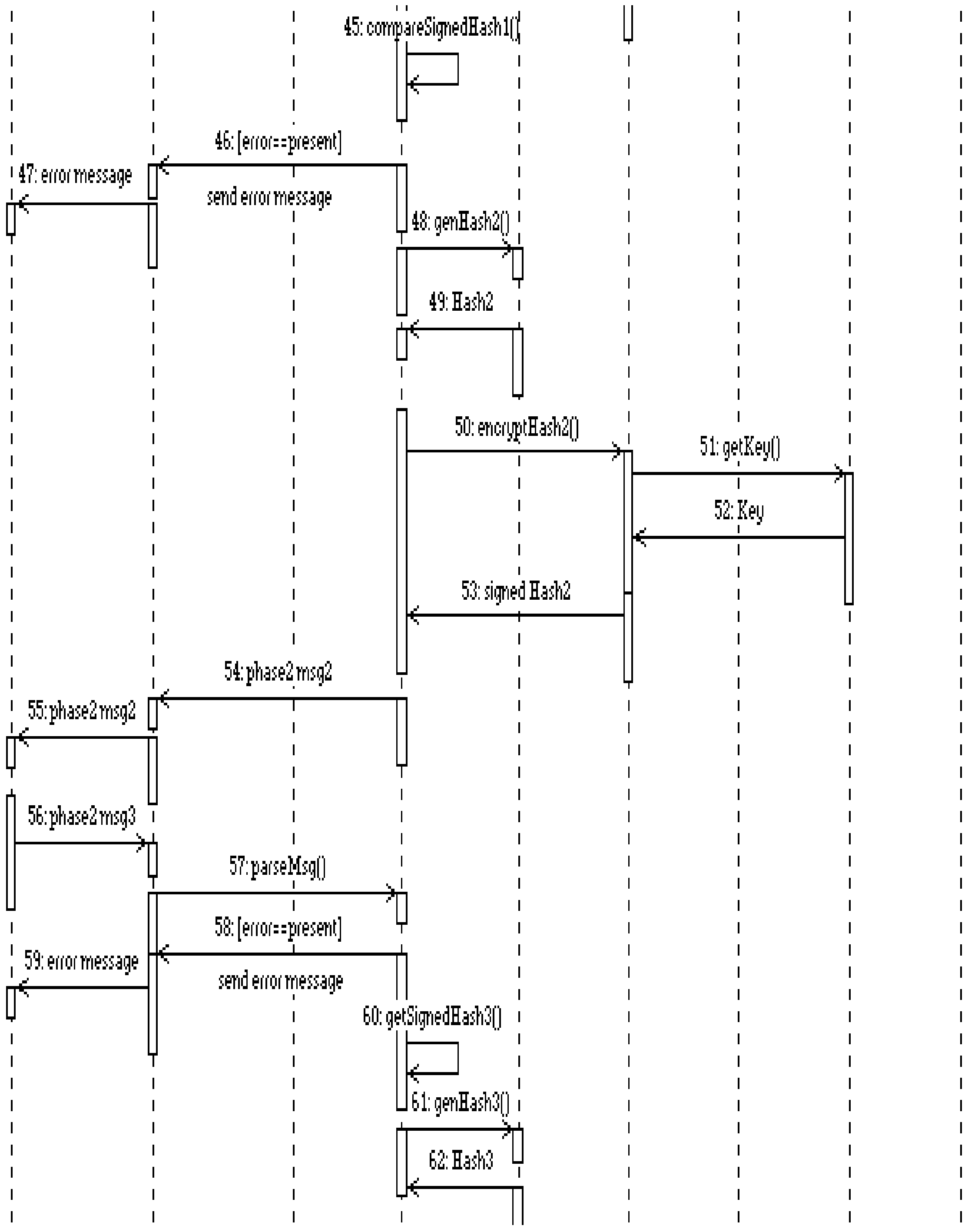

38: Send KdcDHkeyinfo
39: Encoded KdcDH keyinfo
40: Send encoded KdcDHkeyinfo
41: Return SHA-1 digest
42: Send digest
43: Return signature
44: Generate session key
45: Compose client specific part of reply
46: Send client specific part of reply
47: Return encoded value
48: Send(encoded value,confounder,cksum)
49: Return MD5 digest
50: Replace cksum
51: Send(concatenated bitstring,DH key)
52: Return encrypted reply part
53: Compose EncTicketPart
54: Send EncTicketPart
55: Encoded value
56: Send(encoded value,confounder,cksum)
57: Return MD5 digest
58: Replace cksum
59: Send(concatenated bitstring,server secret key)
60: Return encrypted ticket part
61: Compose ticket
62: Compose reply
63: Send reply
64: Return encoded reply
65: Send TGS-REP

