



Jaypee University of Information Technology
Solan (H.P.)

LEARNING RESOURCE CENTER

Acc. Num. *SP04006* Call Num:

General Guidelines:

- ◆ Library books should be used with great care.
- ◆ Tearing, folding, cutting of library books or making any marks on them is not permitted and shall lead to disciplinary action.
- ◆ Any defect noticed at the time of borrowing books must be brought to the library staff immediately. Otherwise the borrower may be required to replace the book by a new copy.
- ◆ The loss of LRC book(s) must be immediately brought to the notice of the Librarian in writing.

Learning Resource Centre-JUIT



SP04006

File Encryption System

**Submitted in partial fulfillment of the Degree of Bachelor of
Technology**

By

BHARAT SAINI - 041280

KUMAR SAURAV – 041220

SAI RAJU – 041221

MANISH KUMAR - 041402



May-2008

**DEPARTMENT OF COMPUTER SCIENCE
JAYPEE UNIVERSITY OF INFORMATION
TECHNOLOGY-WAKNAGHAT**

CERTIFICATE

This is to certify that the work entitled, "Mobile File Encryption System" submitted by Bharat Saini, Kumar Saurav, Sai Raju, Manish Kumar in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science of Jaypee University of Information Technology has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Supervisor:



Brig(Retd) S P Ghrera

HOD(CSE & IT)

Department of Computer Science Engineering and Information Technology,
Jaypee University of Information Technology,
Waknaghat, Solan – 173215, Himachal Pradesh,
INDIA.

ACKNOWLEDGMENT

We wish to express our earnest gratitude to Mr. S P Ghrera, for providing us invaluable guidance and suggestions, which inspired us to submit this project report on time.

We would also like to thank all the staff members of Computer Science and Engineering Department of Jaypee University of Information Technology, Wagnaghat, for providing us all the facilities required for the completion of this project report.

Last but not least we wish to thank all my classmates and friends for their timely suggestions and cooperation during the period of our project report.

Bharat Saini (041280)

Kumar Saurav (041220)

Sai Raju (041221)

Manish Kumar (041402)

Table of Contents

1. Introduction	
1.1 Project Aim.....	14
1.2 Project Scope.....	14
1.3 Cryptography & Related Terms	15
1.4 Cryptography and .Net Framework.....	17
2. Project Design	
2.1 DFD.....	19
2.2 Class Diagram	20
2.3 Flow Chart	22
2.4 Event Diagram	23
2.5 Modular Decomposition Of Project.....	25
2.6 Methodology And Approach.....	26
3. Algorithm Specification	
3.1 Symmetric Algorithms	27
3.2 Asymmetric Algorithms	52
4. Peer-to-Peer Communication.....	58
5. Key Distribution Center.....	120
5.1 Introduction.....	120
5.2 Background and basic concepts.....	120
5.2.1 Classifying keys by algorithm type and intended use.....	121
5.2.2 Key management objectives, threats, and policy.....	122
5.2.3 Simple key establishment models.....	123
5.2.4 Roles of third parties.....	126
5.2.5 Tradeoffs among key establishment protocols.....	129
5.3 Techniques for distributing confidential keys.....	130

5.3.1 Key layering and cryptoperiods.....	131
5.3.2 Key translation centers and symmetric-key certificates.....	132
5.4 Key life cycle issues.....	134
5.4.1 Lifetime protection requirements.....	135
5.4.2 Key management life cycle.....	135
5.5 KDC – The Implementation.....	139
5.6 KDC – Short Comings.....	140
6. Source Code.....	141
6.1 Main Screen Coding.....	141
6.2 Server Setting Code.....	143
6.3 Text Encryption Coding.....	144
6.4 DES Crypta –Analysis Coding.....	159
6.5 File Encryption Coding.....	163
6.6 KDC Server Side Coding.....	170
6.6.1 Login Page Code.....	170
6.6.2 Main Page Code.....	172
6.6.3 Change Key Pairs Code.....	175
6.6.4 Change Password Code.....	177
6.6.6 New User Code.....	180
6.6.7 Update Key Code.....	183
6.7 KDC Server Code.....	186
6.8 KDC Stand Alone Server Code.....	193
6.9 KDC Client Side Coding.....	199
6.9.1 Login Page Code.....	199
6.9.2 Main Page Code.....	203
6.9.3 Change Key Pairs Code.....	205
6.9.4 Change Password Code.....	207
6.9.5 Session Key Connect Code.....	209
6.9.6 Session Key Server Code.....	216
6.9.7 Get IP Code.....	224

6.9.8 KDC Client Code.....	225
6.9.9 Session Key Chat Server Code.....	230
6.9.10 Session Key Chat Client Code.....	241
6.9.11 Settings Code.....	249
7 Installation And Testing	251
7.1 Setup Guide	251
7.2 User Guide.....	256
7.3 Test Data.....	271
8 Conclusion And Future Work.....	277
9 Bibliography.....	278

LIST OF IMPORTANT FIGURES

Fig 1.1	Sender, Receiver, and attacker
Fig 3.1	Symmetric cryptography
Fig 3.2	General Depiction of DES Encryption Algorithm
Fig 3.3	Single Round of DES Algorithm
Fig 3.4	Calculation of $F(R, K)$
Fig 3.5	Flow chart of AES Enc. And Dec.
Fig 3.6	AES data structure
Fig 3.7	Byte Substitution
Fig 3.8	Shift Row
Fig 3.9	Mix Column
Fig 3.10	Add Round Key
Fig 3.11	Key Expansion
Fig 3.12	AES Round
Fig 4.1	Main Screen
Fig 4.2	Symmetric Algorithms (Without User Key)
Fig 4.3	Symmetric Algorithms Using User Key
Fig 4.4	Asymmetric Algorithms
Fig 4.5	RSA
Fig 4.6	RSA (Message Verified)
Fig 4.7	RSA (Message Not Verified)
Fig 4.8	DSA
Fig 4.9	DSA (Message Verified)
Fig 4.10	DSA (Message Not Verified)
Fig 4.11	DES Crypta-Analysis (Incomplete)
Fig 4.12	DES Crypta-Analysis (Complete)
Fig 2.2.1	Client-Server Model

Fig 2.2.2	Distributed Computing
Fig 2.2.3	Peer to Peer Computing
Fig 2.2.4	Pure Peer to Peer
Fig 2.2.5	Basics of Discovery Service
Fig 5.1	Simple key distribution models (symmetric-key)
Fig 5.2	In-line, on-line, and off-line third parties
Fig 5.3	Third party services related to public-key certification
Fig 5.4	Key management: symmetric-key vs. public-key encryption
Fig 5.5	Key management life cycle
Fig 5.6	Key Distribution Center
Fig 5.7	Key Distribution Data Base

LIST OF ABBREVIATIONS

- | | | |
|----|-----|------------------------------|
| 1. | GUI | GRAPHIC USER INTERFACE |
| 2. | DES | DATA ENCRYPTION STANDARD |
| 3. | AES | ADVANCED ENCRYPTION STANDARD |
| 4. | IV | INITIALIZATION VECTOR |
| 5. | WWW | WORLD WIDE WEB |

ABSTRACT

In this age of viruses and hackers, of electronic eavesdropping and electronic fraud, security is paramount. As the disciplines of cryptography and network security have matured, more practical, readily available applications to enforce network security have developed. To protect files/messages transferred over a network or any other means like removable disk, it should be encrypted. Various methods have been developed over several years to encrypt data. Long before the computer was invented, people used various methods to encrypt data. There are many traditional methods which have now transformed into modern techniques which are employed to protect data from unwanted access. As the time lapses the modern algorithms become obsolete and new method are invented for encryption.

CHAPTER 1

INTRODUCTION

1.1 Project Aim

This project encrypts and decrypts various text, files and folders. Encryption and Decryption is done using symmetric, asymmetric and hashing algorithms which are the modern day standards. This project also aims at public distribution of secret key (session key) using Key Distribution Center.

1.2 Project Scope

This project will focus on creating a file-system that a user can carry around with them. Such a file-system provides several advantages. First, combined with strong encryption, it provides a two-factor encryption and security mechanism for protecting the confidentiality of data on disk. Second, it supports ubiquitous computing and user-aware systems. Apart from this, project aims at KDC implementation for secure key exchange between two parties.

1.3 Cryptography & Related Terms

1.3.1 Cryptography

Security is the art of protecting access to information and other computing resources from those whom we do not fully trust. Cryptography is the science of keeping secrets. Cryptography is generally nothing more than hiding large secrets (which are themselves awkward to hide) with small secrets (which are more convenient to hide).

1.3.2 Basic Cryptographic Terminology

A *cipher* is a system or an algorithm used to transform an arbitrary message into a form that is intended to be unintelligible to anyone other than one or more desired recipients. A cipher represents a transformation that maps each possible input message into a unique encrypted output message, and an inverse transformation must exist which can reproduce the original message.

A *key* is used by a cipher as an input that controls the encryption in a desirable manner. A general assumption in cryptography is that the key we choose is critical secret, whereas the details about the cipher design should not be assumed to be secret. A *key space* is the set of all possible keys that can be used by a cipher to encrypt messages.

The original message is referred to as *plaintext*. The word plaintext is not meant to imply that the data is necessarily human readable or that it is ASCII text. The plaintext can be any data (text or binary) that is directly meaningful to someone or to some program.

The encrypted message is referred to as *ciphertext*. Ciphertext makes it possible to transmit sensitive information over an insecure channel or to store sensitive information on an insecure storage medium.

The design and application of ciphers is known as *cryptography*, which is practiced by *cryptographers*. The breaking of ciphers is known as *cryptanalysis*. *Cryptology* refers to the combined mathematical foundation of cryptography and cryptanalysis. A *cryptanalytic attack* is the application of specialized techniques that are used to discover the key and/or plaintext originally used to produce a given ciphertext.

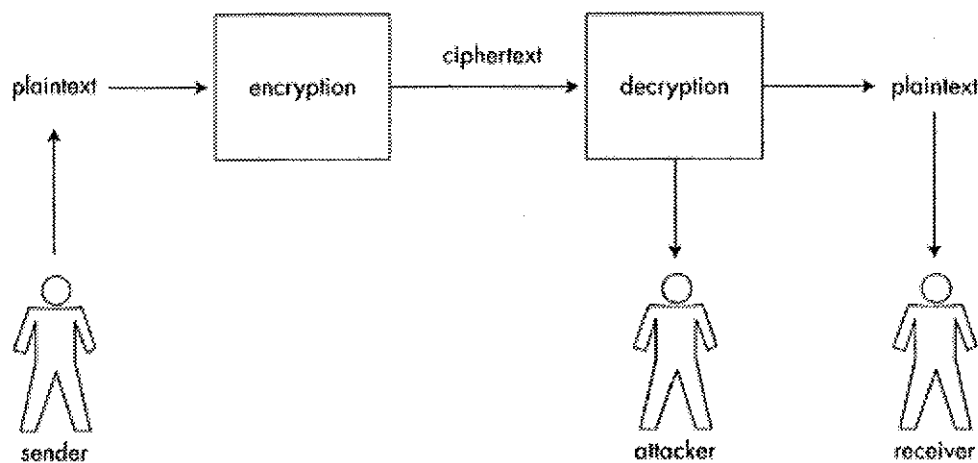


Figure 1.1 Sender, receiver, and attacker

1.3.3 Cryptanalytic Attacks

There is an accepted terminology used to categorize the various possible types of cryptanalytic attacks. The following types of attacks are listed in order from hardest to easiest in terms of analytical difficulty

- *Ciphertext-only attack*: Attacker has only some randomly selected ciphertext.
- *Known plaintext attack*: Attacker has some randomly selected plaintext and corresponding ciphertext.
- *Chosen plaintext attack*: Attacker has some chosen plaintext and corresponding ciphertext.
- *Chosen ciphertext attack*: Attacker has some chosen ciphertext and the corresponding decrypted plaintext.

- *Adaptive chosen plaintext attack*: Attacker can determine the ciphertext of chosen plaintexts in an iterative manner building on previous calculations. This type of attack is also referred to as a differential cryptanalysis attack.

1.4 Cryptography and the .NET Framework

The .NET Framework class library provides the *System.Security.Cryptography* namespace, which supports the most important symmetric and asymmetric ciphers as well as several secure hash algorithms and a cryptographic-quality random number generator. This cryptography architecture is extensible, allowing third parties to provide alternative implementations and additional algorithms, in the form of cryptographic service providers. The *System.Security.Cryptography.XML* namespace implements the W3C standard for digitally signing XML objects, and the *System.Security.Cryptography.X509Certificates* namespace provides some support for working with public certificates. Here are the major standards implemented by the .NET Framework class library.

- DES: Digital Encryption Standard (symmetric block cipher)
- 3DES: Triple DES (symmetric block cipher; stronger alternative to DES)
- Rijndael: AES (symmetric block cipher)

The U.S. government selected Rijndael as the AES (Advanced Encryption Standard) encryption standard in October 2000, replacing the DES.

- RC2: Cipher design by Ronald Rivest (symmetric stream cipher)
- RSA: Cipher design by Rivest, Shamir, and Adleman (asymmetric algorithm for both encryption and digital signatures)
- DSA: Digital Signature Algorithm (asymmetric algorithm only for digital signatures)
- MD5: Message digest (i.e., a secure hash) algorithm developed by Rivest
- SHA-1, SHA-256, SHA-384, SHA-512: Standard secure hash algorithms developed by NIST (National Institute of Standards and Technology) along with the NSA (for use with the Digital Signature Standard)

- Pseudorandom Number Generator (PRNG)
- XML Signatures: Digital signatures for XML data
- X.509: Public certificates standard

The .NET Framework provides the following classes for working with several important symmetric algorithms.

- System.Security.Cryptography.DES
- System.Security.Cryptography.RC2
- System.Security.Cryptography.Rijndael
- System.Security.Cryptography.TripleDES

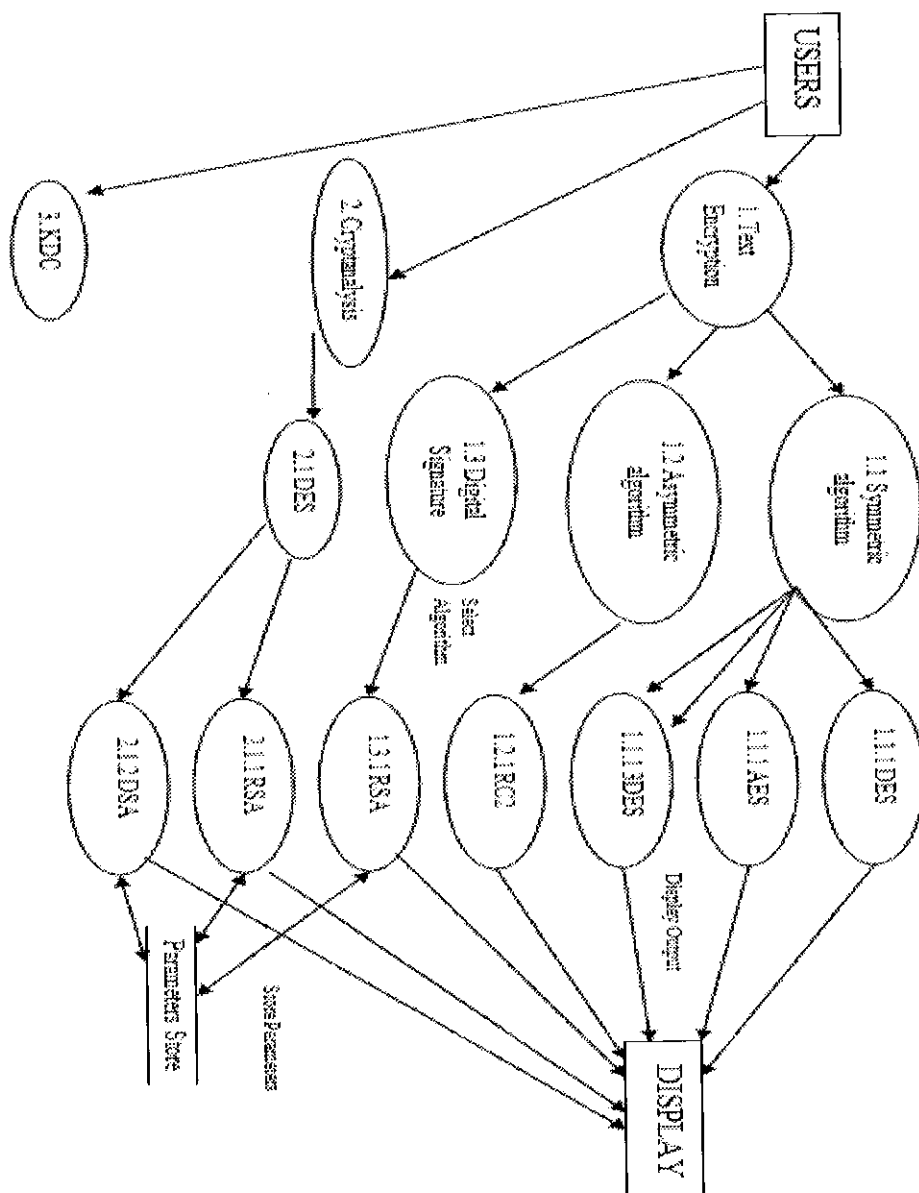
The .NET Framework provides the following classes for working with various asymmetric algorithms.

- System.Security.Cryptography.DSA
- System.Security.Cryptography.RSA

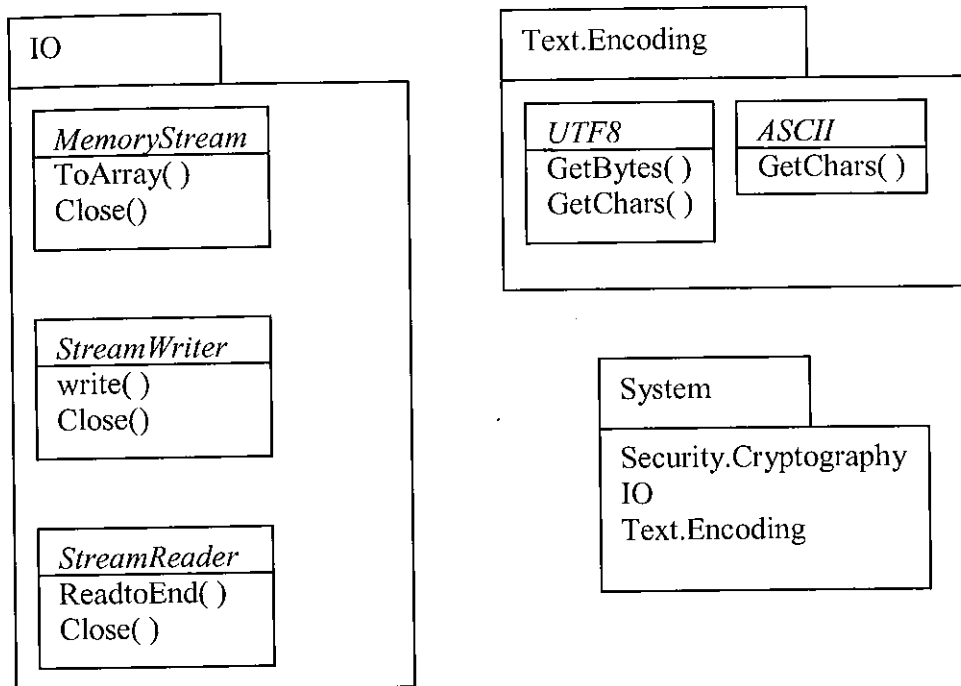
The following classes are provided by the .NET security framework library for secure hash functionality.