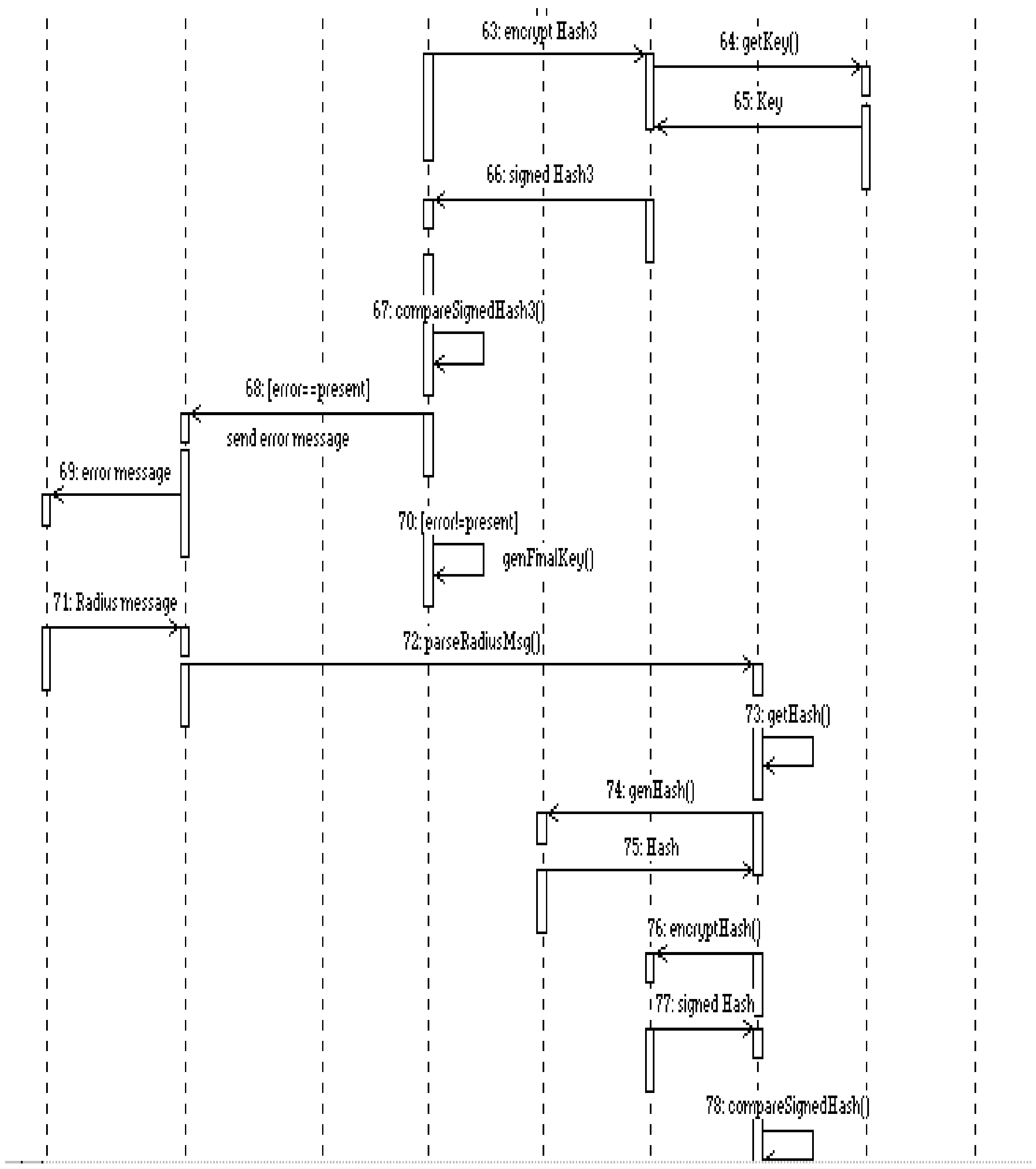
APPENDIX C


: Client
RKS
PHASE1
PHASE2
MD5
3DE SCB
RADIUS
KEY DB
LOG
1:phase1msg1
2:parseMsg()
3: getSA()
4: compareSA()
5: [error==present]
6: error message
send error message
7: [error!=present]genHash()
8: Hash
9: encryptHash()
10: getKey()
11: Key
12: signed Hash
13: getInitNonce()
14: genRespNonce()
15: genKE()

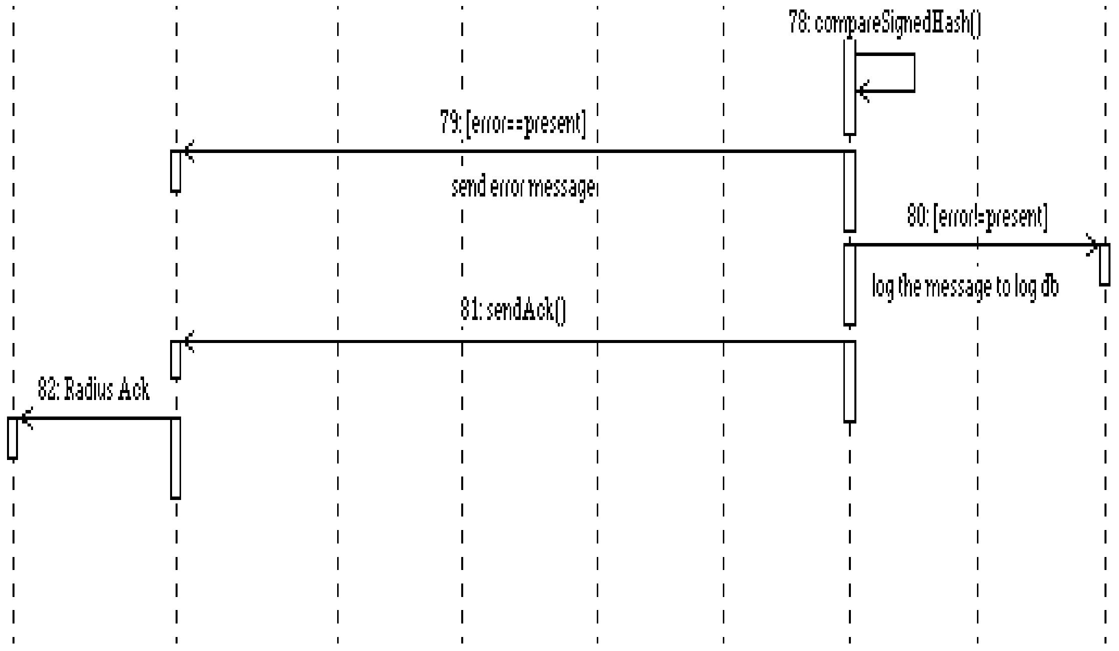

15: genKE()
16: phase1msg2
17:phase1msg2
18: phase1msg3
19: parseMsg()
20: [error==present]
send error message
21: error message
22: [error!=present]getHash()
23: gen Hash()
24: Hash
25: encryptHash()
26: getKey()
27: Key
28: signed Hash
29: compareSignedHash()
30: [error==present]
send error message
31: error message


32: genSKEYID()
33: genSKEYID_d()
34: genSKEYID_a()
35: genSKETID_e()
36: phase2msg1
37: parseMsg()
38: getSignedHash1()
39: genHash1()
40: Hash1
41: encryptHash1()
42: getKey()
43: Key
44: signed Hash1
45: compareSignedHash1()


45: compareSignedHash1()
46: [error==present]
send error message
47: error message
48: genHash2()
49: Hash2
50: encryptHash2()
51: getKey()
52: Key
53: signed Hash2
54: phase2 msg2
55: phase2 msg2
56: phase2 msg3
57: parseMsg()
58: [error==present]
send error message
59: error message
60: getSignedHash3()
61: genHash3()
62: Hash3



63: encrypt Hash3
64: getKey()
65: Key
66: signed Hash3
67: compareSignedHash3()
68: [error==present]
send error message
69: error message
70: [error!=present]
genFinalKey()
71: Radius message
72: parseRadiusMsg()
73: getHash()
74: genHash()
75: Hash
76: encryptHash()
77: signed Hash
78: compareSignedHash()


78: compareSignedHash()
79: [error==present]
send error message
80: [error!=present]
log the message to log db
81: sendAck()
82: Radius Ack

GLOSSARY

Announcement Server
An announcement server plays informational announcements in PacketCable network. Announcements are needed for communications that do not complete and to provide enhanced information services to the user.