- System.Security.Cryptography.KeyedHashAlgorithm
- System.Security.Cryptography.MD5
- System.Security.Cryptography.SHA1
- System.Security.Cryptography.SHA256
- System.Security.Cryptography.SHA384
- System.Security.Cryptography.SHA512

2.1 DFD



2.2 Class Diagram

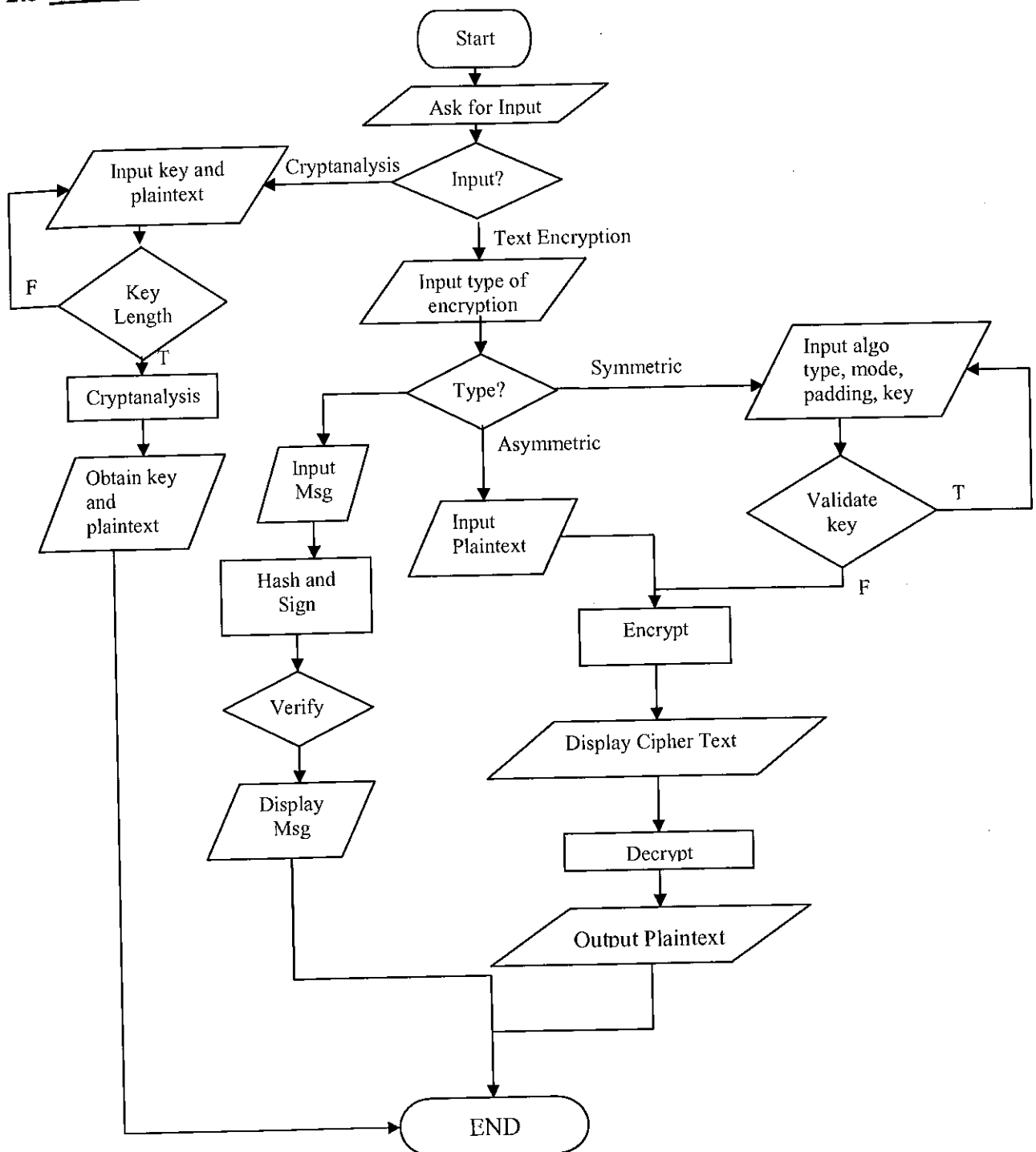


Security.Cryptography

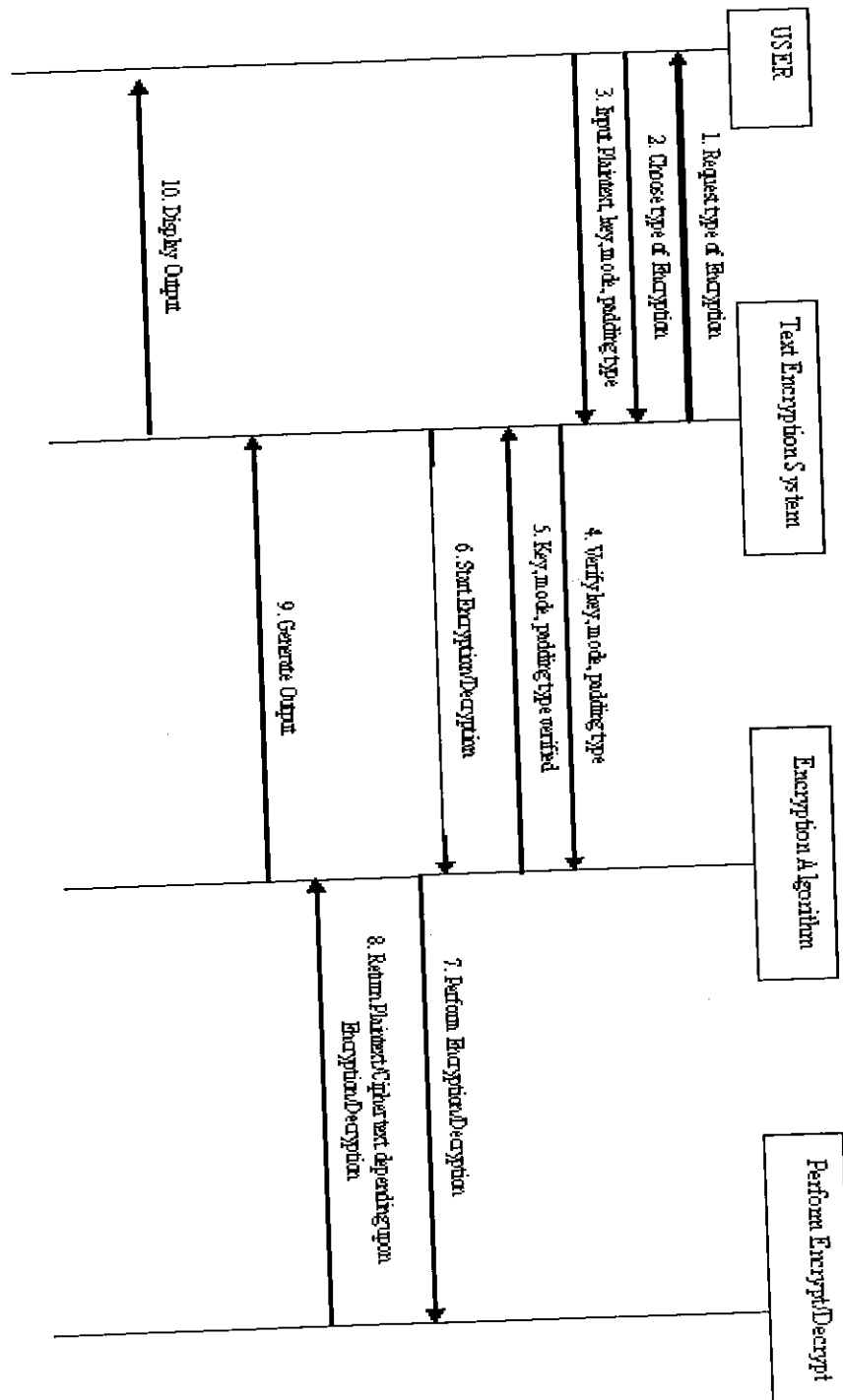
<i>Symmetric Algorithm</i> Key IV Mode Padding GenerateKey() GenerateIV() CreateDecryptor() CreateEncryptor()	<i>RSACryptoServiceProvider</i> SignHash() ExportParameters() toXmlString() Decrypt() Encrypt() ImportParameters() VerifyHash()	<i>DSACryptoServiceProvider</i> SignHash() ExportParameters() toXmlString() ImportParameters() VerifyHash()	
<i>AES</i> Create()	<i>SHA1</i> CryptoServiceProvider() ComputeHash()	<i>CipherMode</i> ECB CBC CFC CTS	<i>PaddingMode</i> PKCS7 Zeros None
<i>DES</i> Create()			<i>CryptoStreamMode</i> Read() Write()
<i>Rijndael</i> Create()	<i>TripleDes</i> Create()	<i>CryptoStream</i>	

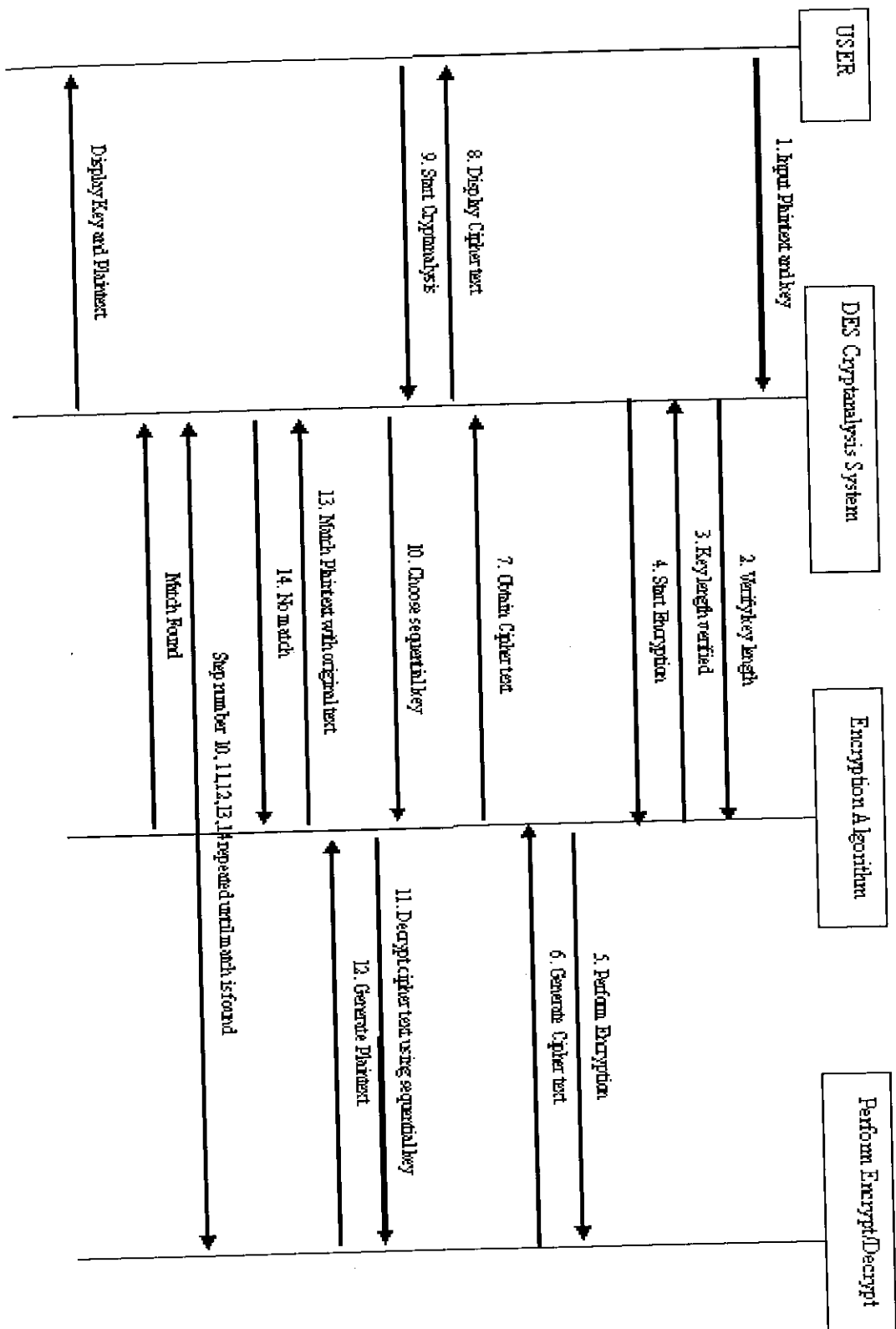


2.3 Flow Chart

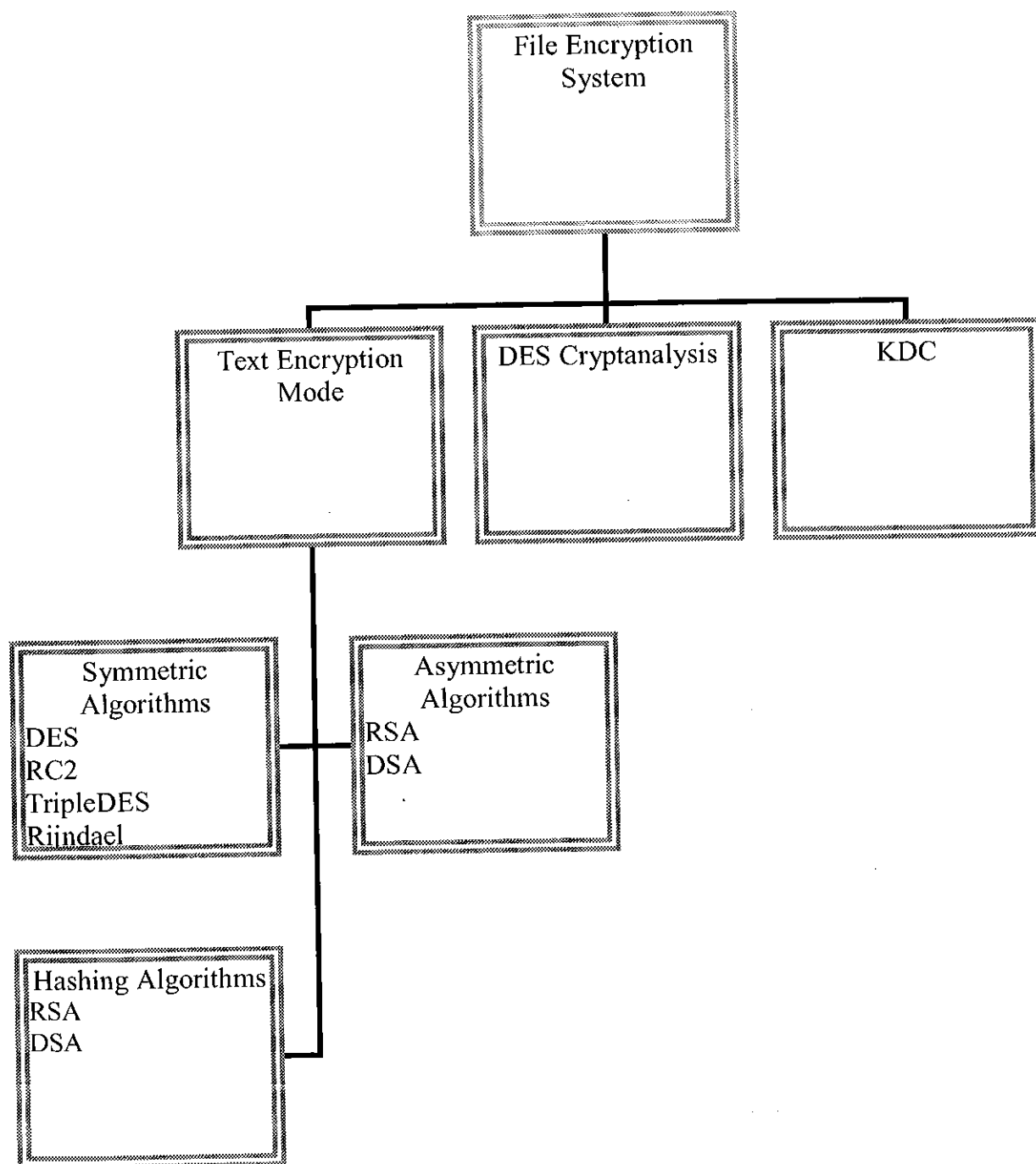


2.4 Event Diagram





2.5 Modular Decomposition Of Project



2.6 Project Design and Methodology

METHODOLOGY & APPROACH

- **GUI development**

For GUI development we studied all the rules for GUI design and various similar kinds of tools and observed how their GUI looked. From them we learnt our GUI like how should be the interface look like and the best possible way to simplify the interface. From all that we came to the conclusion that for such kind of tool like packet capturing tree structure formation or hierarchical would be the best possible way, so we implemented it.

- **Text Encryption**

For the text encryption we had studied various encryption algorithms and modes of these algorithms. After studying these algorithms we searched for a suitable platform which supports the features of cryptography. We found out that .net was an apt platform to implement these algorithms. Then we studied the classes for cryptography which was present in .net. With the help of this knowledge we developed a multifunctional tool which successfully implemented asymmetric, symmetric and digital signatures.

- **Cryptanalysis**

For cryptanalysis we have performed DES cryptanalysis. For DES we have used the brute force technique, which is trying all possible key combinations and getting the original plaintext by comparison. Since all key was not feasible we made the key domain restricted to 0-99999999.

CHAPTER 3

ALGORITHM SPECIFICATION

3.1 Symmetric Algorithms

A symmetric cipher is a cipher in which encryption and decryption use the same key or keys that are mathematically related to one another in such a way that it is easy to compute one key from knowledge of the other, which is effectively a single key. Since the single key is the only secret to encryption and decryption, it is critical that this key be kept strictly private.

Symmetric encryption and decryption are represented mathematically by the following, where E is an encryption function, D is a decryption function, k is the single shared secret key, M is a plaintext message, and C is the corresponding ciphertext message.

$$\text{Encryption: } C = E_k(M)$$

$$\text{Decryption: } M = D_k(C)$$

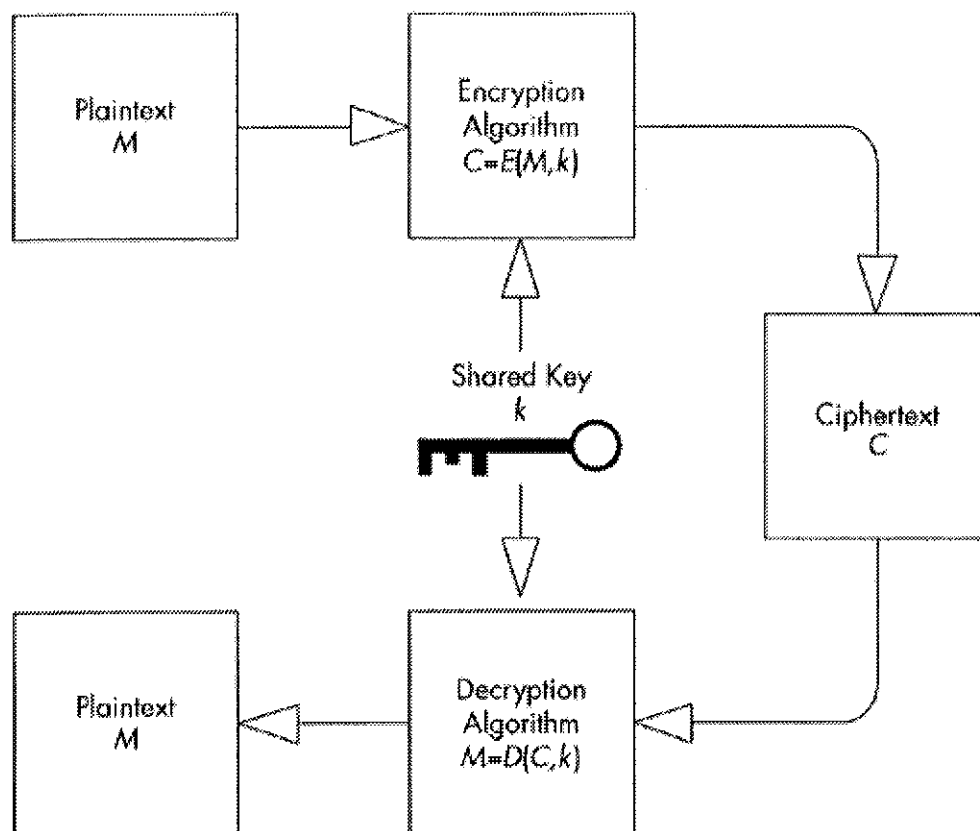


Figure 3.1 Symmetric cryptography

3.1.1 DES (Data Encryption Standard)

The most widely used encryption scheme is based on the Data Encryption Standard (DES) adopted in 1977 by the National Bureau of Standards, now the National Institute of Standards and Technology (NIST), as Federal Information Processing Standard 46 (FIPS PUB 46). The algorithm itself is referred to as the Data Encryption Algorithm (DEA). For DES, data are encrypted in 64-bit blocks using a 56-bit key. The algorithm transforms 64-bit input in a series of steps into a 64-bit output. The same steps, with the same key, are used to reverse the encryption.

The overall scheme for DES encryption is illustrated in Figure 3.2. There are two inputs to the encryption function: the plaintext to be encrypted and the key. In this case, the plaintext must be 64 bits in length and the key is 56 bits in length. Looking at the left-hand side of the figure, we can see that the processing of the plaintext proceeds in three phases. First, the 64-bit plaintext passes through an initial permutation (IP) that

rearranges the bits to produce the permuted input. This is followed by a phase consisting of 16 rounds of the same function, which involves both permutation and substitution functions. The output of the last (sixteenth) round consists of 64 bits that are a function of the input plaintext and the key. The left and right halves of the output are swapped to produce the preoutput. Finally, the preoutput is passed through a permutation (IP-1) that is the inverse of the initial permutation function, to produce the 64-bit ciphertext. The right-hand portion of Figure 3.2 shows the way in which the 56-bit key is used. Initially, the key is passed through a permutation function. Then, for each of the 16 rounds, a subkey (K_i) is produced by the combination of a left circular shift and a permutation. The permutation function is the same for each round, but a different subkey is produced because of the repeated shifts of the key bits.

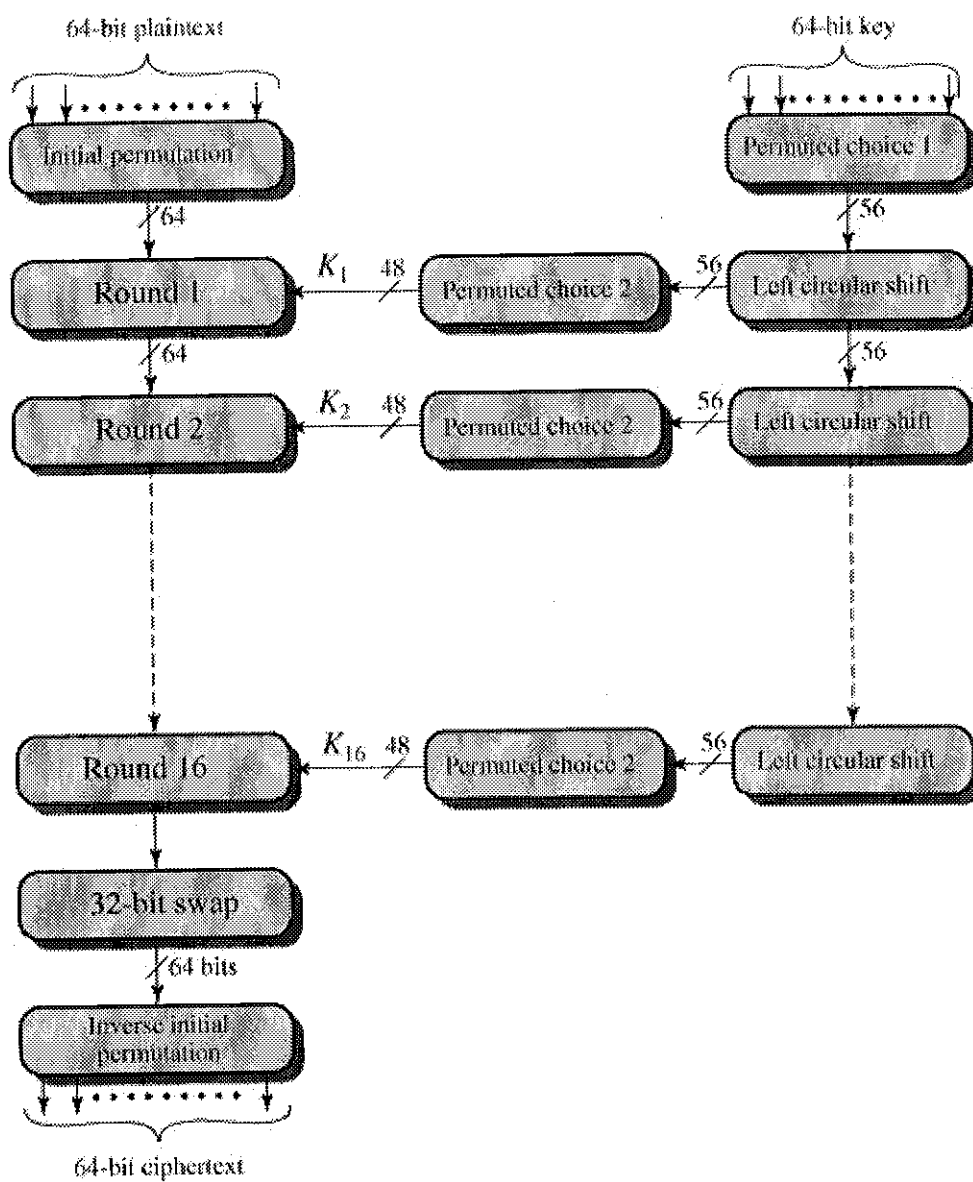


Figure 3.2 General Depiction of DES Encryption Algorithm

Initial Permutation

The initial permutation and its inverse are defined by tables, as shown in Tables 3.1a and 3.1b, respectively. The tables are to be interpreted as follows.

(a) Initial Permutation (IP)							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7
(b) Inverse Initial Permutation (IP ⁻¹)							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

(c) Expansion Permutation (E)							
	32	1	2	3	4	5	
	4	5	6	7	8	9	
	8	9	10	11	12	13	
	12	13	14	15	16	17	
	16	17	18	19	20	21	
	20	21	22	23	24	25	
	24	25	26	27	28	29	
	28	29	30	31	32	1	
(d) Permutation Function (P)							
16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25

TABLE 3.1 Permutation Tables for DES

The input to a table consists of 64 bits numbered from 1 to 64. The 64 entries in the permutation table contain a permutation of the numbers from 1 to 64. Each entry in the permutation table indicates the position of a numbered input bit in the output, which also consists of 64 bits.

Details of Single Round

Figure 3.3 shows the internal structure of a single round. Again, begin by focusing on the left-hand side of the diagram. The left and right halves of each 64-bit intermediate value are treated as separate 32-bit quantities, labeled L (left) and R (right). As in any classic Feistel cipher, the overall processing at each round can be summarized in the following formulas:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \times F(R_{i-1}, K_i)$$

The round key K_i is 48 bits. The R input is 32 bits. This R input is first expanded to 48 bits by using a table that defines a permutation plus an expansion that involves duplication of 16 of the R bits (Table 3.1c). The resulting 48 bits are XORed with K_i . This 48-bit result passes through a substitution function that produces a 32-bit output, which is permuted as defined by Table 3.1d.

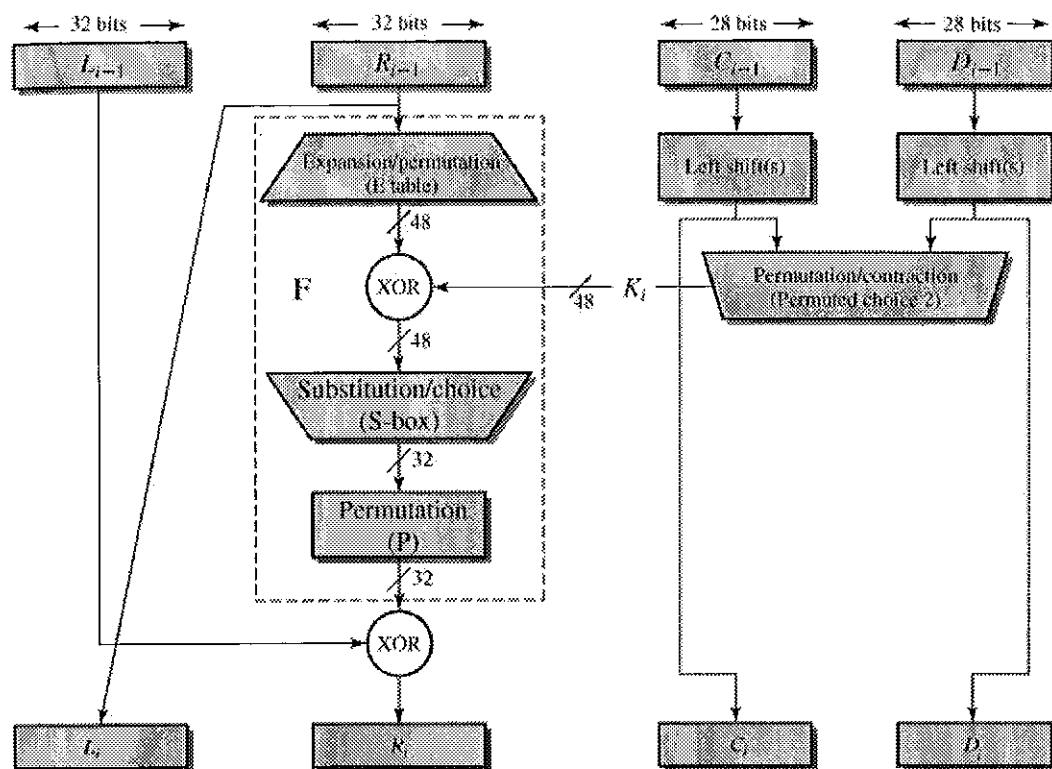


Figure 3.3 Single Round of DES Algorithm

The role of the S-boxes in the function F is illustrated in Figure 3.4. The substitution consists of a set of eight S-boxes, each of which accepts 6 bits as input and produces 4 bits as output. These transformations are defined in Table 3.2, which is interpreted as follows: The first and last bits of the input to box S_i form a 2-bit binary number to select one of four substitutions defined by the four rows in the table for S_i . The middle four bits select one of the sixteen columns. The decimal value in the cell selected by the row and column is then converted to its 4-bit representation to produce the output. For example, in S_1 for input 011001, the row is 01 (row 1) and the column is 1100 (column 12). The value in row 1, column 12 is 9, so the output is 1001.

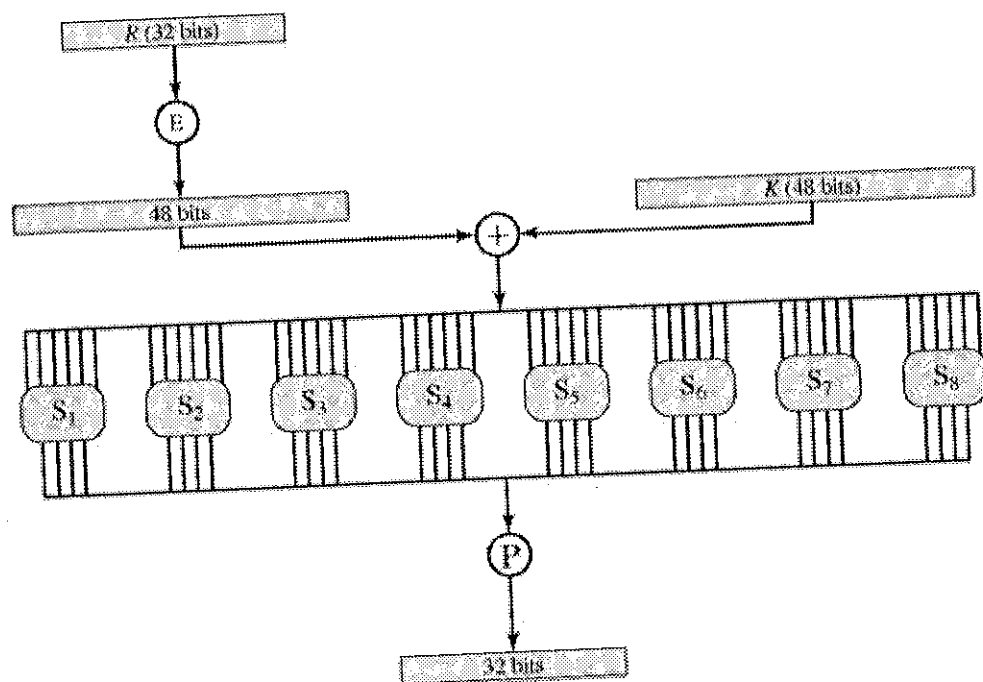


Figure 3.4 Calculation of $F(R, K)$

S_1	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S_3	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S_4	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S_5	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S_6	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S_7	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S_8	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Table 3.2 Definition of DES S-Boxes

Key Generation

Returning to Figures 3.2 and 3.52, we see that a 64-bit key is used as input to the algorithm. The bits of the key are numbered from 1 through 64; every eighth bit is ignored, as indicated by the lack of shading in Table 3.1a. The key is first subjected to a permutation governed by a table labeled Permuted Choice One (Table 3.b). The resulting 56-bit key is then treated as two 28-bit quantities, labeled C_0 and D_0 . At each round, C_{i-1} and D_{i-1} are separately subjected to a circular left shift, or rotation, of 1 or 2 bits, as governed by Table 3.3d. These shifted values serve as input to the next round. They also serve as input to Permuted Choice Two (Table 3.3c), which produces a 48-bit output that serves as input to the function $F(R_{i-1}, K_i)$.

(a) Input Key																
1	2	3	4	5	6	7	8									
9	10	11	12	13	14	15	16									
17	18	19	20	21	22	23	24									
25	26	27	28	29	30	31	32									
33	34	35	36	37	38	39	40									
41	42	43	44	45	46	47	48									
49	50	51	52	53	54	55	56									
57	58	59	60	61	62	63	64									
(b) Permuted Choice One (PC-1)																
57	49	41	33	25	17	9										
1	58	50	42	34	26	18										
10	2	59	51	43	35	27										
19	11	3	60	52	44	36										
63	55	47	39	31	23	15										
7	62	54	46	38	30	22										
14	6	61	53	45	37	29										
21	13	5	28	20	12	4										
(c) Permuted Choice Two (PC-2)																
14	17	11	24	1	5	3	28									
15	6	21	10	23	19	12	4									
26	8	16	7	27	20	13	2									
41	52	31	37	47	55	30	40									
51	45	33	48	44	49	39	56									
34	53	46	42	50	36	29	32									
(d) Schedule of Left Shifts																
Round number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bits rotated	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Table 3.3 DES Key Schedule Calculation

3.1.2 Rijndael (AES)

Many experts now feel that DES is near the end of its useful life. In the late 1970s, a 56-bit key would have been viewed by many as sufficient for securing most sensitive commercial data, given the cost and speed of computers available at that time. Unfortunately, this is no longer quite so true, since computers are now much cheaper and faster. In general, you should always use encryption technology that will incur sufficient cost or time for potential attackers to consider it not worthwhile mounting an attack. With the doubling of digital circuit-switching speeds every 18 months, progress made in cryptanalysis research, along with the advent of inexpensive, customizable gate-array hardware and the massive distributed computing model made possible by the Internet, times have certainly changed. Triple DES was a very capable alternative, but it was only intended to be a temporary fix. A replacement for DES has recently gone through an international competition, and the official DES replacement, formerly referred to as AES (Advanced Encryption Standard), is the Rijndael algorithm. The Rijndael specification can be found in FIPS-197, and, like its predecessor, it is intended for use by U.S. government organizations to protect unclassified sensitive information. It is expected that, over time, the Rijndael algorithm will have a huge impact on commercial software security, especially in financial applications.

Unlike the DES algorithm, which uses a fixed 56-bit key size, the Rijndael algorithm is more flexible in that it is able to use key sizes of 128, 192, or 256 bits (referred to as AES-128, AES-192, and AES-256, respectively). Whereas the data block size of DES was fixed at 64 bits, Rijndael can work with 128, 192, or 256 bit data blocks. Rijndael was originally designed to accommodate other key and block sizes, but the AES standard ignores these alternatives. The number of rounds in Rijndael depends on the block and key size. Nine rounds are used if the block and the key sizes are both 128 bits. Eleven rounds are used if either the block or the key size is greater than 128 bits but neither is greater than 192 bits. Thirteen rounds are used if either the block or key size is 256 bits.

An additional modified final round is added to the end of the algorithm, so 10, 12, or 14 rounds are actually performed.

The Rijndael algorithm is much more complex than DES, which was originally designed for simple and efficient digital-circuit hardware implementations. Rijndael involves advanced mathematical concepts, such as algebra on polynomials with coefficients in the Galois field $GF(2^8)$, where the familiar concepts of addition and multiplication take on bizarre new meanings. These details are beyond the scope of this book; however, you are encouraged to read the FIPS-197 document for these details. Fortunately, you do not need to understand these gruesome details to use the Rijndael algorithm in your own .NET programs, since the .NET Framework hides the tough stuff quite nicely!

The input to the AES encryption and decryption algorithms is a single 128-bit block, depicted in FIPS PUB 197, as a square matrix of bytes. This block is copied into the State array, which is modified at each stage of encryption or decryption. After the final stage, State is copied to an output.

The key is expanded into 44/52/60 lots of 32-bit words (see later), with 4 used in each round.

The data computation then consists of an “add round key” step, then 9/11/13 rounds with all 4 steps, and a final 10th/12th/14th step of byte subs + mix cols + add round key. This can be viewed as alternating XOR key & scramble data bytes operations. All of the steps are easily reversed, and can be efficiently implemented using XOR’s & table lookups.

AES Parameters

Key size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext block size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of rounds	10	12	14
Round key size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded key size (words/bytes)	44/176	52/208	60/240

Table 3.3 AES Parameters

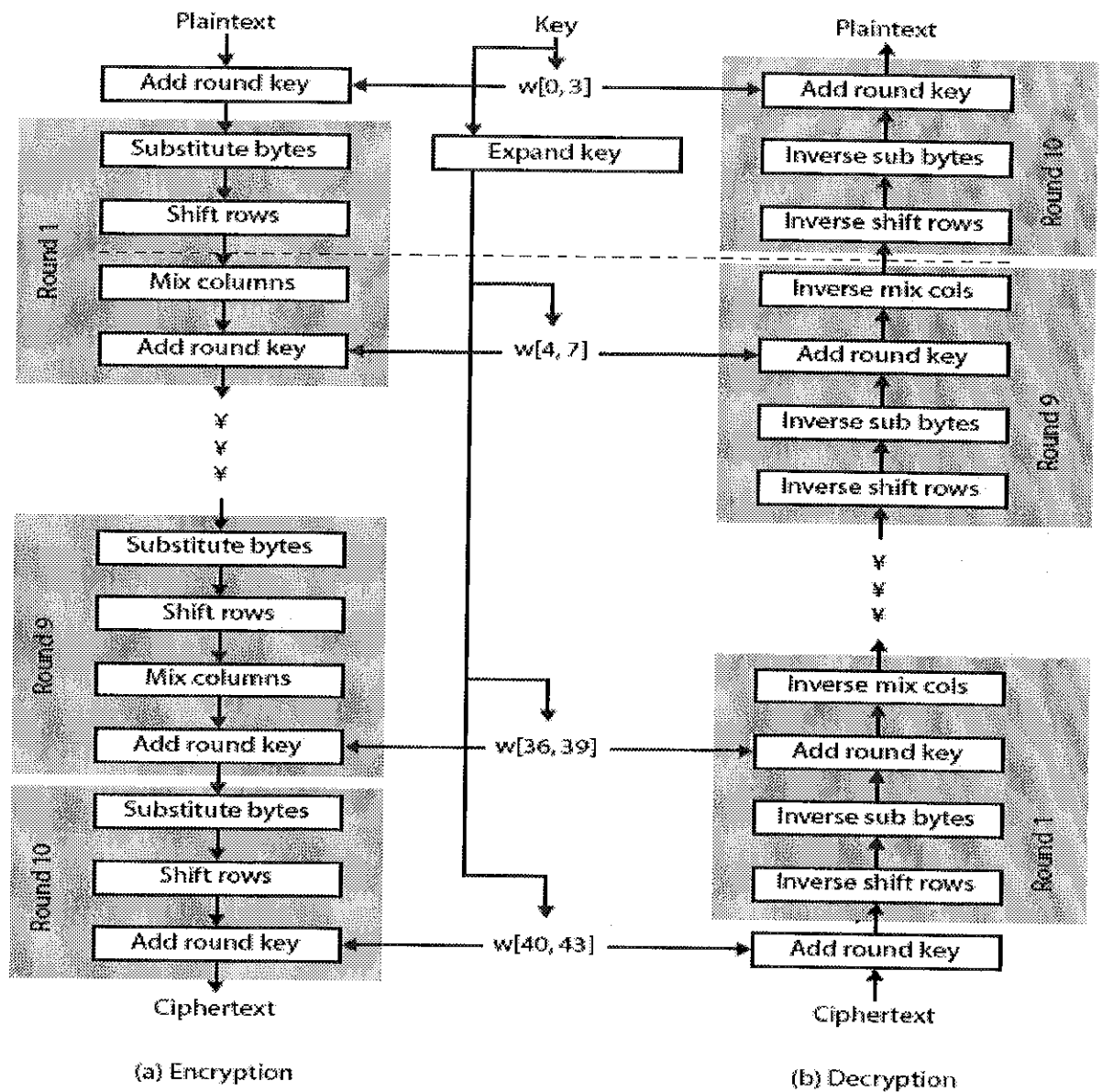
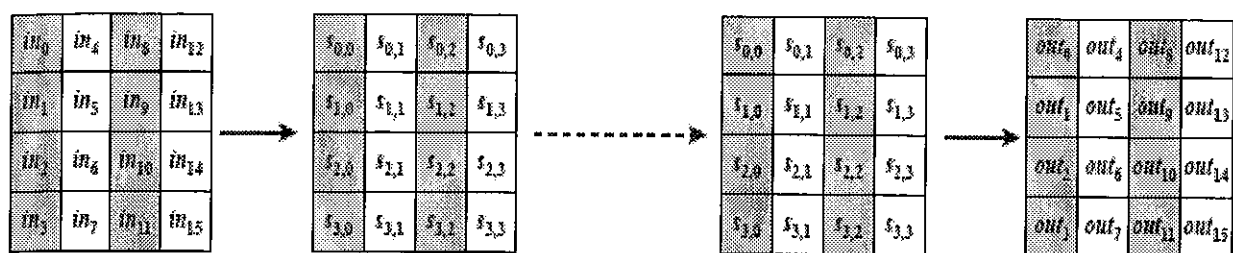
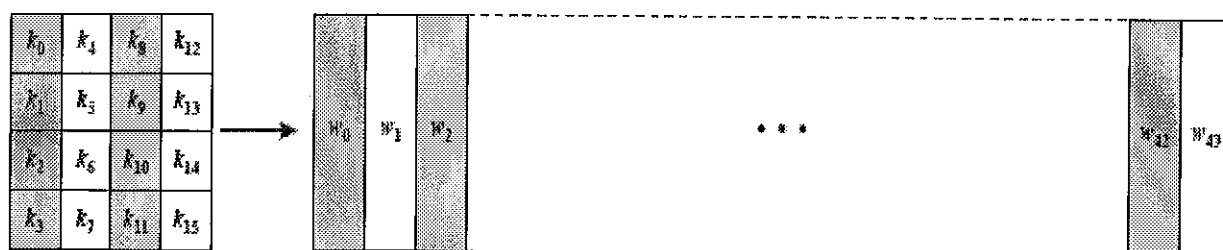


Figure 3.5 Flow chart of AES enc. And Dec.

AES Data Structure



(a) Input, state array, and output



(b) Key and expanded key

Figure 3.6 AES data structure

Byte Substitution

The Substitute bytes stage uses an S-box to perform a byte-by-byte substitution of the block. There is a single 8-bit wide S-box used on every byte. This S-box is a permutation of all 256 8-bit values, constructed using a transformation which treats the values as polynomials in $GF(2^8)$ – however it is fixed, so really only need to know the table when implementing. Decryption requires the inverse of the table. These tables are given in Stallings Table 4.5.

The table was designed to be resistant to known cryptanalytic attacks. Specifically, the Rijndael developers sought a design that has a low correlation between input bits and output bits, with the property that the output cannot be described as a simple mathematical function of the input, with no fixed points and no “opposite fixed points”.

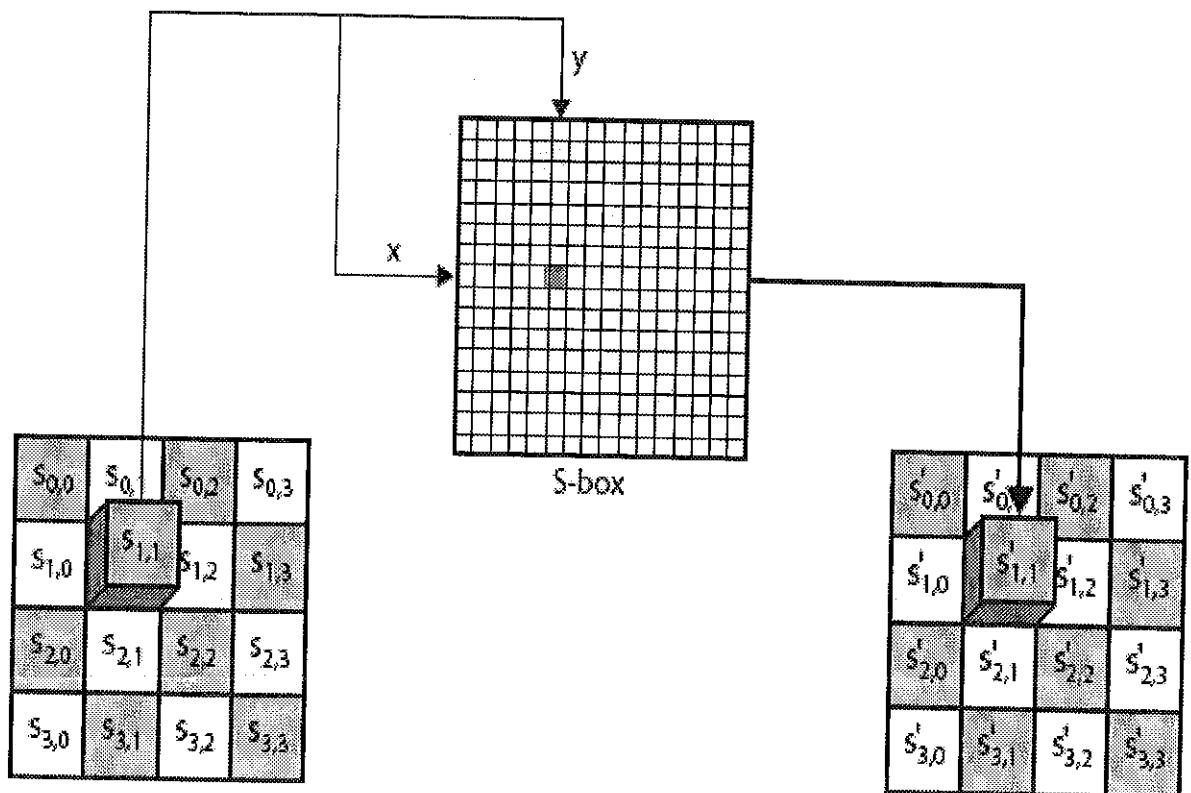


Figure 3.7 Byte Substitution

Shift Rows

The ShiftRows stage provides a simple “permutation” of the data, whereas the other steps involve substitutions. Further, since the state is treated as a block of columns, it is this step which provides for diffusion of values between columns. It performs a circular rotate on each row of 0, 1, 2 & 3 places for respective rows. When decrypting it performs the circular shifts in the opposite direction for each row. This row shift moves an individual byte from one column to another, which is a linear distance of a multiple of 4 bytes, and ensures that the 4 bytes of one column are spread out to four different columns.