Asymmetric Key An encryption key or a decryption key used in a public key cryptography, where encryption and decryption keys are always distinct.

Authentication The process of verifying the claimed identity of an entity to another entity.

Authorization The act of giving access to a service or device if one has the permission to have the access.

CBC Cipher block chaining mode is an option in block ciphers that combine (XOR) the previous block of ciphertext with the current block of plaintext before
encrypting that block of the message.

Cipher An algorithm that transforms data between plaintext and ciphertext.

Ciphertext The (encrypted) message output from a cryptographic algorithm that is in a format that is unintelligible.

Cleartext The original (unencrypted) state of a message or data.

CM DOCSIS Cable Modem.

CMS Cryptographic Message Syntax

CMS Call Management Server. Controls the audio call connections. Also called a Call Agent in MGCP/SGCP terminology.

CMTS Cable Modem Termination System, the device at a cable head-end which implements the DOCSIS RFI MAC protocol and connects to CMs over an HFC network.

Codec COder-DECoder

Cryptographic algorithm
An algorithm used to transfer text between plaintext and ciphertext.

Decipherment A procedure applied to ciphertext to translate it into plaintext.

Decryption A procedure applied to ciphertext to translate it into plaintext.

Decryption key The key in the cryptographic algorithm to translate the ciphertext to plaintext

DHCP Dynamic Host Configuration Protocol.

Digital certificate A binding between an entity's public key and one or more attributes relating to its identity, also known as a public key certificate

Digital signature A data value generated by a public key algorithm based on the contents of a block of data and a private key, yielding an individualized cryptographic checksum

DNS Domain Name Server

DOCSIS Data Over Cable System Interface Specification.

DQoS Dynamic Quality of Service, i.e. assigned on the fly for each call depending on the QoS requested

E-MTA Embedded MTA – a single node which contains both an MTA and a cable modem.

Encipherment A method used to translate information in plaintext into ciphertext.

Encryption A method used to translate information in plaintext into ciphertext.

Encryption Key The key used in a cryptographic algorithm to translate the plaintext to ciphertext.

Event Message Message capturing a single portion of a call connection

Header Protocol control information located at the beginning of a protocol data unit.

HFC Hybrid Fiber/Coaxial , HFC system is a broadband bi-directional shared media transmission system using fiber trunks between the head-end and the fiber nodes, and coaxial distribution from the fiber nodes to the customer locations.

HTTP Hyper Text Transfer Protocol. Refer to IETF RFC 1945 and RFC 2068.

IETF Internet Engineering Task Force. A body responsible, among other things, for developing standards used in the Internet.

IKE Internet Key Exchange is a key management mechanism used to negotiate and derive keys for SAs in IPSec.

IKE– A notation defined to refer to the use of IKE with pre-shared keys for authentication.

IKE+ A notation defined to refer to the use of IKE, which requires digital certificates for authentication.

Integrity A way to ensure that information is not modified except by those who are authorized to do so.

IP Internet Protocol. An Internet network-layer protocol.

IPSec Internet Protocol Security, a collection of Internet standards for protecting IP packets with encryption and authentication.

ISDN Integrated Services Digital Network

ISUP ISDN User Part is a protocol within the SS7 suite of protocols that is used for call signaling within an SS7 network.

ISTP Internet Signaling Transport Protocol

ISTP – User Any element, node, or software process that uses the ISTP stack for signaling communications.

ITU International Telecommunication Union

Kerberos A secret-key network authentication protocol that uses a choice of cryptographic algorithms for encryption and a centralized key database for authentication.

Key A mathematical value input into the selected cryptographic algorithm.

Key Exchange The swapping of public keys between entities to be used to encrypt communication between the entities.

Key Management The process of distributing shared symmetric keys needed to run a security protocol.