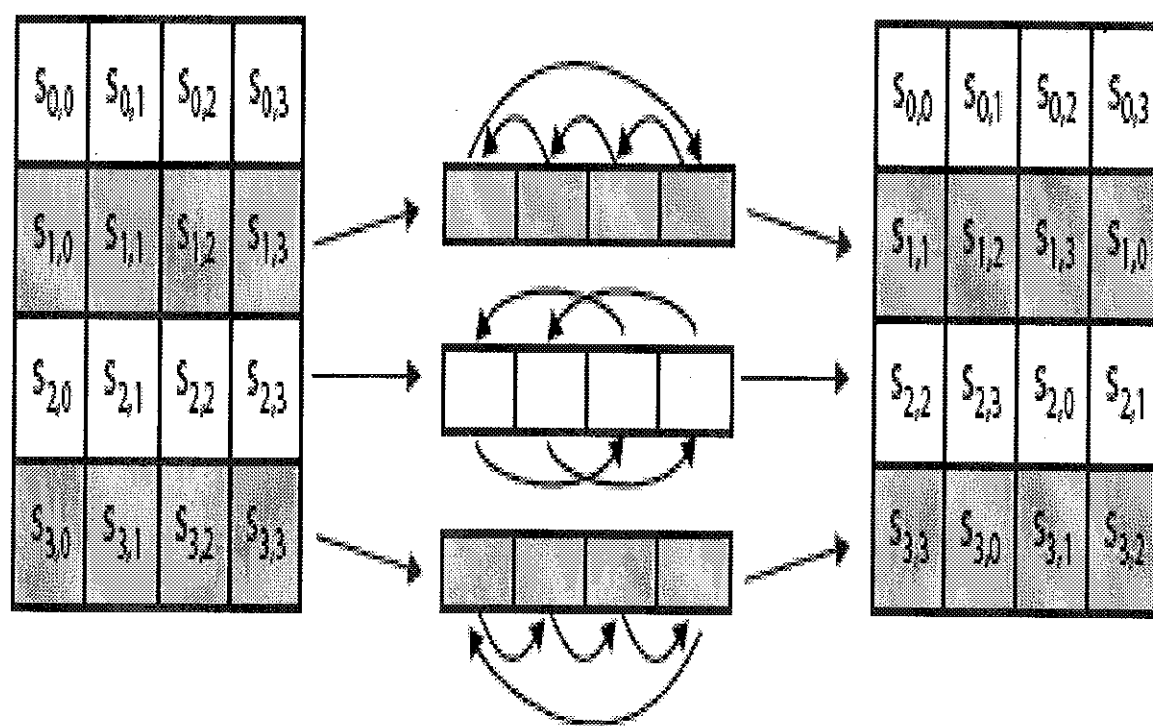


Fig 3.8 Shift Row

Mix Column

The MixColumns stage is a substitution that makes use of arithmetic over $GF(2^8)$. Each byte of a column is mapped into a new value that is a function of all four bytes in that column. It is designed as a matrix multiplication where each byte is treated as a polynomial in $GF(2^8)$. The inverse used for decryption involves a different set of constants.

The constants used are based on a linear code with maximal distance between code words – this gives good mixing of the bytes within each column. Combined with the “shift rows” step provides good avalanche, so that within a few rounds, all output bits depend on all input bits.

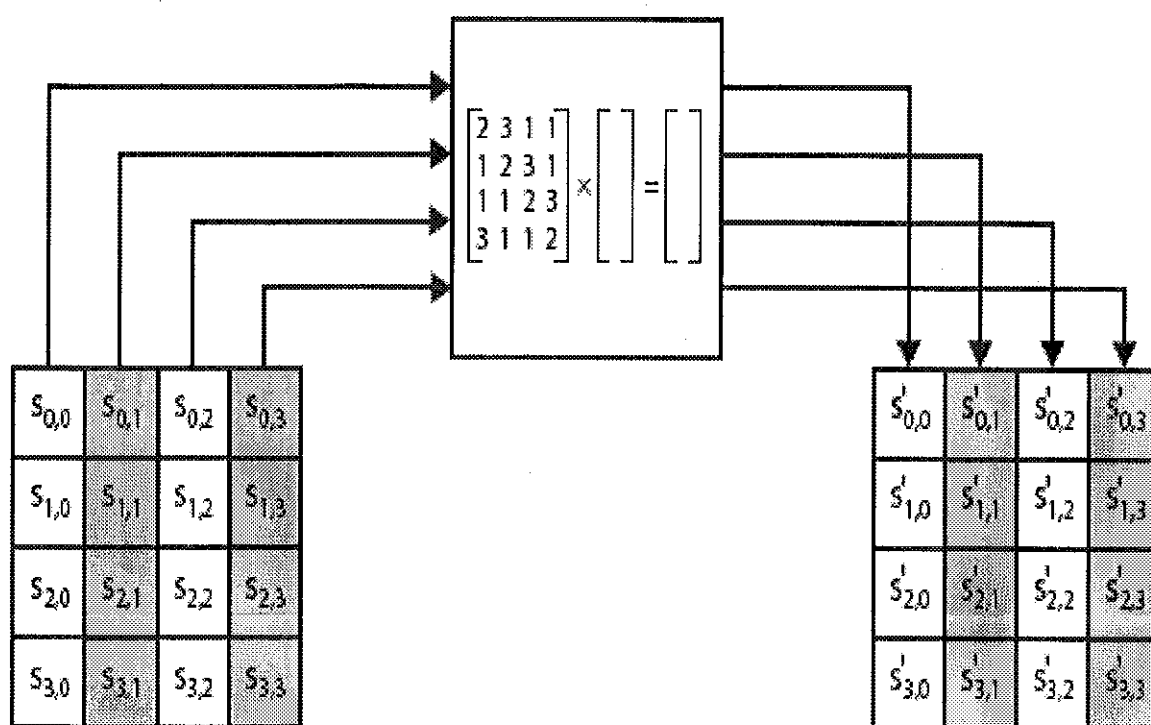


Figure 3.9 Mix Column

Add Round Key

Lastly is the Add Round Key stage which is a simple bitwise XOR of the current block with a portion of the expanded key. Note this is the only step which makes use of the key and obscures the result, hence MUST be used at start and end of each round, since otherwise could undo effect of other steps. But the other steps provide confusion/diffusion/non-linearity. That us you can look at the cipher as a series of XOR with key then scramble/permute block repeated. This is efficient and highly secure it is believed.

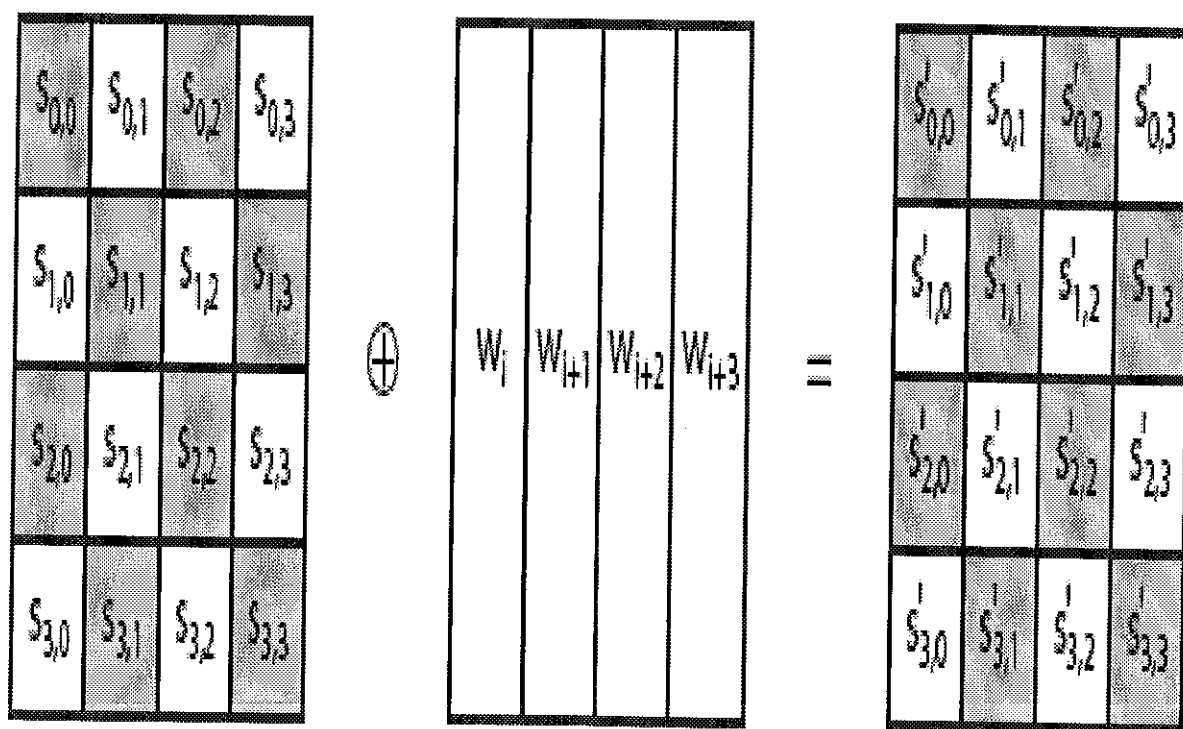


Figure 3.10 Add Round Key

AES Key Expansion

The AES key expansion algorithm takes as input a 4-word (16-byte) key and produces a linear array of words, providing a 4-word round key for the initial AddRoundKey stage and each of the 10/12/14 rounds of the cipher. It involves copying the key into the first group of 4 words, and then constructing subsequent groups of 4 based on the values of the previous & 4th back words. The first word in each group of 4 gets “special treatment” with rotate + S-box + XOR constant on the previous word before XOR’ing the one from 4 back. In the 256-bit key/14 round version, there’s also an extra step on the middle word.

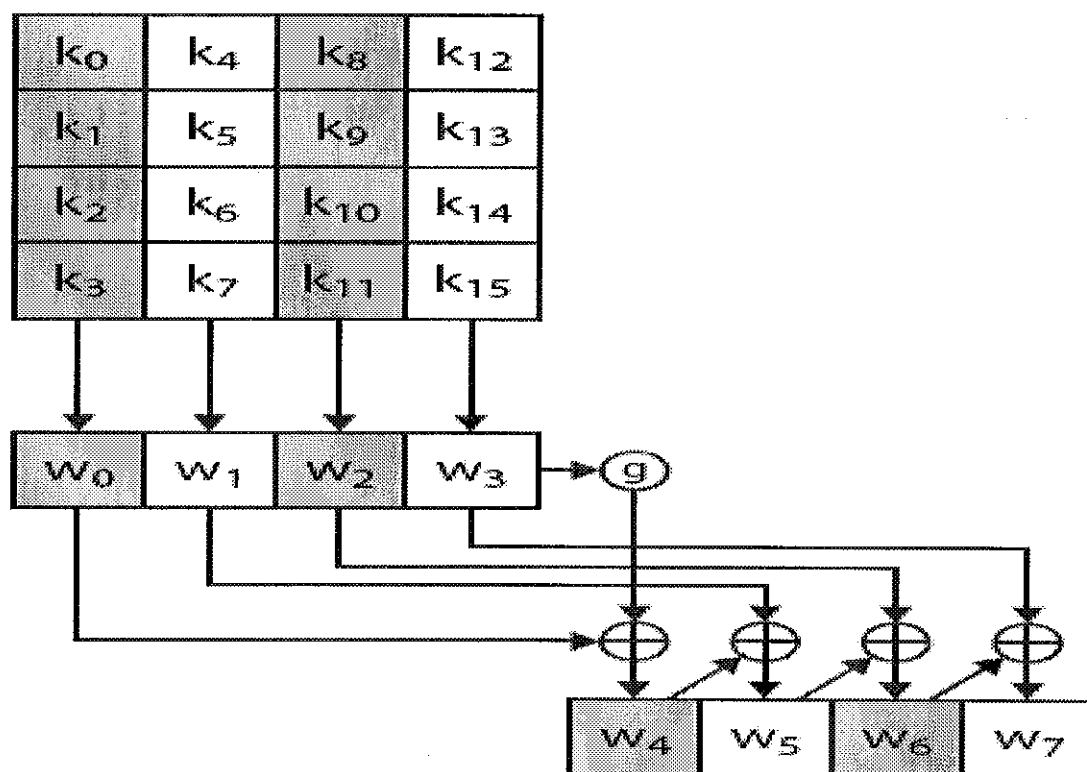


Fig 3.11 key Expansion

AES Round

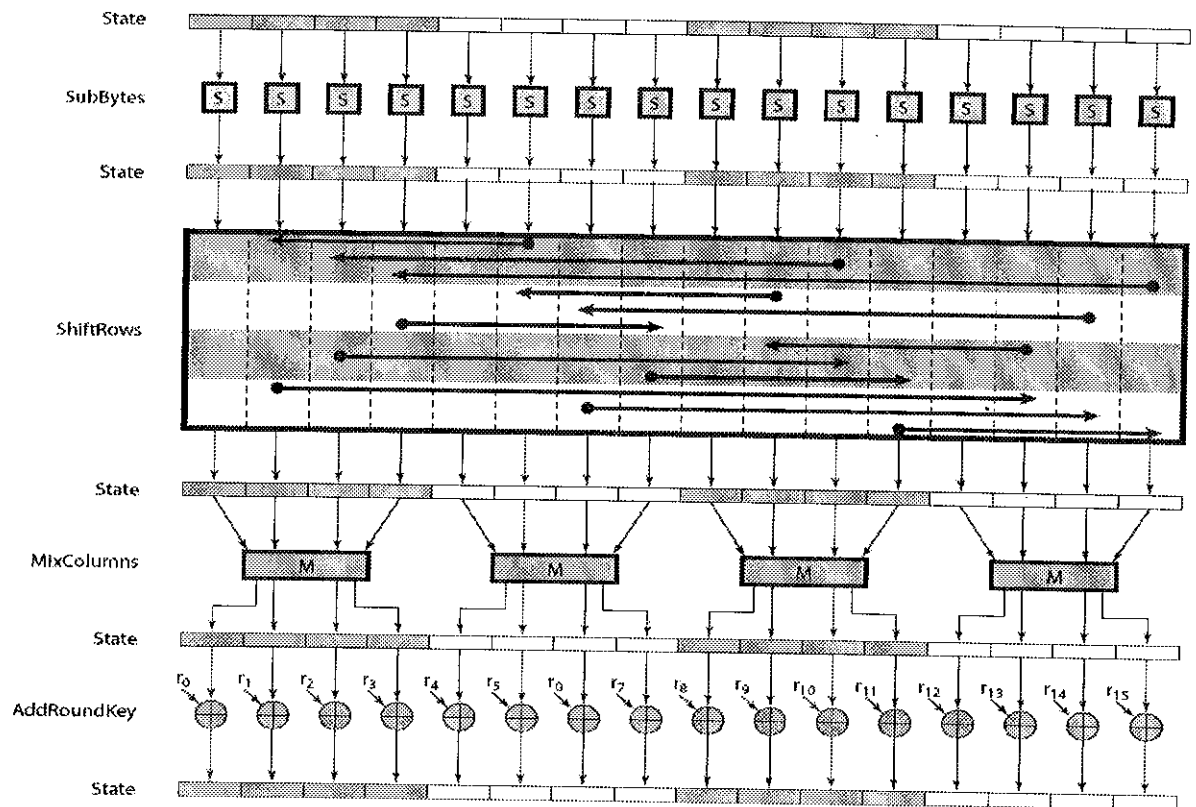


Fig 3.12 AES Round

AES Decryption

The AES decryption cipher is not identical to the encryption cipher. The sequence of transformations for decryption differs from that for encryption, although the form of the key schedules for encryption and decryption is the same. This has the disadvantage that two separate software or firmware modules are needed for applications that require both encryption and decryption. There is, however, an equivalent version of the decryption algorithm that has the same structure as the encryption algorithm, with the same sequence of transformations as the encryption algorithm (with transformations replaced by their inverses). To achieve this equivalence, a change in key schedule is needed.

By constructing an equivalent inverse cipher with steps in same order as for encryption, we can derive a more efficient implementation. Clearly swapping the byte substitutions and shift rows has no effect, since work just on bytes. Swapping the mix columns and add round key steps requires the inverse mix columns step be applied to the round keys first – this makes the decryption key schedule a little more complex with this construction, but allows the use of same h/w or s/w for the data en/decrypt computation.

3.1.3 Triple -DES (3-DES)

The banking industry has adopted the ANSI X9.52 standard, also known as Triple DES, TDES, or 3DES. Triple DES was used as a more powerful interim alternative to DES. Triple DES is based directly on the basic design of DES and is quite backward-compatible with the original DES design and modes. The improvement comes from the fact that each 64-bit block of plaintext is encrypted three times, using the DES algorithm in which three distinct keys are used. Triple DES encrypts each block with the first key, then decrypts the result using the second key, and finally encrypts again using the third key. Triple DES increases the effective key size of DES, but the algorithm takes approximately three times as much time to compute compared to DES.

Triple DES encryption

$$C = E_{k_3}(D_{k_2}(E_{k_1}(M)))$$

Triple DES decryption

$$M = D_{k_1}(E_{k_2}(D_{k_3}(C))),$$

where k_1, k_2, k_3 are the three 56-bit keys.

3.1.4 RC2

RC2 is a registered trademark of RSA Data Security, Incorporated. The RC2 algorithm is a symmetric block cipher designed by Ronald Rivest in 1987, using a 64-bit input data block size. It was designed to be a drop-in replacement for DES, with improved performance, and a variable key size (from one byte up to 128 bytes) that allows for fine-tuning of its cryptographic strength. Effectively, RC2 can be made more or less secure than DES simply by choosing an appropriate key size. One of the strengths of RC2 is that it is designed to be about twice as fast as DES in typical software implementations. According to the folks at RSA Security, RC stands for either Ron's Code or Rivest's Cipher; however, it does not seem all that clear that they have settled the official story on this!

RC2 is licensed for use in several commercial Internet software packages, including Lotus Notes, Microsoft Internet Explorer, Outlook Express, and Netscape Communicator. S/MIME (Secure/Multipurpose Internet Mail Extensions) allows for the choice of using RC2 (along with DES and Triple DES) to provide interoperable, secure email. The security services supported by RC2 in these applications are authentication via digital signatures and privacy via encryption.

RC2 has been submitted to the IETF organization as a Request For Comment. For the detailed RC2 specification, see the RFC 2268 at <http://www.ietf.org/rfc/rfc2268.txt>.

3.2 Asymmetric Algorithms

There are several asymmetric algorithms in existence today, including RSA, DSA, ElGamal, and ECC. Currently, the most popular is RSA, which stands for Rivest, Shamir, and Adelman, the names of its inventors. RSA is based on the problem of factoring large composite numbers into prime factors. RSA can be used for confidentiality or symmetric key exchange as well as for digital signatures. DSA, which was proposed by NIST in 1991, stands for Digital Signature Algorithm. DSA is somewhat less flexible, since it can be used for digital signatures but not for confidentiality or symmetric key exchange. The ElGamal algorithm, which was invented by Taher ElGamal, is based on the problem of calculating the discrete logarithm in a finite field. ECC stands for Elliptic Curve Cryptography, which was independently proposed in 1985 by Neal Koblitz and V. S. Miller. ECC is not actually an algorithm, but an alternate algebraic system for implementing algorithms, such as DSA, using peculiar mathematical objects known as elliptic curves over finite fields. ElGamal and ECC are not currently supported by .NET out of the box; however, the .NET Framework has been designed to be extensible, making it possible for you or other vendors to provide implementations.

Some asymmetric algorithms, such as RSA and ElGamal, can be used for both encryption and digital signatures. Other asymmetric algorithms, such as DSA, are useful only for implementing digital signatures. It is also generally true that asymmetric algorithms tend to be much slower and less secure than symmetric algorithms for a comparable key size. To be effective, asymmetric algorithms should be used with a larger key size, and, to achieve acceptable performance, they are most applicable to small data sizes. Therefore, asymmetric algorithms are usually used to encrypt hash values and symmetric session keys, both of which tend to be rather small in size compared to typical plaintext data.

3.2.1 RSA

The most common asymmetric cipher currently in use is RSA, which is fully supported by the .NET Security Framework. Ron Rivest, Adi Shamir, and Leonard Adleman invented the RSA cipher in 1978 in response to the ideas proposed by Hellman, Diffie, and Merkel. Later in this chapter, we shall see how to use the high-level implementation of RSA provided by the .NET Security Framework. But first, let's look at how RSA works at a conceptual level.

Here is how RSA works. First, we randomly generate a public and private key pair. As is always the case in cryptography, it is very important to generate keys in the most random and therefore, unpredictable manner possible. Then, we encrypt the data with the public key, using the RSA algorithm. Finally, we decrypt the encrypted data with the private key and verify that it worked by comparing the result with the original data. Note that we are encrypting with the public key and decrypting with the private key. This achieves confidentiality. In the next chapter, we look at the flip side of this approach, encrypting with the private key and decrypting with the public key, to achieve authentication and integrity checking.