MAC Message Authentication Code - a fixed length data item that is sent together with a message to ensure integrity, also known as a MIC.

MD5 Message Digest 5 - a one-way hash algorithm which maps variable length plaintext into fixed length (16 byte) ciphertext.

MIB Management Information Base

MSO Multi-System Operator, a cable company that operates many head-end locations in several cities.

MTA Media Terminal Adapter – contains the interface to a subscribers' CPE, a network interface, CODECs, and all signaling and encapsulation functions required for VoIP transport, class features signaling, and QoS signaling.

Nonce A random value used only once which is sent in a communications protocol exchange to prevent replay attacks.

Non-Repudiation The ability to prevent a sender from denying later that he or she sent a message or performed an action.

OSS Operations Systems Support. The back office software used for configuration, performance, fault, accounting and security management.

PDU Protocol Data Unit

PKINIT The extension to the Kerberos protocol that provides a method for using public key cryptography during initial authentication.

Proxy A facility that indirectly provides some service or acts as a representative in delivering information there by eliminating a host from having to support the services themselves.

PSTN Public Switched Telephone Network.

Public Key The key used in public key cryptography that belongs to an individual entity and is distributed publicly. Other entities use this key to encrypt data to be sent to the owner of the key.

Public Key Certificate

A binding between an entity’s public key and one or more attributes relating to its identity, also known as a digital certificate.

Public Key Cryptography

A procedure that uses a pair of keys, a public key and a private key for encryption and decryption, also known as asymmetric algorithm. A user’s public key is publicly available for others to use to send a message to the owner of the key. A users private key is kept secret and is the only key which can decrypt messages sent encrypted by the users public key.

QoS Quality of Service, guarantees network bandwidth and availability for applications.

RADIUS Remote Access Dial-In User Service, an internet protocol (RFC 2138 and RFC 2139) originally designed for allowing users dial-in access to the internet through remote servers.

RKS Record Keeping Server, the device which collects and correlates the various Event Messages

Secret Key The cryptographic key used in a symmetric key algorithm, which results in the secrecy of the encrypted data depending solely upon keeping the key a secret, also known as a symmetric key.

Session Key A cryptographic key intended to encrypt data for a limited period of time, typically between a pair of entities.

TGS Ticket Granting Server used to grant Kerberos tickets.

UDP User Datagram Protocol, a connectionless protocol built upon Internet Protocol (IP).

VoIP Voice over IP

CHAPTER 14

References And Bibliography

[1] *PacketCable Product Specification*, Cable Television Laboratories Inc.,November 25, 1998.

[2] *PacketCable Security Specification*, PKT-SP-SEC-I01-991201, Cable Television Laboratories, Inc., December 1, 1999, http://www.PacketCable.com./

[3] *PacketCable 1.0 Architecture Framework Technical Report*, PKT-TR-ARCH-I01-991201, December 1, 1999, Cable Television Laboratories, Inc., http://www.PacketCable.com./

[4] PacketCable Event Messages White Paper, PKT-OSS-TD01-990329.

[5] C. Rigney, "RADIUS Accounting", IETF RFC-2139, April 1997

[6] PacketCable Architecture Call Flow Technical Report, On-Net MTA to On-Net MTA, PKT-TR-CF-ON-ON-D01-991201, December 1, 1999, Cable Television Laboratories, Inc., http://www.PacketCable.com./

[7] PacketCable Architecture Call Flow Technical Report, On-Net MTA to PSTN, PKT-TR-CF-ON-PSTN-D01-991201, December 1, 1999, Cable Television Laboratories, Inc., http://www.PacketCable.com/

[8] PacketCable Architecture Call Flow Technical Report, PSTN to On-Net MTA, PKT-TR-CF-PSTN-ON-D01-991201, December 1, 1999, Cable Television Laboratories, Inc., http://www.PacketCable.com/

[9] Data Encryption Standard, FIPS-PUB 46-3, October 25, 1999.

[10] "PacketCable MTA MIB," PKT-SP-MIBS-MTA-I01-991201, Cable Television Laboratories, Inc., December 1, 1999. http://www.PacketCable.com./

[11] Simple Network Management Protocol Version 2 (SNMPv2), William Stallings.

[12] Networking Essentials, Tanenbaum.