Here are the steps for generating the public and private key pair.

1. Randomly select two prime numbers p and q . For the algebra to work properly, these two primes must not be equal. To make the cipher strong, these prime numbers should be large, and they should be in the form of arbitrary precision integers with a size of at least 1024 bits.
2. Calculate the product: $n = p \cdot q$.
3. Calculate the Euler totient for these two primes, which is represented by the Greek letter ϕ . This is easily computed with the formula $\phi = (p - 1) \cdot (q - 1)$.

4. Now that we have the values n and ϕ , the values p and q will no longer be useful to us. However, we must ensure that nobody else will ever be able to discover these values. Destroy them, leaving no trace behind so that they cannot be used against us in the future. Otherwise, it will be very easy for an attacker to reconstruct our key pair and decipher our ciphertext.

5. Randomly select a number e (the letter e is used because we will use this value during encryption) that is greater than 1, less than ϕ , and relatively prime to ϕ . Two numbers are said to be relatively prime if they have no prime factors in common. Note that e does not necessarily have to be prime. The value of e is used along with the value n to represent the public key used for encryption.

6. Calculate the unique value d (to be used during decryption) that satisfies the requirement that, if $d \cdot e$ is divided by ϕ , then the remainder of the division is 1. The mathematical notation for this is $d \cdot e = 1(\text{mod } \phi)$. In mathematical jargon, we say that d is the multiplicative inverse of e modulo ϕ . The value of d is to be kept secret. If you know the value of ϕ , the value of d can be easily obtained from e using a technique known as the Euclidean algorithm. If you know n (which is public), but not p or q (which have been destroyed), then the value of ϕ is very hard to determine. The secret value of d together with the value n represents the private key.

Once we have generated a public/private key pair, we can encrypt a message with the public key with the following steps.

1. Take a positive integer m to represent a piece of plaintext message. In order for the algebra to work properly, the value of m must be less than the modulus n , which was originally computed as $p \cdot q$. Long messages must therefore be broken into small enough pieces that each piece can be uniquely represented by an integer of this bit size, and each piece is then individually encrypted.

2. Calculate the ciphertext c using the public key containing e and n . This is calculated using the equation $c = m^e (\text{mod } n)$.

Finally, we can perform the decryption procedure with the private key using the following steps.

1. Calculate the original plaintext message from the ciphertext using the private key containing d and n . This is calculated using the equation $m = c^d \pmod{n}$.
2. Compare this value of m with the original m , and you should see that they are equal, since decryption is the inverse operation to encryption.

3.2.2 DSA

DSA is a NIST federal standard, used along with the Secure Hash Standard, to attach digital signatures to data. NIST published the first version of the DSA algorithm as part of the DSS (Digital Signature Standard, FIPS 186) in May 1994. DSA is based on the discrete logarithm problem, which is described in the next two sections.

This description is intended only as an outline of the scheme, which can be used as a basis for an actual DSA implementation. From these details, you can see that the implementation of DSA is not trivial, and we are rather fortunate that cryptographic libraries, such as the .NET Framework, provide this implementation for us.

GENERATING THE KEY

1. Choose a prime p with a bit size of 512, 576, 640, 704, 768, 832, 896, 960, or 1024.
2. Choose a 160-bit prime q that evenly divides $p - 1$.
3. Choose numbers g and h such that $g = h^{((p-1)/q)} \bmod p > 1$ and $1 < h < p - 1$.
4. Randomly choose a private key x in the range $0 < x < q$.
5. Calculate $y = g^x \bmod p$.
6. The resulting public key is the set of numbers (p, q, g, y) .
7. The resulting private key is the number x .

SIGNING THE MESSAGE

1. Obtain the plaintext message m to be signed.
2. Choose a random value s such that $1 < s < q$.
3. Calculate $r = (g^s \bmod p) \bmod q$.
4. Calculate $s = (H(m) - r * x) s^{-1} \bmod q$, where $H(m)$ is the SHA-1 algorithm.
5. The resulting signature is the set of numbers (r, s) .

VERIFYING THE SIGNATURE

1. Calculate $w = (s)^{-1} \pmod{q}$.
2. Calculate $u_1 = H(m) * w \pmod{q}$.
3. Calculate $u_2 = r * w \pmod{q}$.
4. Calculate $v = (g^{u_1} * y^{u_2} \pmod{p}) \pmod{q}$.
5. The signature is valid if and only if $v = r$.

CHAPTER – 4

PEER TO PEER COMMUNICATION

4.1 INTRODUCTION - COMPUTER NETWORK

What is a socket?

A socket is an object that represents a low-level access point to the IP stack. This socket can be open or closed or one of a set number of intermediate states. A socket can send and receive data down this connection. Data is generally sent in blocks of a few kilobytes at a time for efficiency; each of these blocks is called a packet.

All packets that travel on the Internet must use the Internet protocol. This means that the source IP address, destination address must be included in the packet. Most packets also contain a port number. A port is simply a number between 1 and 65,535 that is used to differentiate higher protocols, such as email or FTP (Table 3.1). Ports are important when it comes to programming your own network applications because no two applications can use the same port. It is recommended that experimental programs use port numbers above 1024.

Packets that contain port numbers come in two flavors: UDP and TCP/IP. UDP has lower latency than TCP/IP, especially on startup. Where data integrity is not of the utmost concern, UDP can prove easier to use than TCP, but it should never be used where data integrity is more important than performance; however, data sent via UDP can sometimes arrive in the wrong order and be effectively useless to the receiver. TCP/IP is more complex than UDP and has generally longer latencies, but it does guarantee that data does not become corrupted when traveling over the Internet. TCP is ideal for file transfer, where a corrupt file is more unacceptable than a slow download; however, it is unsuited to Internet radio, where the odd sound out of place is more acceptable than long gaps of silence.

Using TCP/IP to transfer files

Most networked applications use TCP/IP because there is no risk of data becoming corrupted while traveling over the Internet. It is said to be connection oriented; that is, both client and server after a setup phase treat a set of IP packets as being sent along a virtual channel, allowing for data that is too large to fit into a single IP packet to be sent and for retransmission to occur when packets are lost.

This sample application will allow you to send any file from one computer to another. Again, it is client/server based, so you will need either two computers or to run both the client and server on the same computer.

Socket-level networking in .NET

It is often necessary to understand network code written by other developers in order to debug it or adapt it to your own application. After all, no program is ever written without referring to some existing code. This book will consistently use the most concise code possible, but it is important to realize that there are many techniques to implement networked applications in .NET. It is equally important to be able to understand and recognize these techniques when they are used in code written by other developers.

The most important class in .NET networking is the Socket class. This can be used for either TCP/IP or UDP as either a client or server; however, it requires the help of the Dns class to resolve IP addresses and is quite difficult to use. Three other classes exist, which are simpler to use, but less flexible: TcpListener, TcpClient, and UdpClient. To illustrate the differences between the two techniques, listed below is code that demonstrates how a socket can be made to listen for incoming connections on port 8080 and display any received data on screen.

Socket-level programming is the foundation of all network programming. This chapter should provide enough information to assist you in implementing any TCP- or UDP-based protocol, proprietary or otherwise. Not all network protocols need to be coded at the socket level; extensive support for HTTP is provided through classes provided by the .NET framework. Leveraging this ready-made functionality can cut down on the development time required for socket-level implementation.

Network Hardware

It is now time to turn our attention from the applications and social aspects of networking (the fun stuff) to the technical issues involved in network design (the work stuff). There is no generally accepted taxonomy into which all computer networks fit, but two dimensions stand out as important: transmission technology and scale. We will now examine each of these in turn.

Broadly speaking, there are two types of transmission technology that are in widespread use. They are as follows:

1. Broadcast links.
2. Point-to-point links.

Broadcast networks have a single communication channel that is shared by all the machines on the network. Short messages, called packets in certain contexts, sent by any machine are received by all the others. An address field within the packet specifies the intended recipient. Upon receiving a packet, a machine checks the address field. If the packet is intended for the receiving machine, that machine processes the packet; if the packet is intended for some other machine, it is just ignored.

As an analogy, consider someone standing at the end of a corridor with many rooms off it and shouting "Watson, come here. I want you." Although the packet may actually be received (heard) by many people, only Watson responds. The others just ignore it. Another analogy is an airport announcement asking all flight 644 passengers to report to gate 12 for immediate boarding.

Broadcast systems generally also allow the possibility of addressing a packet to all destinations by using a special code in the address field. When a packet with this code is transmitted, it is received and processed by every machine on the network. This mode of operation is called broadcasting. Some broadcast systems also support transmission to a subset of the machines, something known as multicasting. One possible scheme is to reserve one bit to indicate multicasting. The remaining $n - 1$ address bits can hold a group number. Each machine can "subscribe" to any or all of the groups. When a packet is sent to a certain group, it is delivered to all machines subscribing to that group.

In contrast, point-to-point networks consist of many connections between individual pairs of machines. To go from the source to the destination, a packet on this type of network may have to first visit one or more intermediate machines. Often multiple routes, of different lengths, are possible, so finding good ones is important in point-to-point networks. As a general rule (although there are many exceptions), smaller, geographically localized networks tend to use broadcasting, whereas larger networks usually are point-to-point. Point-to-point transmission with one sender and one receiver is sometimes called unicasting.

An alternative criterion for classifying networks is their scale. At the top are the personal area networks, networks that are meant for one person. For example, a wireless network connecting a computer with its mouse, keyboard, and printer is a personal area network. Also, a PDA that controls the user's hearing aid or pacemaker fits in this category. Beyond the personal area networks come longer-range networks. These can be divided into local, metropolitan, and wide area networks. Finally, the connection of two or more networks is called an internetwork. The worldwide Internet is a well-known example of an internetwork. Distance is important as a classification metric because different techniques are used at different scales. In this book we will be concerned with networks at all these scales. Below we give a brief introduction to network hardware.

Local Area Networks

Local area networks, generally called LANs, are privately-owned networks within a single building or campus of up to a few kilometers in size. They are widely used to connect personal computers and workstations in company offices and factories to share resources (e.g., printers) and exchange information. LANs are distinguished from other kinds of networks by three characteristics: (1) their size, (2) their transmission technology, and (3) their topology.

LANs are restricted in size, which means that the worst-case transmission time is bounded and known in advance. Knowing this bound makes it possible to use certain kinds of designs that would not otherwise be possible. It also simplifies network management.

LANs may use a transmission technology consisting of a cable to which all the machines are attached, like the telephone company party lines once used in rural areas. Traditional LANs run at speeds of 10 Mbps to 100 Mbps, have low delay (microseconds or nanoseconds), and make very few errors. Newer LANs operate at up to 10 Gbps. In this book, we will adhere to tradition and measure line speeds in megabits/sec (1 Mbps is 1,000,000 bits/sec) and gigabits/sec (1 Gbps is 1,000,000,000 bits/sec).

Various topologies are possible for broadcast LANs. In a bus (i.e., a linear cable) network, at any instant at most one machine is the master and is allowed to transmit. All other machines are required to refrain from sending. An arbitration mechanism is needed to resolve conflicts when two or more machines want to transmit simultaneously. The arbitration mechanism may be centralized or distributed. IEEE 802.3, popularly called Ethernet, for example, is a bus-based broadcast network with decentralized control, usually operating at 10 Mbps to 10 Gbps. Computers on an Ethernet can transmit whenever they want to; if two or more packets collide, each computer just waits a random time and tries again later.

A second type of broadcast system is the ring. In a ring, each bit propagates around on its own, not waiting for the rest of the packet to which it belongs. Typically, each bit

circumnavigates the entire ring in the time it takes to transmit a few bits, often before the complete packet has even been transmitted. As with all other broadcast systems, some rule is needed for arbitrating simultaneous accesses to the ring. Various methods, such as having the machines take turns, are in use. IEEE 802.5 (the IBM token ring), is a ring-based LAN operating at 4 and 16 Mbps. FDDI is another example of a ring network.

Broadcast networks can be further divided into static and dynamic, depending on how the channel is allocated. A typical static allocation would be to divide time into discrete intervals and use a round-robin algorithm, allowing each machine to broadcast only when its time slot comes up. Static allocation wastes channel capacity when a machine has nothing to say during its allocated slot, so most systems attempt to allocate the channel dynamically (i.e., on demand).

Dynamic allocation methods for a common channel are either centralized or decentralized. In the centralized channel allocation method, there is a single entity, for example, a bus arbitration unit, which determines who goes next. It might do this by accepting requests and making a decision according to some internal algorithm. In the decentralized channel allocation method, there is no central entity; each machine must decide for itself whether to transmit. You might think that this always leads to chaos, but it does not. Later we will study many algorithms designed to bring order out of the potential chaos.

Metropolitan Area Networks

A metropolitan area network, or MAN, covers a city. The best-known example of a MAN is the cable television network available in many cities. This system grew from earlier community antenna systems used in areas with poor over-the-air television reception. In these early systems, a large antenna was placed on top of a nearby hill and signal was then piped to the subscribers' houses.

At first, these were locally-designed, ad hoc systems. Then companies began jumping into the business, getting contracts from city governments to wire up an entire city. The next step was television programming and even entire channels designed for cable only.

Often these channels were highly specialized, such as all news, all sports, all cooking, all gardening, and so on. But from their inception until the late 1990s, they were intended for television reception only.

Starting when the Internet attracted a mass audience, the cable TV network operators began to realize that with some changes to the system, they could provide two-way Internet service in unused parts of the spectrum. At that point, the cable TV system began to morph from a way to distribute television to a metropolitan area network. In this figure we see both television signals and Internet being fed into the centralized head end for subsequent distribution to people's homes.

Wide Area Networks

A wide area network, or WAN, spans a large geographical area, often a country or continent. It contains a collection of machines intended for running user (i.e., application) programs. We will follow traditional usage and call these machines hosts. The hosts are connected by a communication subnet, or just subnet for short. The hosts are owned by the customers (e.g., people's personal computers), whereas the communication subnet is typically owned and operated by a telephone company or Internet service provider. The job of the subnet is to carry messages from host to host, just as the telephone system carries words from speaker to listener. Separation of the pure communication aspects of the network (the subnet) from the application aspects (the hosts), greatly simplifies the complete network design.

In most wide area networks, the subnet consists of two distinct components: transmission lines and switching elements. Transmission lines move bits between machines. They can be made of copper wire, optical fiber, or even radio links. Switching elements are specialized computers that connect three or more transmission lines. When data arrive on an incoming line, the switching element must choose an outgoing line on which to forward them. These switching computers have been called by various names in the past; the name router is now most commonly used. Unfortunately, some people pronounce it "rooter" and others have it rhyme with "doubter." Determining the correct pronunciation

will be left as an exercise for the reader. (Note: the perceived correct answer may depend on where you live.)

In this model each host is frequently connected to a LAN on which a router is present, although in some cases a host can be connected directly to a router. The collection of communication lines and routers (but not the hosts) form the subnet.

A short comment about the term "subnet" is in order here. Originally, its only meaning was the collection of routers and communication lines that moved packets from the source host to the destination host. However, some years later, it also acquired a second meaning in conjunction with network addressing. Unfortunately, no widely-used alternative exists for its initial meaning, so with some hesitation we will use it in both senses. From the context, it will always be clear which is meant.

In most WANs, the network contains numerous transmission lines, each one connecting a pair of routers. If two routers that do not share a transmission line wish to communicate, they must do this indirectly, via other routers. When a packet is sent from one router to another via one or more intermediate routers, the packet is received at each intermediate router in its entirety, stored there until the required output line is free, and then forwarded. A subnet organized according to this principle is called a store-and-forward or packet-switched subnet. Nearly all wide area networks (except those using satellites) have store-and-forward subnets. When the packets are small and all the same size, they are often called cells.

The principle of a packet-switched WAN is so important that it is worth devoting a few more words to it. Generally, when a process on some host has a message to be sent to a process on some other host, the sending host first cuts the message into packets, each one bearing its number in the sequence. These packets are then injected into the network one at a time in quick succession. The packets are transported individually over the network and deposited at the receiving host, where they are reassembled into the original message and delivered to the receiving process.

Routing decisions are made locally. When a packet arrives at router A, it is up to A to decide if this packet should be sent on the line to B or the line to C. How A makes that decision is called the routing algorithm. Many of them exist. Not all WANs are packet switched. A second possibility for a WAN is a satellite system. Each router has an antenna through which it can send and receive. All routers can hear the output from the satellite, and in some cases they can also hear the upward transmissions of their fellow routers to the satellite as well. Sometimes the routers are connected to a substantial point-to-point subnet, with only some of them having a satellite antenna. Satellite networks are inherently broadcast and are most useful when the broadcast property is important.

Wireless Networks

Digital wireless communication is not a new idea. As early as 1901, the Italian physicist Glielmo Marconi demonstrated a ship-to-shore wireless telegraph, using Morse Code (dots and dashes are binary, after all). Modern digital wireless systems have better performance, but the basic idea is the same.

To a first approximation, wireless networks can be divided into three main categories:

1. System interconnection.
2. Wireless LANs.
3. Wireless WANs.

System interconnection is all about interconnecting the components of a computer using short-range radio. Almost every computer has a monitor, keyboard, mouse, and printer connected to the main unit by cables. So many new users have a hard time plugging all the cables into the right little holes (even though they are usually color coded) that most computer vendors offer the option of sending a technician to the user's home to do it. Consequently, some companies got together to design a short-range wireless network called Bluetooth to connect these components without wires. Bluetooth also allows digital cameras, headsets, scanners, and other devices to connect to a computer by merely being brought within range. No cables, no driver installation, just put them down, turn them on, and they work. For many people, this ease of operation is a big plus.

In the simplest form, system interconnection networks use the master-slave paradigm. The system unit is normally the master, talking to the mouse, keyboard, etc., as slaves. The master tells the slaves what addresses to use, when they can broadcast, how long they can transmit, what frequencies they can use, and so on.

The next step up in wireless networking is the wireless LANs. These are systems in which every computer has a radio modem and antenna with which it can communicate with other systems. Often there is an antenna on the ceiling that the machines talk to. However, if the systems are close enough, they can communicate directly with one another in a peer-to-peer configuration. Wireless LANs are becoming increasingly common in small offices and homes, where installing Ethernet is considered too much trouble, as well as in older office buildings, company cafeterias, conference rooms, and other places. There is a standard for wireless LANs, called IEEE 802.11, which most systems implement and which is becoming very widespread. The third kind of wireless network is used in wide area systems. The radio network used for cellular telephones is an example of a low-bandwidth wireless system. This system has already gone through three generations. The first generation was analog and for voice only. The second generation was digital and for voice only. The third generation is digital and is for both voice and data. In a certain sense, cellular wireless networks are like wireless LANs, except that the distances involved are much greater and the bit rates much lower. Wireless LANs can operate at rates up to about 50 Mbps over distances of tens of meters. Cellular systems operate below 1 Mbps, but the distance between the base station and the computer or telephone is measured in kilometers rather than in meters.

In addition to these low-speed networks, high-bandwidth wide area wireless networks are also being developed. The initial focus is high-speed wireless Internet access from homes and businesses, bypassing the telephone system. This service is often called local multipoint distribution service. We will study it later in the book. A standard for it, called IEEE 802.16, has also been developed. Almost all wireless networks hook up to the wired network at some point to provide access to files, databases, and the Internet. There are many ways these connections can be realized, depending on the circumstances. For example, we depict an airplane with a number of people using modems and seat-back

telephones to call the office. Each call is independent of the other ones. Here each seat comes equipped with an Ethernet connector into which passengers can plug their computers. A single router on the aircraft maintains a radio link with some router on the ground, changing routers as it flies along. This configuration is just a traditional LAN, except that its connection to the outside world happens to be a radio link instead of a hardwired line.

Home Networks

Home networking is on the horizon. The fundamental idea is that in the future most homes will be set up for networking. Every device in the home will be capable of communicating with every other device, and all of them will be accessible over the Internet. This is one of those visionary concepts that nobody asked for (like TV remote controls or mobile phones), but once they arrived nobody can imagine how they lived without them.

Many devices are capable of being networked. Some of the more obvious categories (with examples) are as follows:

1. Computers (desktop PC, notebook PC, PDA, shared peripherals).
2. Entertainment (TV, DVD, VCR, camcorder, camera, stereo, MP3).
3. Telecommunications (telephone, mobile telephone, intercom, fax).
4. Appliances (microwave, refrigerator, clock, furnace, airco, lights).
5. Telemetry (utility meter, smoke/burglar alarm, thermostat, babycam).

Home computer networking is already here in a limited way. Many homes already have a device to connect multiple computers to a fast Internet connection. Networked entertainment is not quite here, but as more and more music and movies can be downloaded from the Internet, there will be a demand to connect stereos and televisions to it. Also, people will want to share their own videos with friends and family, so the connection will need to go both ways. Telecommunications gear is already connected to the outside world, but soon it will be digital and go over the Internet. The average home probably has a dozen clocks (e.g., in appliances), all of which have to be reset twice a

year when daylight saving time (summer time) comes and goes. If all the clocks were on the Internet, that resetting could be done automatically. Finally, remote monitoring of the home and its contents is a likely winner. Probably many parents would be willing to spend some money to monitor their sleeping babies on their PDAs when they are eating out, even with a rented teenager in the house. While one can imagine a separate network for each application area, integrating all of them into a single network is probably a better idea.

Home networking has some fundamentally different properties than other network types. First, the network and devices have to be easy to install. The author has installed numerous pieces of hardware and software on various computers over the years, with mixed results. A series of phone calls to the vendor's helpdesk typically resulted in answers like (1) Read the manual, (2) Reboot the computer, (3) Remove all hardware and software except ours and try again, (4) Download the newest driver from our Web site, and if all else fails, (5) Reformat the hard disk and then reinstall Windows from the CD-ROM. Telling the purchaser of an Internet refrigerator to download and install a new version of the refrigerator's operating system is not going to lead to happy customers. Computer users are accustomed to putting up with products that do not work; the car-, television-, and refrigerator-buying public is far less tolerant. They expect products to work for 100% from the word go.

Second, the network and devices have to be foolproof in operation. Air conditioners used to have one knob with four settings: OFF, LOW, MEDIUM, and HIGH. Now they have 30-page manuals. Once they are networked, expect the chapter on security alone to be 30 pages. This will be beyond the comprehension of virtually all the users.