[13] "PacketCable NCS MIB," PKT-SP-MIBS-NCS-I01-991201, Cable Television Laboratories, Inc., December 1, 1999. http://www.PacketCable.com./

[14] SNMPv2-TM, RFC1449.

[15] SNMPv2-TC, RFC1903.

[16] Operations Support System Interface Specification Radio Frequency Interface, sp-ossi-rfi-i03-990113, Cable Television Laboratories, Inc., January 13, 1999, http://www.CableLabs.com/

[17] A Simple Network Management Protocol (SNMP), IETF RFC-1157, May 1990.

[18] "PacketCable Provisioned QoS Specification**,"** PKT-SP-PQoS-D02-990603, June 18, 1999, Cable Television Laboratories, Inc.

[19] *PacketCable Event Messages,* PKT-SP-EM-I01-991201, December 1, 1999, Cable Television Laboratories, Inc., http://www.PacketCable.com./

[20] *PacketCable OSS Overview,* PKT-TR-OSS-I01-991201, December 1, 1999, Cable Television Laboratories, Inc., http://www.PacketCable.com./

[21] *HMAC: Keyed-Hashing for Message Authentication*, IETF (Krawczyk, Bellare,and Canetti), Internet Proposed Standard, RFC 2104, March 1996.

[22] *The Kerberos Network Authentication Service (V5),* IETF Draft, Clifford Neuman, John Kohl, Theodore Ts'o, http://www.ietf.org/internet-drafts/draft-ietf-cat-kerberos-revisions-04.txt, July, 1999.

[23] *Public Key Cryptography for Initial Authentication in Kerberos*, IETF Draft (B.Tung, C. Neuman, J. Wray, A. Medvinsky, M. Hur, S. Medvinsky, J. Trostle), http://www.ietf.org/internet-drafts/draft-ietf-cat-kerberos-pk-init-09.txt, July, 1999.

[24] *Cryptographic Message Syntax*, IETF (R. Housley), Internet Proposed Standard, RFC 2630, June 1999.

[25] *RADIUS Accounting*, IETF (C. Rigney), Internet Proposed Standard, RFC 2139,April 1997.

[26] *Secure Hash Algorithm*, Department of Commerce, NIST, FIPS 180-1, April,1995.

[27] *PKCS #1: RSA Encryption Standard. Version 1.5*, RSA Laboratories, November, 1993.

[28] *PKCS #1: RSA Encryption Standard. Version 2*, RSA Laboratories, September, 1998.

[29] *PKCS #7: Cryptographic Message Syntax Standard*, RSA Laboratories, November, 1993.

[30] *The ESP CBC-Mode Cipher Algorithms*, IETF (R. Pereira, R. Adams), Internet Proposed Standard, RFC 2451, November 1998.

[31] *The Use of HMAC-SHA-1-96 within ESP and AH*, IETF (C. Madson, R. Glenn) ,Internet Proposed Standard, RFC 2404, November 1998.

[32] *The Internet Key Exchange (IKE),* IETF (D. Harkins, D. Carrel), Internet Proposed Standard, RFC 2409, November 1998.

[33] *PacketCable MTA MIB*, PKT-SP-MIBS-MTA-I01-991201, Cable Television Laboratories, Inc., December 1, 1999. http://www.PacketCable.com./

[34] Schneier “Applied Cryptography,” John Wiley & Sons Inc, second edition,1996.

[35] *Operations Support System Framework for Data Over Cable Services*, TR-DOCS-OSSIW08-961016, Cable Television Laboratories Inc., October 16, 1996.

[36] *PacketCable MTA Device Provisioning Specification*, PKT-SP-PROV-I01-991201, December 1, 1999, Cable Television Laboratories, Inc., http://www.PacketCable.com./

[37] RADIUS White paper, Lucent Technologies, February 2000.

[38] Diffie-Hellman Key Exchange Method, RFC-2631.

[39] Abstract Syntax Notation (ASN.1), France Telecom, 1997.

[40] Linux Unleashed.

[41] Redhat Unleashed.

[42] Linux Complete Command Reference.

[43] ASN.1, Communication between heterogeneous systems, Oliver Dubuisson, June 5, 2000.

[44] ASN.1 Complete, Prof. John Larmouth, May 31, 1999.