Third, low price is essential for success. People will not pay a \$50 premium for an Internet thermostat because few people regard monitoring their home temperature from work that important. For \$5 extra, it might sell, though.

Fourth, the main application is likely to involve multimedia, so the network needs sufficient capacity. There is no market for Internet-connected televisions that show shaky

movies at 320 x 240 pixel resolution and 10 frames/sec. Fast Ethernet, the workhorse in most offices, is not good enough for multimedia. Consequently, home networks will need better performance than that of existing office networks and at lower prices before they become mass market items.

Fifth, it must be possible to start out with one or two devices and expand the reach of the network gradually. This means no format wars. Telling consumers to buy peripherals with IEEE 1394 (FireWire) interfaces and a few years later retracting that and saying USB 2.0 is the interface-of-the-month is going to make consumers skittish. The network interface will have to remain stable for many years; the wiring (if any) will have to remain stable for decades.

Sixth, security and reliability will be very important. Losing a few files to an e-mail virus is one thing; having a burglar disarm your security system from his PDA and then plunder your house is something quite different.

An interesting question is whether home networks will be wired or wireless. Most homes already have six networks installed: electricity, telephone, cable television, water, gas, and sewer. Adding a seventh one during construction is not difficult, but retrofitting existing houses is expensive. Cost favors wireless networking, but security favors wired networking. The problem with wireless is that the radio waves they use are quite good at going through fences. Not everyone is overjoyed at the thought of having the neighbors piggybacking on their Internet connection and reading their e-mail on its way to the printer. We will study how encryption can be used to provide security, but in the context of a home network, security has to be foolproof, even with inexperienced users. This is easier said than done, even with highly sophisticated users.

In short, home networking offers many opportunities and challenges. Most of them relate to the need to be easy to manage, dependable, and secure, especially in the hands of nontechnical users, while at the same time delivering high performance at low cost.

Internetworks

Many networks exist in the world, often with different hardware and software. People connected to one network often want to communicate with people attached to a different one. The fulfillment of this desire requires that different, and frequently incompatible networks, be connected, sometimes by means of machines called gateways to make the connection and provide the necessary translation, both in terms of hardware and software. A collection of interconnected networks is called an internetwork or internet. These terms will be used in a generic sense, in contrast to the worldwide Internet (which is one specific internet), which we will always capitalize.

A common form of internet is a collection of LANs connected by a WAN. In fact, if we were to replace the label "subnet" by "WAN," nothing else in the figure would have to change. The only real technical distinction between a subnet and a WAN in this case is whether hosts are present. If the system within the gray area contains only routers, it is a subnet; if it contains both routers and hosts, it is a WAN. The real differences relate to ownership and use.

Subnets, networks, and internetworks are often confused. Subnet makes the most sense in the context of a wide area network, where it refers to the collection of routers and communication lines owned by the network operator. As an analogy, the telephone system consists of telephone switching offices connected to one another by high-speed lines, and to houses and businesses by low-speed lines. These lines and equipment, owned and managed by the telephone company, form the subnet of the telephone system. The telephones themselves (the hosts in this analogy) are not part of the subnet. The combination of a subnet and its hosts forms a network. In the case of a LAN, the cable and the hosts form the network. There really is no subnet.

An internetwork is formed when distinct networks are interconnected. In our view, connecting a LAN and a WAN or connecting two LANs forms an internetwork, but there is little agreement in the industry over terminology in this area. One rule of thumb is that if different organizations paid to construct different parts of the network and each

maintains its part, we have an internetwork rather than a single network. Also, if the underlying technology is different in different parts (e.g., broadcast versus point-to-point), we probably have two networks.

Protocol Hierarchies

To reduce their design complexity, most networks are organized as a stack of layers or levels, each one built upon the one below it. The number of layers, the name of each layer, the contents of each layer, and the function of each layer differ from network to network. The purpose of each layer is to offer certain services to the higher layers, shielding those layers from the details of how the offered services are actually implemented. In a sense, each layer is a kind of virtual machine, offering certain services to the layer above it.

This concept is actually a familiar one and used throughout computer science, where it is variously known as information hiding, abstract data types, data encapsulation, and object-oriented programming. The fundamental idea is that a particular piece of software (or hardware) provides a service to its users but keeps the details of its internal state and algorithms hidden from them.

Layer n on one machine carries on a conversation with layer n on another machine. The rules and conventions used in this conversation are collectively known as the layer n protocol. Basically, a protocol is an agreement between the communicating parties on how communication is to proceed. As an analogy, when a woman is introduced to a man, she may choose to stick out her hand. He, in turn, may decide either to shake it or kiss it, depending, for example, on whether she is an American lawyer at a business meeting or a European princess at a formal ball. Violating the protocol will make communication more difficult, if not completely impossible.

The entities comprising the corresponding layers on different machines are called peers. The peers may be processes, hardware devices, or even human beings. In other words, it is the peers that communicate by using the protocol.

In reality, no data are directly transferred from layer n on one machine to layer n on another machine. Instead, each layer passes data and control information to the layer immediately below it, until the lowest layer is reached. Below layer 1 is the physical medium through which actual communication occurs. Virtual communication is shown by dotted lines and physical communication by solid lines.

Between each pair of adjacent layers is an interface. The interface defines which primitive operations and services the lower layer makes available to the upper one. When network designers decide how many layers to include in a network and what each one should do, one of the most important considerations is defining clean interfaces between the layers. Doing so, in turn, requires that each layer perform a specific collection of well-understood functions. In addition to minimizing the amount of information that must be passed between layers, clear-cut interfaces also make it simpler to replace the implementation of one layer with a completely different implementation (e.g., all the telephone lines are replaced by satellite channels) because all that is required of the new implementation is that it offer exactly the same set of services to its upstairs neighbor as the old implementation did. In fact, it is common that different hosts use different implementations.

A set of layers and protocols is called a network architecture. The specification of an architecture must contain enough information to allow an implementer to write the program or build the hardware for each layer so that it will correctly obey the appropriate protocol. Neither the details of the implementation nor the specification of the interfaces is part of the architecture because these are hidden away inside the machines and not visible from the outside. It is not even necessary that the interfaces on all machines in a network be the same, provided that each machine can correctly use all the protocols. A list of protocols used by a certain system, one protocol per layer, is called a protocol stack. The subjects of network architectures, protocol stacks, and the protocols themselves are the principal topics of this book.

An analogy may help explain the idea of multilayer communication. Imagine two philosophers (peer processes in layer 3), one of whom speaks Urdu and English and one

of whom speaks Chinese and French. Since they have no common language, they each engage a translator (peer processes at layer 2), each of whom in turn contacts a secretary (peer processes in layer 1). Philosopher 1 wishes to convey his affection for oryctolagus cuniculus to his peer. The translators have agreed on a neutral language known to both of them, Dutch, so the message is converted to "Ik vind konijnen leuk." The choice of language is the layer 2 protocol and is up to the layer 2 peer processes.

Design Issues for the Layers

Some of the key design issues that occur in computer networks are present in several layers. Below, we will briefly mention some of the more important ones. Every layer needs a mechanism for identifying senders and receivers. Since a network normally has many computers, some of which have multiple processes, a means is needed for a process on one machine to specify with whom it wants to talk. As a consequence of having multiple destinations, some form of addressing is needed in order to specify a specific destination.

Another set of design decisions concerns the rules for data transfer. In some systems, data only travel in one direction; in others, data can go both ways. The protocol must also determine how many logical channels the connection corresponds to and what their priorities are. Many networks provide at least two logical channels per connection, one for normal data and one for urgent data.

Error control is an important issue because physical communication circuits are not perfect. Many error-detecting and error-correcting codes are known, but both ends of the connection must agree on which one is being used. In addition, the receiver must have some way of telling the senders which messages have been correctly received and which have not.

Not all communication channels preserve the order of messages sent on them. To deal with a possible loss of sequencing, the protocol must make explicit provision for the receiver to allow the pieces to be reassembled properly. An obvious solution is to number

the pieces, but this solution still leaves open the question of what should be done with pieces that arrive out of order.

An issue that occurs at every level is how to keep a fast sender from swamping a slow receiver with data. Various solutions have been proposed and will be discussed later. Some of them involve some kind of feedback from the receiver to the sender, either directly or indirectly, about the receiver's current situation. Others limit the sender to an agreed-on transmission rate. This subject is called flow control.

Another problem that must be solved at several levels is the inability of all processes to accept arbitrarily long messages. This property leads to mechanisms for disassembling, transmitting, and then reassembling messages. A related issue is the problem of what to do when processes insist on transmitting data in units that are so small that sending each one separately is inefficient. Here the solution is to gather several small messages heading toward a common destination into a single large message and dismember the large message at the other side.

When it is inconvenient or expensive to set up a separate connection for each pair of communicating processes, the underlying layer may decide to use the same connection for multiple, unrelated conversations. As long as this multiplexing and demultiplexing is done transparently, it can be used by any layer. Multiplexing is needed in the physical layer, for example, where all the traffic for all connections has to be sent over at most a few physical circuits.

When there are multiple paths between source and destination, a route must be chosen. Sometimes this decision must be split over two or more layers. For example, to send data from London to Rome, a high-level decision might have to be made to transit France or Germany based on their respective privacy laws. Then a low-level decision might have to make to select one of the available circuits based on the current traffic load. This topic is called routing.

Connection-Oriented and Connectionless Services

Layers can offer two different types of service to the layers above them: connection-oriented and connectionless. In this section we will look at these two types and examine the differences between them.

Connection-oriented service is modeled after the telephone system. To talk to someone, you pick up the phone, dial the number, talk, and then hang up. Similarly, to use a connection-oriented network service, the service user first establishes a connection, uses the connection, and then releases the connection. The essential aspect of a connection is that it acts like a tube: the sender pushes objects (bits) in at one end, and the receiver takes them out at the other end. In most cases the order is preserved so that the bits arrive in the order they were sent.

In some cases when a connection is established, the sender, receiver, and subnet conduct a negotiation about parameters to be used, such as maximum message size, quality of service required, and other issues. Typically, one side makes a proposal and the other side can accept it, reject it, or make a counterproposal.

In contrast, connectionless service is modeled after the postal system. Each message (letter) carries the full destination address, and each one is routed through the system independent of all the others. Normally, when two messages are sent to the same destination, the first one sent will be the first one to arrive. However, it is possible that the first one sent can be delayed so that the second one arrives first.

Each service can be characterized by a quality of service. Some services are reliable in the sense that they never lose data. Usually, a reliable service is implemented by having the receiver acknowledge the receipt of each message so the sender is sure that it arrived. The acknowledgement process introduces overhead and delays, which are often worth it but are sometimes undesirable.

A typical situation in which a reliable connection-oriented service is appropriate is file transfer. The owner of the file wants to be sure that all the bits arrive correctly and in the

same order they were sent. Very few file transfer customers would prefer a service that occasionally scrambles or loses a few bits, even if it is much faster.

Reliable connection-oriented service has two minor variations: message sequences and byte streams. In the former variant, the message boundaries are preserved. When two 1024-byte messages are sent, they arrive as two distinct 1024-byte messages, never as one 2048-byte message. In the latter, the connection is simply a stream of bytes, with no message boundaries. When 2048 bytes arrive at the receiver, there is no way to tell if they were sent as one 2048-byte message, two 1024-byte messages, or 2048 1-byte messages. If the pages of a book are sent over a network to a phototypesetter as separate messages, it might be important to preserve the message boundaries. On the other hand, when a user logs into a remote server, a byte stream from the user's computer to the server is all that is needed. Message boundaries are not relevant.

As mentioned above, for some applications, the transit delays introduced by acknowledgements are unacceptable. One such application is digitized voice traffic. It is preferable for telephone users to hear a bit of noise on the line from time to time than to experience a delay waiting for acknowledgements. Similarly, when transmitting a video conference, having a few pixels wrong is no problem, but having the image jerk along as the flow stops to correct errors is irritating.

Not all applications require connections. For example, as electronic mail becomes more common, electronic junk is becoming more common too. The electronic junk-mail sender probably does not want to go to the trouble of setting up and later tearing down a connection just to send one item. Nor is 100 percent reliable delivery essential, especially if it costs more. All that is needed is a way to send a single message that has a high probability of arrival, but no guarantee. Unreliable (meaning not acknowledged) connectionless service is often called datagram service, in analogy with telegram service, which also does not return an acknowledgement to the sender.

In other situations, the convenience of not having to establish a connection to send one short message is desired, but reliability is essential. The acknowledged datagram service

can be provided for these applications. It is like sending a registered letter and requesting a return receipt. When the receipt comes back, the sender is absolutely sure that the letter was delivered to the intended party and not lost along the way.

Still another service is the request-reply service. In this service the sender transmits a single datagram containing a request; the reply contains the answer. For example, a query to the local library asking where Uighur is spoken falls into this category. Request-reply is commonly used to implement communication in the client-server model: the client issues a request and the server responds to it.

Service Primitives

A service is formally specified by a set of primitives (operations) available to a user process to access the service. These primitives tell the service to perform some action or report on an action taken by a peer entity. If the protocol stack is located in the operating system, as it often is, the primitives are normally system calls. These calls cause a trap to kernel mode, which then turns control of the machine over to the operating system to send the necessary packets.

The set of primitives available depends on the nature of the service being provided. The primitives for connection-oriented service are different from those of connectionless service. As a minimal example of the service primitives that might be provided to implement a reliable byte stream in a client-server environment.

These primitives might be used as follows. First, the server executes LISTEN to indicate that it is prepared to accept incoming connections. A common way to implement LISTEN is to make it a blocking system call. After executing the primitive, the server process is blocked until a request for connection appears.

Next, the client process executes CONNECT to establish a connection with the server. The CONNECT call needs to specify who to connect to, so it might have a parameter giving the server's address. The operating system then typically sends a packet to the peer asking it to connect, as shown by (1). The client process is suspended until there is a

response. When the packet arrives at the server, it is processed by the operating system there. When the system sees that the packet is requesting a connection, it checks to see if there is a listener. If so, it does two things: unblocks the listener and sends back an acknowledgement (2). The arrival of this acknowledgement then releases the client. At this point the client and server are both running and they have a connection established. It is important to note that the acknowledgement (2) is generated by the protocol code itself, not in response to a user-level primitive. If a connection request arrives and there is no listener, the result is undefined. In some systems the packet may be queued for a short time in anticipation of a LISTEN.

The obvious analogy between this protocol and real life is a customer (client) calling a company's customer service manager. The service manager starts out by being near the telephone in case it rings. Then the client places the call. When the manager picks up the phone, the connection is established.

The next step is for the server to execute RECEIVE to prepare to accept the first request. Normally, the server does this immediately upon being released from the LISTEN, before the acknowledgement can get back to the client. The RECEIVE call blocks the server.

Then the client executes SEND to transmit its request (3) followed by the execution of RECEIVE to get the reply.

The arrival of the request packet at the server machine unblocks the server process so it can process the request. After it has done the work, it uses SEND to return the answer to the client (4). The arrival of this packet unblocks the client, which can now inspect the answer. If the client has additional requests, it can make them now. If it is done, it can use DISCONNECT to terminate the connection. Usually, an initial DISCONNECT is a blocking call, suspending the client and sending a packet to the server saying that the connection is no longer needed (5). When the server gets the packet, it also issues a DISCONNECT of its own, acknowledging the client and releasing the connection. When the server's packet (6) gets back to the client machine, the client process is released and

the connection is broken. In a nutshell, this is how connection-oriented communication works.

Of course, life is not so simple. Many things can go wrong here. The timing can be wrong (e.g., the CONNECT is done before the LISTEN), packets can get lost, and much more. We will look at these issues in great detail later, but for the moment, briefly summarizes how client-server communication might work over a connection-oriented network.

Given that six packets are required to complete this protocol, one might wonder why a connectionless protocol is not used instead. The answer is that in a perfect world it could be, in which case only two packets would be needed: one for the request and one for the reply. However, in the face of large messages in direction (e.g., a megabyte file), transmission errors, and lost packets, the situation changes. If the reply consisted of hundreds of packets, some of which could be lost during transmission, how would the client know if some pieces were missing? How would the client know whether the last packet actually received was really the last packet sent? Suppose that the client wanted a second file. How could it tell packet 1 from the second file from a lost packet 1 from the first file that suddenly found its way to the client? In short, in the real world, a simple request-reply protocol over an unreliable network is often inadequate. We will study a variety of protocols in detail that overcome these and other problems. For the moment, suffice it to say that having a reliable, ordered byte stream between processes is sometimes very convenient.

The Relationship of Services to Protocols

Services and protocols are distinct concepts, although they are frequently confused. This distinction is so important, however, that we emphasize it again here. A service is a set of primitives (operations) that a layer provides to the layer above it. The service defines what operations the layer is prepared to perform on behalf of its users, but it says nothing at all about how these operations are implemented. A service relates to an interface between two layers, with the lower layer being the service provider and the upper layer being the service user.

A protocol, in contrast, is a set of rules governing the format and meaning of the packets, or messages that are exchanged by the peer entities within a layer. Entities use protocols to implement their service definitions. They are free to change their protocols at will, provided they do not change the service visible to their users. In this way, the service and the protocol are completely decoupled.

In other words, services relate to the interfaces between layers. In contrast, protocols relate to the packets sent between peer entities on different machines. It is important not to confuse the two concepts.

An analogy with programming languages is worth making. A service is like an abstract data type or an object in an object-oriented language. It defines operations that can be performed on an object but does not specify how these operations are implemented. A protocol relates to the implementation of the service and as such is not visible to the user of the service.

Many older protocols did not distinguish the service from the protocol. In effect, a typical layer might have had a service primitive `SEND PACKET` with the user providing a pointer to a fully assembled packet. This arrangement meant that all changes to the protocol were immediately visible to the users. Most network designers now regard such a design as a serious blunder.

Reference Models

Now that we have discussed layered networks in the abstract, it is time to look at some examples. In the next two sections we will discuss two important network architectures, the OSI reference model and the TCP/IP reference model. Although the protocols associated with the OSI model are rarely used any more, the model itself is actually quite general and still valid, and the features discussed at each layer are still very important. The TCP/IP model has the opposite properties: the model itself is not of much use but the protocols are widely used. For this reason we will look at both of them in detail. Also, sometimes you can learn more from failures than from successes.

The OSI Reference Model

This model is based on a proposal developed by the International Standards Organization (ISO) as a first step toward international standardization of the protocols used in the various layers (Day and Zimmermann, 1983). It was revised in 1995 (Day, 1995). The model is called the ISO OSI (Open Systems Interconnection) Reference Model because it deals with connecting open systems—that is, systems that are open for communication with other systems. We will just call it the OSI model for short.

The OSI model has seven layers. The principles that were applied to arrive at the seven layers can be briefly summarized as follows:

1. A layer should be created where a different abstraction is needed.
2. Each layer should perform a well-defined function.
3. The function of each layer should be chosen with an eye toward defining internationally standardized protocols.
4. The layer boundaries should be chosen to minimize the information flow across the interfaces.
5. The number of layers should be large enough that distinct functions need not be thrown together in the same layer out of necessity and small enough that the architecture does not become unwieldy.

Below we will discuss each layer of the model in turn, starting at the bottom layer. Note that the OSI model itself is not a network architecture because it does not specify the exact services and protocols to be used in each layer. It just tells what each layer should do. However, ISO has also produced standards for all the layers, although these are not part of the reference model itself. Each one has been published as a separate international standard.

The Physical Layer

The physical layer is concerned with transmitting raw bits over a communication channel. The design issues have to do with making sure that when one side sends a 1 bit, it is received by the other side as a 1 bit, not as a 0 bit. Typical questions here are how many volts should be used to represent a 1 and how many for a 0, how many nanoseconds a bit lasts, whether transmission may proceed simultaneously in both directions, how the initial connection is established and how it is torn down when both sides are finished, and how many pins the network connector has and what each pin is used for. The design issues here largely deal with mechanical, electrical, and timing interfaces, and the physical transmission medium, which lies below the physical layer.

The Data Link Layer

The main task of the data link layer is to transform a raw transmission facility into a line that appears free of undetected transmission errors to the network layer. It accomplishes this task by having the sender break up the input data into data frames (typically a few hundred or a few thousand bytes) and transmit the frames sequentially. If the service is reliable, the receiver confirms correct receipt of each frame by sending back an acknowledgement frame.

Another issue that arises in the data link layer (and most of the higher layers as well) is how to keep a fast transmitter from drowning a slow receiver in data. Some traffic regulation mechanism is often needed to let the transmitter know how much buffer space the receiver has at the moment. Frequently, this flow regulation and the error handling are integrated.

Broadcast networks have an additional issue in the data link layer: how to control access to the shared channel. A special sublayer of the data link layer, the medium access control sublayer, deals with this problem.

The Network Layer

The network layer controls the operation of the subnet. A key design issue is determining how packets are routed from source to destination. Routes can be based on static tables that are "wired into" the network and rarely changed. They can also be determined at the start of each conversation, for example, a terminal session (e.g., a login to a remote machine). Finally, they can be highly dynamic, being determined anew for each packet, to reflect the current network load.

If too many packets are present in the subnet at the same time, they will get in one another's way, forming bottlenecks. The control of such congestion also belongs to the network layer. More generally, the quality of service provided (delay, transit time, jitter, etc.) is also a network layer issue.

When a packet has to travel from one network to another to get to its destination, many problems can arise. The addressing used by the second network may be different from the first one. The second one may not accept the packet at all because it is too large. The protocols may differ, and so on. It is up to the network layer to overcome all these problems to allow heterogeneous networks to be interconnected.

In broadcast networks, the routing problem is simple, so the network layer is often thin or even nonexistent.

The Transport Layer

The basic function of the transport layer is to accept data from above, split it up into smaller units if need be, pass these to the network layer, and ensure that the pieces all arrive correctly at the other end. Furthermore, all this must be done efficiently and in a way that isolates the upper layers from the inevitable changes in the hardware technology.

The transport layer also determines what type of service to provide to the session layer, and, ultimately, to the users of the network. The most popular type of transport

connection is an error-free point-to-point channel that delivers messages or bytes in the order in which they were sent. However, other possible kinds of transport service are the transporting of isolated messages, with no guarantee about the order of delivery, and the broadcasting of messages to multiple destinations. The type of service is determined when the connection is established. (As an aside, an error-free channel is impossible to achieve; what people really mean by this term is that the error rate is low enough to ignore in practice.)

The transport layer is a true end-to-end layer, all the way from the source to the destination. In other words, a program on the source machine carries on a conversation with a similar program on the destination machine, using the message headers and control messages. In the lower layers, the protocols are between each machine and its immediate neighbors, and not between the ultimate source and destination machines, which may be separated by many routers. The difference between layers 1 through 3, which are chained, and layers 4 through 7, which are end-to-end.

The Session Layer

The session layer allows users on different machines to establish sessions between them. Sessions offer various services, including dialog control (keeping track of whose turn it is to transmit), token management (preventing two parties from attempting the same critical operation at the same time), and synchronization (checkpointing long transmissions to allow them to continue from where they were after a crash).

The Presentation Layer

Unlike lower layers, which are mostly concerned with moving bits around, the presentation layer is concerned with the syntax and semantics of the information transmitted. In order to make it possible for computers with different data representations to communicate, the data structures to be exchanged can be defined in an abstract way, along with a standard encoding to be used "on the wire." The presentation layer manages

these abstract data structures and allows higher-level data structures (e.g., banking records), to be defined and exchanged.

The Application Layer

The application layer contains a variety of protocols that are commonly needed by users. One widely-used application protocol is HTTP (HyperText Transfer Protocol), which is the basis for the World Wide Web. When a browser wants a Web page, it sends the name of the page it wants to the server using HTTP. The server then sends the page back. Other application protocols are used for file transfer, electronic mail, and network news.

The TCP/IP Reference Model

Let us now turn from the OSI reference model to the reference model used in the grandparent of all wide area computer networks, the ARPANET, and its successor, the worldwide Internet. Although we will give a brief history of the ARPANET later, it is useful to mention a few key aspects of it now. The ARPANET was a research network sponsored by the DoD (U.S. Department of Defense). It eventually connected hundreds of universities and government installations, using leased telephone lines. When satellite and radio networks were added later, the existing protocols had trouble interworking with them, so a new reference architecture was needed. Thus, the ability to connect multiple networks in a seamless way was one of the major design goals from the very beginning. This architecture later became known as the TCP/IP Reference Model, after its two primary protocols. It was first defined in (Cerf and Kahn, 1974). A later perspective is given in (Leiner et al., 1985). The design philosophy behind the model is discussed in (Clark, 1988).

Given the DoD's worry that some of its precious hosts, routers, and internetwork gateways might get blown to pieces at a moment's notice, another major goal was that the network be able to survive loss of subnet hardware, with existing conversations not being broken off. In other words, DoD wanted connections to remain intact as long as the source and destination machines were functioning, even if some of the machines or

transmission lines in between were suddenly put out of operation. Furthermore, a flexible architecture was needed since applications with divergent requirements were envisioned, ranging from transferring files to real-time speech transmission.

The Internet Layer

All these requirements led to the choice of a packet-switching network based on a connectionless internetwork layer. This layer, called the internet layer, is the linchpin that holds the whole architecture together. Its job is to permit hosts to inject packets into any network and have them travel independently to the destination (potentially on a different network). They may even arrive in a different order than they were sent, in which case it is the job of higher layers to rearrange them, if in-order delivery is desired. Note that "internet" is used here in a generic sense, even though this layer is present in the Internet.

The analogy here is with the (snail) mail system. A person can drop a sequence of international letters into a mail box in one country, and with a little luck, most of them will be delivered to the correct address in the destination country. Probably the letters will travel through one or more international mail gateways along the way, but this is transparent to the users. Furthermore, that each country (i.e., each network) has its own stamps, preferred envelope sizes, and delivery rules is hidden from the users.

The internet layer defines an official packet format and protocol called IP (Internet Protocol). The job of the internet layer is to deliver IP packets where they are supposed to go. Packet routing is clearly the major issue here, as is avoiding congestion. For these reasons, it is reasonable to say that the TCP/IP internet layer is similar in functionality to the OSI network layer.

The Transport Layer

The layer above the internet layer in the TCP/IP model is now usually called the transport layer. It is designed to allow peer entities on the source and destination hosts to carry on a conversation, just as in the OSI transport layer. Two end-to-end transport protocols have been defined here. The first one, TCP (Transmission Control Protocol), is a reliable connection-oriented protocol that allows a byte stream originating on one machine to be delivered without error on any other machine in the internet. It fragments the incoming byte stream into discrete messages and passes each one on to the internet layer. At the destination, the receiving TCP process reassembles the received messages into the output stream. TCP also handles flow control to make sure a fast sender cannot swamp a slow receiver with more messages than it can handle.

The second protocol in this layer, UDP (User Datagram Protocol), is an unreliable, connectionless protocol for applications that do not want TCP's sequencing or flow control and wish to provide their own. It is also widely used for one-shot, client-server-type request-reply queries and applications in which prompt delivery is more important than accurate delivery, such as transmitting speech or video. Since the model was developed, IP has been implemented on many other networks.

The Application Layer

The TCP/IP model does not have session or presentation layers. No need for them was perceived, so they were not included. Experience with the OSI model has proven this view correct: they are of little use to most applications.

On top of the transport layer is the application layer. It contains all the higher-level protocols. The early ones included virtual terminal (TELNET), file transfer (FTP), and electronic mail (SMTP). The virtual terminal protocol allows a user on one machine to log onto a distant machine and work there. The file transfer protocol provides a way to move data efficiently from one machine to another. Electronic mail was originally just a kind of file transfer, but later a specialized protocol (SMTP) was developed for it. Many

other protocols have been added to these over the years: the Domain Name System (DNS) for mapping host names onto their network addresses, NNTP, the protocol for moving USENET news articles around, and HTTP, the protocol for fetching pages on the World Wide Web, and many others.

The Host-to-Network Layer

Below the internet layer is a great void. The TCP/IP reference model does not really say much about what happens here, except to point out that the host has to connect to the network using some protocol so it can send IP packets to it. This protocol is not defined and varies from host to host and network to network. Books and papers about the TCP/IP model rarely discuss it.

A Comparison of the OSI and TCP/IP Reference Models

The OSI and TCP/IP reference models have much in common. Both are based on the concept of a stack of independent protocols. Also, the functionality of the layers is roughly similar. For example, in both models the layers up through and including the transport layer are there to provide an end-to-end, network-independent transport service to processes wishing to communicate. These layers form the transport provider. Again in both models, the layers above transport are application-oriented users of the transport service.

Despite these fundamental similarities, the two models also have many differences. In this section we will focus on the key differences between the two reference models. It is important to note that we are comparing the reference models here, not the corresponding protocol stacks. The protocols themselves will be discussed later. For an entire book comparing and contrasting TCP/IP and OSI, see (Piscitello and Chapin, 1993).

Three concepts are central to the OSI model:

1. Services.
2. Interfaces.

3. Protocols.

Probably the biggest contribution of the OSI model is to make the distinction between these three concepts explicit. Each layer performs some services for the layer above it. The service definition tells what the layer does, not how entities above it access it or how the layer works. It defines the layer's semantics.

A layer's interface tells the processes above it how to access it. It specifies what the parameters are and what results to expect. It, too, says nothing about how the layer works inside.

Finally, the peer protocols used in a layer are the layer's own business. It can use any protocols it wants to, as long as it gets the job done (i.e., provides the offered services). It can also change them at will without affecting software in higher layers.

These ideas fit very nicely with modern ideas about object-oriented programming. An object, like a layer, has a set of methods (operations) that processes outside the object can invoke. The semantics of these methods define the set of services that the object offers. The methods' parameters and results form the object's interface. The code internal to the object is its protocol and is not visible or of any concern outside the object.

The TCP/IP model did not originally clearly distinguish between service, interface, and protocol, although people have tried to retrofit it after the fact to make it more OSI-like. For example, the only real services offered by the internet layer are SEND IP PACKET and RECEIVE IP PACKET.

As a consequence, the protocols in the OSI model are better hidden than in the TCP/IP model and can be replaced relatively easily as the technology changes. Being able to make such changes is one of the main purposes of having layered protocols in the first place.

The OSI reference model was devised before the corresponding protocols were invented. This ordering means that the model was not biased toward one particular set of protocols,

a fact that made it quite general. The downside of this ordering is that the designers did not have much experience with the subject and did not have a good idea of which functionality to put in which layer.

For example, the data link layer originally dealt only with point-to-point networks. When broadcast networks came around, a new sublayer had to be hacked into the model. When people started to build real networks using the OSI model and existing protocols, it was discovered that these networks did not match the required service specifications (wonder of wonders), so convergence sublayers had to be grafted onto the model to provide a place for papering over the differences. Finally, the committee originally expected that each country would have one network, run by the government and using the OSI protocols, so no thought was given to internetworking. To make a long story short, things did not turn out that way.

With TCP/IP the reverse was true: the protocols came first, and the model was really just a description of the existing protocols. There was no problem with the protocols fitting the model. They fit perfectly. The only trouble was that the model did not fit any other protocol stacks. Consequently, it was not especially useful for describing other, non-TCP/IP networks.

Turning from philosophical matters to more specific ones, an obvious difference between the two models is the number of layers: the OSI model has seven layers and the TCP/IP has four layers. Both have (inter)network, transport, and application layers, but the other layers are different.

Another difference is in the area of connectionless versus connection-oriented communication. The OSI model supports both connectionless and connection-oriented communication in the network layer, but only connection-oriented communication in the transport layer, where it counts (because the transport service is visible to the users). The TCP/IP model has only one mode in the network layer (connectionless) but supports both modes in the transport layer, giving the users a choice. This choice is especially important for simple request-response protocols.

Ethernet

Both the Internet and ATM were designed for wide area networking. However, many companies, universities, and other organizations have large numbers of computers that must be connected. This need gave rise to the local area network. In this section we will say a little bit about the most popular LAN, Ethernet.

The story starts out in pristine Hawaii in the early 1970s. In this case, "pristine" can be interpreted as "not having a working telephone system." While not being interrupted by the phone all day long makes life more pleasant for vacationers, it did not make life more pleasant for researcher Norman Abramson and his colleagues at the University of Hawaii who were trying to connect users on remote islands to the main computer in Honolulu. Stringing their own cables under the Pacific Ocean was not in the cards, so they looked for a different solution.

The one they found was short-range radios. Each user terminal was equipped with a small radio having two frequencies: upstream (to the central computer) and downstream (from the central computer). When the user wanted to contact the computer, it just transmitted a packet containing the data in the upstream channel. If no one else was transmitting at that instant, the packet probably got through and was acknowledged on the downstream channel. If there was contention for the upstream channel, the terminal noticed the lack of acknowledgement and tried again. Since there was only one sender on the downstream channel (the central computer), there were never collisions there. This system, called ALOHANET, worked fairly well under conditions of low traffic but bogged down badly when the upstream traffic was heavy.

About the same time, a student named Bob Metcalfe got his bachelor's degree at M.I.T. and then moved up the river to get his Ph.D. at Harvard. During his studies, he was exposed to Abramson's work. He became so interested in it that after graduating from Harvard, he decided to spend the summer in Hawaii working with Abramson before starting work at Xerox PARC (Palo Alto Research Center). When he got to PARC, he saw that the researchers there had designed and built what would later be called personal

computers. But the machines were isolated. Using his knowledge of Abramson's work, he, together with his colleague David Boggs, designed and implemented the first local area network (Metcalfe and Boggs, 1976).

They called the system Ethernet after the luminiferous ether, through which electromagnetic radiation was once thought to propagate. (When the 19th century British physicist James Clerk Maxwell discovered that electromagnetic radiation could be described by a wave equation, scientists assumed that space must be filled with some ethereal medium in which the radiation was propagating. Only after the famous Michelson-Morley experiment in 1887 did physicists discover that electromagnetic radiation could propagate in a vacuum.)

The transmission medium here was not a vacuum, but a thick coaxial cable (the ether) up to 2.5 km long (with repeaters every 500 meters). Up to 256 machines could be attached to the system via transceivers screwed onto the cable. A cable with multiple machines attached to it in parallel is called a multidrop cable. The system ran at 2.94 Mbps. Ethernet had a major improvement over ALOHANET: before transmitting, a computer first listened to the cable to see if someone else was already transmitting. If so, the computer held back until the current transmission finished. Doing so avoided interfering with existing transmissions, giving a much higher efficiency. ALOHANET did not work like this because it was impossible for a terminal on one island to sense the transmission of a terminal on a distant island. With a single cable, this problem does not exist.

Despite the computer listening before transmitting, a problem still arises: what happens if two or more computers all wait until the current transmission completes and then all start at once? The solution is to have each computer listen during its own transmission and if it detects interference, jam the ether to alert all senders. Then back off and wait a random time before retrying. If a second collision happens, the random waiting time is doubled, and so on, to spread out the competing transmissions and give one of them a chance to go first.

The Xerox Ethernet was so successful that DEC, Intel, and Xerox drew up a standard in 1978 for a 10-Mbps Ethernet, called the DIX standard. With two minor changes, the DIX standard became the IEEE 802.3 standard in 1983.

Unfortunately for Xerox, it already had a history of making seminal inventions (such as the personal computer) and then failing to commercialize on them, a story told in *Fumbling the Future* (Smith and Alexander, 1988). When Xerox showed little interest in doing anything with Ethernet other than helping standardize it, Metcalfe formed his own company, 3Com, to sell Ethernet adapters for PCs. It has sold over 100 million of them.

Ethernet continued to develop and is still developing. New versions at 100 Mbps, 1000 Mbps, and still higher have come out. Also the cabling has improved, and switching and other features have been added. In passing, it is worth mentioning that Ethernet (IEEE 802.3) is not the only LAN standard. The committee also standardized a token bus (802.4) and a token ring (802.5). The need for three more-or-less incompatible standards has little to do with technology and everything to do with politics. At the time of standardization, General Motors was pushing a LAN in which the topology was the same as Ethernet (a linear cable) but computers took turns in transmitting by passing a short packet called a token from computer to computer. A computer could only send if it possessed the token, thus avoiding collisions. General Motors announced that this scheme was essential for manufacturing cars and was not prepared to budge from this position. This announcement notwithstanding, 802.4 has basically vanished from sight.

Similarly, IBM had its own favorite: its proprietary token ring. The token was passed around the ring and whichever computer held the token was allowed to transmit before putting the token back on the ring. Unlike 802.4, this scheme, standardized as 802.5, is still in use at some IBM sites, but virtually nowhere outside of IBM sites. However, work is progressing on a gigabit version (802.5v), but it seems unlikely that it will ever catch up with Ethernet. In short, there was a war between Ethernet, token bus, and token ring, and Ethernet won, mostly because it was there first and the challengers were not as good.

Wireless LANs: 802.11

Almost as soon as notebook computers appeared, many people had a dream of walking into an office and magically having their notebook computer be connected to the Internet. Consequently, various groups began working on ways to accomplish this goal. The most practical approach is to equip both the office and the notebook computers with short-range radio transmitters and receivers to allow them to communicate. This work rapidly led to wireless LANs being marketed by a variety of companies.

The trouble was that no two of them were compatible. This proliferation of standards meant that a computer equipped with a brand X radio would not work in a room equipped with a brand Y base station. Finally, the industry decided that a wireless LAN standard might be a good idea, so the IEEE committee that standardized the wired LANs was given the task of drawing up a wireless LAN standard. The standard it came up with was named 802.11. A common slang name for it is WiFi. It is an important standard and deserves respect, so we will call it by its proper name, 802.11.

The proposed standard had to work in two modes:

1. In the presence of a base station.
2. In the absence of a base station.

In the former case, all communication was to go through the base station, called an access point in 802.11 terminology. In the latter case, the computers would just send to one another directly. This mode is now sometimes called ad hoc networking. A typical example is two or more people sitting down together in a room not equipped with a wireless LAN and having their computers just communicate directly.

In particular, some of the many challenges that had to be met were: finding a suitable frequency band that was available, preferably worldwide; dealing with the fact that radio signals have a finite range; ensuring that users' privacy was maintained; taking limited battery life into account; worrying about human safety (do radio waves cause cancer?);

understanding the implications of computer mobility; and finally, building a system with enough bandwidth to be economically viable.

At the time the standardization process started (mid-1990s), Ethernet had already come to dominate local area networking, so the committee decided to make 802.11 compatible with Ethernet above the data link layer. In particular, it should be possible to send an IP packet over the wireless LAN the same way a wired computer sent an IP packet over Ethernet. Nevertheless, in the physical and data link layers, several inherent differences with Ethernet exist and had to be dealt with by the standard.

First, a computer on Ethernet always listens to the ether before transmitting. Only if the ether is idle does the computer begin transmitting. With wireless LANs, that idea does not work so well. Suppose that computer A is transmitting to computer B, but the radio range of A's transmitter is too short to reach computer C. If C wants to transmit to B it can listen to the ether before starting, but the fact that it does not hear anything does not mean that its transmission will succeed. The 802.11 standard had to solve this problem.

The second problem that had to be solved is that a radio signal can be reflected off solid objects, so it may be received multiple times (along multiple paths). This interference results in what is called multipath fading.

The third problem is that a great deal of software is not aware of mobility. For example, many word processors have a list of printers that users can choose from to print a file. When the computer on which the word processor runs is taken into a new environment, the built-in list of printers becomes invalid.

The fourth problem is that if a notebook computer is moved away from the ceiling-mounted base station it is using and into the range of a different base station, some way of handing it off is needed. Although this problem occurs with cellular telephones, it does not occur with Ethernet and needed to be solved. In particular, the network envisioned consists of multiple cells, each with its own base station, but with the base stations connected by Ethernet. From the outside, the entire system should look like a single

Ethernet. The connection between the 802.11 system and the outside world is called a portal.

After some work, the committee came up with a standard in 1997 that addressed these and other concerns. The wireless LAN it described ran at either 1 Mbps or 2 Mbps. Almost immediately, people complained that it was too slow, so work began on faster standards. A split developed within the committee, resulting in two new standards in 1999. The 802.11a standard uses a wider frequency band and runs at speeds up to 54 Mbps. The 802.11b standard uses the same frequency band as 802.11, but uses a different modulation technique to achieve 11 Mbps. Some people see this as psychologically important since 11 Mbps is faster than the original wired Ethernet.

4.2 PEER – TO –PEER COMMUNICATION

The Evolution of Peer-to-Peer

Peer-to-peer is an almost magical term that's often used, rarely explained, and frequently misunderstood. In the popular media, peer-to-peer is often described as a copyright-violating technology that underlies song-swapping and file-sharing systems such as Napster and Gnutella. In the world of high-tech business, peer-to-peer networking is a revolution that promises to harness the combined computing power of ordinary personal computers and revolutionize the way we communicate. And to Internet pioneers, peer-to-peer is as much a philosophy as it is a model of development, one that contains the keys needed to defeat censorship and create global communities. All of these descriptions contain part of the answer, but none will help you build your own peer-to-peer systems, or explain why you should.

Brief History of Programming

The easiest way to understand peer-to-peer applications is by comparing them to other models of programming architecture. To understand peer-to-peer programming, you need to realize that it's part revolution, part evolution. On the one hand, peer-to-peer

programming is the latest in a long line of schisms that have shaken up the programming world. Like them, it promises to change the face of software development forever. On the other hand, peer-to-peer programming borrows heavily from the past. It's likely that peer-to-peer concepts may end up enhancing existing systems, rather than replacing them.

The Birth of Client-Server

In a traditional business environment, software is centralized around a server. In the not-so-distant past, this role was played by a mainframe. The mainframe performed all the work, processing information, accessing data stores, and so on. The clients were marginalized and computationally unimportant: "dumb terminals." They were nothing more than an interface to the mainframe. As Windows development gained in popularity, servers replaced the mainframe, and dumb terminals were upgraded to low-cost Windows stations that assumed a more important role. This was the start of the era of client-server development.

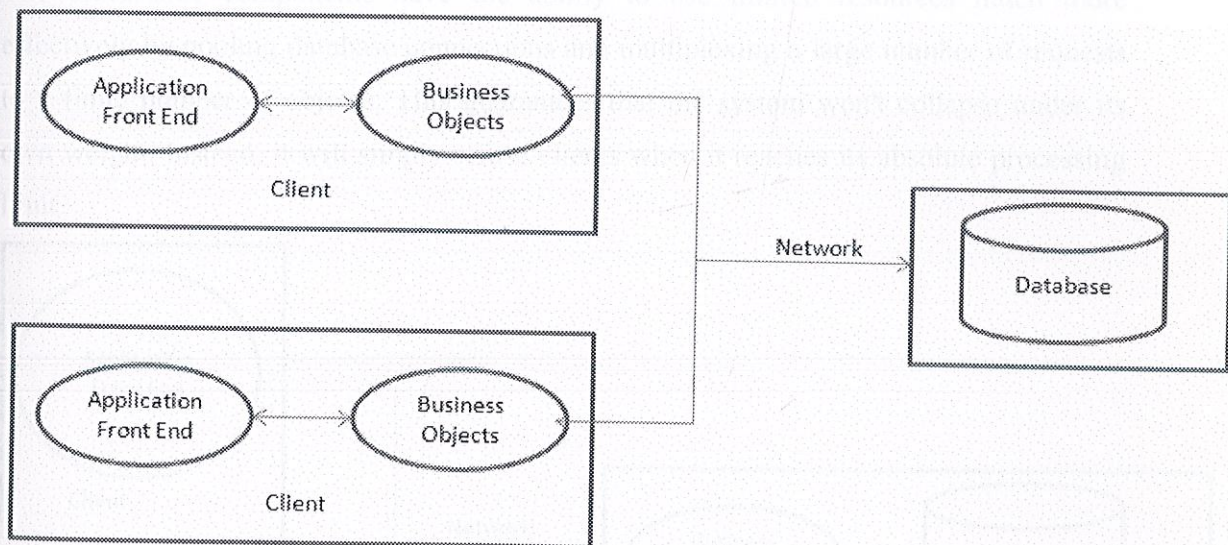


Fig 2.2.1: Client-Server Model

Distributed Computing

In a distributed system, the client doesn't need to directly process the business and data-access logic or connect directly to the database. Instead, the client interacts with a set of components running on a server computer, which in turn communicates with a data store

or another set of components. Thus, unlike a client-server system, a significant part of the business code executes on the server computer.

By dividing an application into multiple layers, it becomes possible for several computers to contribute in the processing of a single request. This distribution of logic typically slows down individual client requests (because of the additional overhead required for network communication), but it improves the overall throughput for the entire system. Thus, distributed systems are much more scalable than client-server systems and can handle larger client loads.

Here are some of the key innovations associated with distributed computing: If more computing power is needed, you can simply move components to additional servers instead of providing a costly server upgrade.

If good stateless programming practices are followed, you can replace individual servers with a clustered group of servers, thereby improving scalability.

The server-side components have the ability to use limited resources much more effectively by pooling database connections and multiplexing a large number of requests to a finite number of objects. This guarantees that the system won't collapse under its own weight. Instead, it will simply refuse clients when it reaches its absolute processing limit.

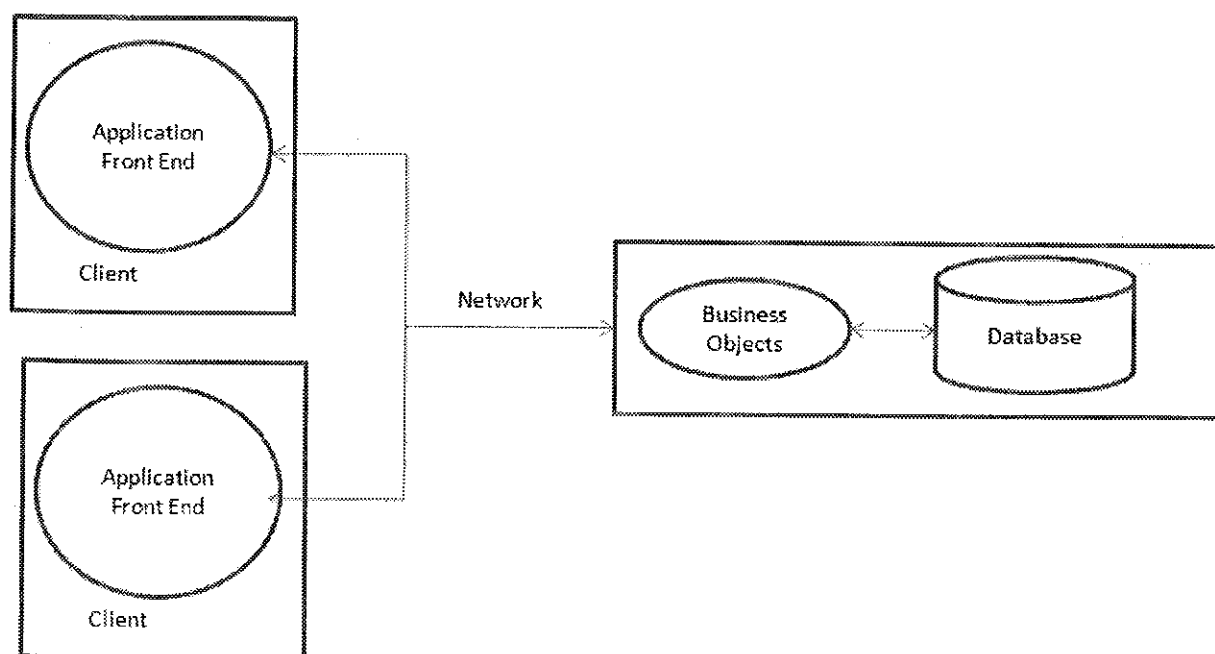


Fig 2.2.2: Distributed Computing

Peer-to-Peer Appears

The dependency on a central set of servers isn't necessarily a problem. In fact, in some environments it's unavoidable. The reliability, availability, and manageability of a distributed system such as the one shown in are hard to beat. In all honesty, you aren't likely to use peer-to-peer technology to build a transaction-processing backbone for an e-commerce website. However, there are other situations that a server-based system can't deal with nearly as well. You'll see some of these examples at the end of this section.

Peer-to-peer technology aims to free applications of their dependence on a central server or group of servers, and it gives them the ability to create global communities, harness wasted CPU cycles, share isolated resources, and operate independently from central authorities. In peer-to-peer design, computers communicate directly with each other. Instead of a sharp distinction between servers that provide resources and clients that consume them, every computer becomes an equal peer that can exhibit clientlike behavior (making a request) and server-like behavior (filling a request). This increases the value of each computer on the network. No longer is it restricted to being a passive client consumer—a peer-to-peer node can participate in shared work or provide resources to other peers.

Peer-to-peer is most often defined as a technology that takes advantage of resources "at the edges of the network" because it bypasses the central server for direct interaction. As you can see in the following figure, this approach actually complicates the overall system.

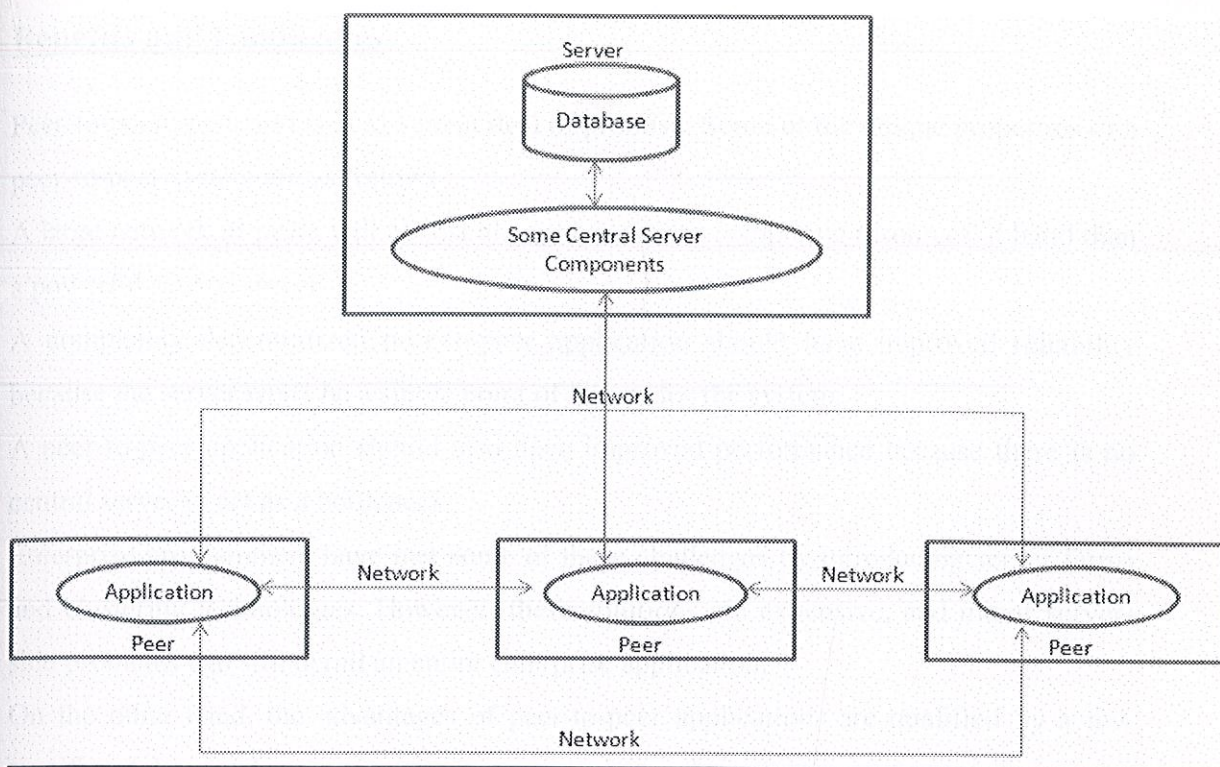


Fig 2.2.3: Peer to Peer Computing

Evaluating the Peer-to-Peer Model

The inevitable question is this: Can a peer-to-peer application perform better than a client-server application? Unfortunately, this question is not easily answered. It not only depends on the type of application, but on the type of peer-to-peer design, the number of users, and the overall traffic patterns. The most honest answer is probably this: There are some development niches in which peer-to-peer applications will perform better and require fewer resources. However, a peer-to-peer application can easily introduce new headaches and scalability challenges, which can't be dismissed easily. In order to create a successful peer-to-peer application, you must understand both the advantages and drawbacks of a peer-to-peer design.

Benefits and Challenges

Peer-to-peer applications hold a great deal of promise. Some of the unique properties of a peer-to-peer system are as follows:

A large network of peers will almost always have more computing resources at hand than a powerful central server.

A completely decentralized peer-to-peer application should have improved reliability because the server won't be a single point of failure for the system.

A peer-to-peer application should also have improved performance because there is no central server to act as a bottleneck.

Enterprise programmers have met some of these challenges by introducing server farms and clustering technologies. However, these solutions are expensive, and minor server-side problems can still derail an entire enterprise application.

On the other hand, the advantages of peer-to-peer applications are qualified by a few significant drawbacks:

As a peer-to-peer design becomes more decentralized, the code becomes more complex and the required network bandwidth to manage the peer discovery process increases. This might not be a problem if the bandwidth is spread out equitably over the network, but it often does become a problem in an intranet where the network must also be used for a critical client-server business application.

Although peer-to-peer systems don't rely on a central server, they do rely on the cooperation of multiple peers. This cooperating can be damaged by the variable connectivity of the Internet, where peers might abruptly disappear, even in the middle of serving a request. Similarly, in fully decentralized peer-to-peer systems, low-bandwidth clients can become "mini-bottlenecks" for their part of the network.

Peer-to-peer programming introduces significant challenges with network addressing due to the way the Internet works with dynamic IP addresses, proxy servers, network address translation (NAT), and firewalls.

It's also difficult to predict how a decentralized peer-to-peer solution will scale as the user community grows. Like all complex systems, a peer-to-peer network can display a dramatically different behavior at a certain "critical mass" of peers. Gnutella, a peer-to-

peer protocol used for popular file-sharing applications, is in some respects an enormous in-progress experiment. As the network has grown wildly beyond what was originally expected, connectivity has suffered—frequently. At times, entire islands of peers have broken off from the main pool, able to communicate within their community, but unable to access other parts of the Gnutella network.

Ingenious techniques such as smart caching and optimized routing have been developed to meet the challenges of large peer-to-peer networks. However, it's still hard to predict how these solutions will play out on a large scale over a loosely connected network that might include hundreds of thousands of peers. These emergent behaviors are impossible to plan for. The only way to solve them is with an iterative process of development that involves frequent testing and updates. Ultimately, a peer-to-peer system may become more robust and perform better than a classic enterprise application, but it will take ongoing development work.

Peer-to-Peer and Security

Security is a concern with any type of application, and peer-to-peer systems are no exception. The key difference is that with server-based programming, the server is in complete control. If the server adopts rigorous privacy standards and security safeguards, your information is safe, and you're in a "benevolent dictator" situation. However, if the server falls short of its commitment in any way, you'll have no protection.

In a decentralized peer-to-peer application, peers lack the protection of the server. On the one hand, they're also free from monitoring and have control of their private information. It's difficult to track an individual peer's actions, which remain publicly exposed, but lost in a sea of information. Nevertheless, malicious peers can connect directly to other peers to steal information or cause other types of problems.

Doing away with a central authority is both liberating and dangerous. For example, a malicious user can easily place a virus in a file-swapping peer-to-peer application disguised as another popular type of application and infect countless users, without being subject to any type of punishment or even being removed from the system. In addition,

the decentralized nature of peer discovery makes it difficult for an organization to enforce any kind of access control (short of blocking Internet access on certain ports). For these reasons, peer-to-peer application programmers need to consider security from the initial design stage. Some peer-to-peer applications handle security issues by allowing users to assign different levels of trust to certain peers. Other peer-to-peer systems rely on encryption to mask communication and certificates to validate peer identities.

Systems with Which Clients Need to Interact

The server-based model emphasizes one-way communication from the client to the server. That means that the client must initiate every interaction. This poses difficulty if you want to create a collaborative application such as a real-time chat, a multiplayer game, or a groupware application. With the introduction of a little peer-to-peer code, the problem becomes much more manageable.

Systems with Which Clients Need to Share Content

In the server-based system, everything needs to be routed through the central server. This taxes the computing power and network bandwidth of a small section of the overall network. Thus, you'll need a disproportionately powerful server to handle a relatively small volume of requests. If, however, the central server is used simply to locate other peers, it can become a "jumping off" point for a true peer-to-peer interaction, which is much more efficient. This is the infamous Napster model.

In some cases, you might use the file-sharing abilities of a peer-to-peer application to support other features.

Systems for Which a Central Server Would Be a Liability

This is generally the case if an application operated outside the bounds of local law (or in an area that could be subjected to future prosecution). For example, Napster, despite being partly peer-to-peer, required a central server for content lookup and for resolving

peer addresses, and was thus subjected to legal intervention. Gnutella, a more radically "pure" peer-to-peer application, isn't vulnerable in the same way. Similarly, consider the case of the legendary remailer anon.penet.fi, which was forced to close in 1996 because the anonymity of users could not be guaranteed against court orders that might have forced it to reveal account identities. Pure peer-to-peer systems, because they have no central server, are impervious to censorship and other forms of control.

Systems That Would Otherwise Be Prohibitively Expensive

You could build many peer-to-peer applications as server-based applications, but you'd require a significant hardware investment and ongoing work from a network support team to manage them. Depending on the type of application, this might not be realistic. For example, SETI@Home could not afford a supercomputer to chew through astronomical data in its search for unusual signals. However, by harnessing individual chunks of CPU time on a large network of peers, the same task could be completed in a sustainable, affordable way. Another example is a virtual file system that can provide terabytes of storage by combining small portions of an individual peer hard drive. In many ways, these applications represent the ideal peer-to-peer niche.

Thus, peer-to-peer applications don't always provide new features, but sometimes provide a more economical way to perform the same tasks as other application types. They allow specialized applications to flourish where the support would otherwise not exist. This includes every type of peer-to-peer application, from those that promote collaboration and content sharing, to those that work together to complete CPU-intensive tasks.

4.3 Peer-to-Peer Technologies in .NET

The .NET class library provides multiple technologies for communicating between computers. Some of these are layered on top of one another. You choose a high-level or low-level technology based on how much control you need and how much simplicity you would like.

Some of these technologies include-

The low-level networking classes in the System.Net.Sockets namespace, which wrap the Windows Sockets (Winsock) interface and allow you to create and access TCP channels directly. These classes are the heart of most peer-to-peer applications.

The networking classes in the System.Net namespace, which allow you to use a request or response access pattern with a URI over (HTTP).

The higher-level Remoting infrastructure, which allows you to interact with (or create) objects in other application domains using pluggable channels including TCP and HTTP and formats including binary encoding or SOAP. These classes can also be used in a peer-to-peer application, with some adaptation. The higher-level web services infrastructure, which provides fixed services as static class methods. This model is not suited for peer-to-peer communication, but is useful when creating a discovery service. In addition, there are several .NET technologies that you'll need to use and understand as part of any peer-to-peer application that isn't trivially simple. This includes threading, serialization, code access security, and encryption. You'll get a taste of all of these in this book. Finally, it's worth mentioning a few third-party tools that you'll see later in the report.

Windows Messenger is Microsoft's instant-messaging product. There is some published information available on the Windows Messenger protocol, and even a .NET library that allows you to harness it in your own software.

Groove is a platform for collaborative applications that manages the synchronization of a shared space. It's not free, but it's powerful, and shows how it can help your applications. Intel provides a free .NET Peer-to-Peer Accelerator Kit, which extends .NET's Remoting infrastructure with support for security, discovery, and limited firewall traversal. It's still an early product, but it promises to eliminate some of the connectivity headaches with peer-to-peer on the Web.

Peer-to-peer applications represent a fundamental shift from most other types of enterprise development and a return to some of the concepts that shaped the early Internet. They provide the key to collaboration and distributed computing, but require a new programming model that offers additional complexity and isn't yet built into development platforms such as .NET.

Peer-to-Peer Characteristics

One characteristic you won't find in the peer-to-peer world is consistency. The more you learn about different peer-to-peer applications, the more you'll see the same problems solved in different ways. This is typical of any relatively new programming model in which different ideas and techniques will compete in the field. In the future, peer-to-peer applications will probably settle on more common approaches. But even today, most of these techniques incorporate a few core ingredients, which are discussed in the following sections.

Peer Identity

In a peer-to-peer system, a peer's identity is separated into two pieces: a unique identifier, and a set of information specifying how to contact the peer. This separation is important—it allows users in a chat application to communicate based on user names, not IP addresses, and it allows peers to be tracked for a long period of time, even as their connection information changes. The connectivity information that you need depends on the way you are connecting with the peer, although it typically includes information such as a port number and IP address. The peer ID is a little trickier. How can you guarantee that each peer's identifier is unique on a large network that changes frequently?

There are actually two answers. One approach is to create a central component that stores a master list of user information. This is the model that chat applications such as Windows Messenger use. In this case, the central database needs to store authentication information as well, in order to ensure that peers are who they claim to be. It's an effective compromise, but a departure from pure peer-to-peer programming.

A more flexible approach is to let the application create a peer identifier dynamically. The best choice is to use a globally unique identifier (GUID). GUIDs are 128-bit integers that are represented in hexadecimal notation (for example, 382c74c3-721d-4f34-80e5-57657b6cbc27). The range of GUID values is such that a dynamically generated GUID is statistically unique—in other words, the chance of two randomly generated GUIDs having the same value is so astonishingly small that it can be ignored entirely.

In .NET, you can create GUIDs using the `System.Guid` structure. A peer can be associated with a new GUID every time it joins the network, or a GUID value can be

generated once and stored on the peer's local hard drive if you need a more permanent identity. Best of all, GUIDs aren't limited to identifying peers. They can also track tasks in a distributed-computing application or files in a file-sharing application. GUIDs can also be used to uniquely identify messages as they are routed around a decentralized peer-to-peer network, thereby ensuring that duplicate copies of the same message are ignored. Regardless of the approach you take, creating a peer-to-peer application involves creating a virtual namespace that maps peers to some type of peer identifier. Before you begin to code, you need to determine the type of peer identifier and the required peer connection information.

Peer Discovery

Another challenge in peer-to-peer programming is determining how peers find each other on a network. Because the community of peers always changes, joining the network is not as straightforward as connecting to a well-known server to launch a client-server application.

The most common method of peer discovery in .NET applications is to use a central discovery server, which will provide a list of peers that are currently online. In order for this approach to work, peers must contact the discovery server regularly and update their connectivity information. If no communication is received from a peer within a set amount of time, the peer is considered to be no longer active, and the peer record is removed from the server.

When a peer wants to communicate with another peer, it first contacts the discovery server to learn about other active peers. It might ask for a list of nearby peers, or supply a peer identifier and request the corresponding connectivity information it needs to connect to the peer. The peer-to-peer examples presented in the second and third part of this book all use some form of centralized server.

The discovery-server approach is the easiest way to quickly implement a peer-to-peer network, but it isn't suitable for all scenarios. In some cases, there is no fixed server or group of servers that can play the discovery role. In this case, peers need to use another form of discovery. Some options include Sending a network broadcast message to find

any nearby peers. This technique is limited because broadcast messages cannot cross routers from one network to another. Sending a multicast broadcast message to find nearby peers. This technique can cross networks, but it only works if the network supports multicasting. Reading a list of super-peers from some location (typically a text file or a web page), and trying to contact them directly. This requires a fixed location to post the peer information.

The last approach is not perfect, but it's the one most commonly used in decentralized peer-to-peer applications such as Gnutella.

The Server-Mode/Client-Mode Model

Peer to peer applications often play two roles, and act both as a client and server. For example, in a file-sharing application every interaction is really a client-server interaction in which a client requests a file and a server provides it. The difference with peer-to-peer applications is every peer can play both roles, usually with the help of threading code that performs each task simultaneously. This is known as the server-mode/client-mode (SM/CM) model.

Peer-to-Peer Topology

Peer-to-peer applications don't necessarily abolish the central server completely. In fact, there are a variety of peer-to-peer designs. Some are considered "pure" peer-to-peer, and don't include any central components, while others are hybrid designs.

Peer-to-Peer with a Discovery Server

One of the most common peer-to-peer compromises involves a discovery server, which is a repository that lists all the connected peers. Often, a discovery server maps user names to peer connectivity information such as an IP address. When users start the application, they're logged in and added to the registry. After this point, they must periodically contact the discovery server to confirm that they're logged in and that their connection information hasn't changed.

There is more than one way for peers to use the information in a discovery server. In a simple application, peers may simply download a list of nearby users and contact them directly with future requests. However, it's also possible that the peer will need to

communicate with a specific user (for example, in the case of a chat application). In this scenario, the discovery server can be structured to allow peer lookups by name, e-mail address, or some other fixed unique identifier. The peer interaction works like this:

1. The peer contacts the discovery server with a request to find the contact information for a specific user (for example, `someone@somewhere.com`).
2. The discovery server returns the user's IP address and port information.
3. The peer contacts the desired user directly.

This approach is also known as brokered or mediated peer-to-peer because the discovery server plays a central role in facilitating user interaction.

Peer-to-Peer with a Coordination Server

Some peer-to-peer applications benefit from a little more help on the server side. These applications combine peer-to-peer interaction with a central component that not only contains peer lookup information, but also includes some application-specific logic.

One example is Napster, which uses a central discovery and lookup server. In this system, peers register their available resources at periodic intervals. If a user needs to find a specific resource, the user queries the lookup server, which will then return a list of peers that have the desired resource. This helps to reduce network traffic and ensures that the peers don't waste time communicating if they have nothing to offer each other. The file-transfer itself is still peer-to-peer. This blend of peer-to-peer and traditional application design can greatly improve performance. By using a centralized server intelligently for a few critical tasks, network traffic can be reduced dramatically.

One question that arises with this sort of design is exactly how much responsibility the central server should assume. For example, you might create a messaging application in which communication is routed through the centralized server so that it can be analyzed or even logged. Similarly, you might design a content-sharing application that caches files on the server. These designs will add simplicity, but they can also lead to massive server bottlenecks for large peer-to-peer systems. As you'll discover in this book, a key

part of the art of peer-to-peer programming with .NET is choosing the right blend between pure peer-to-peer design and more traditional enterprise programming.

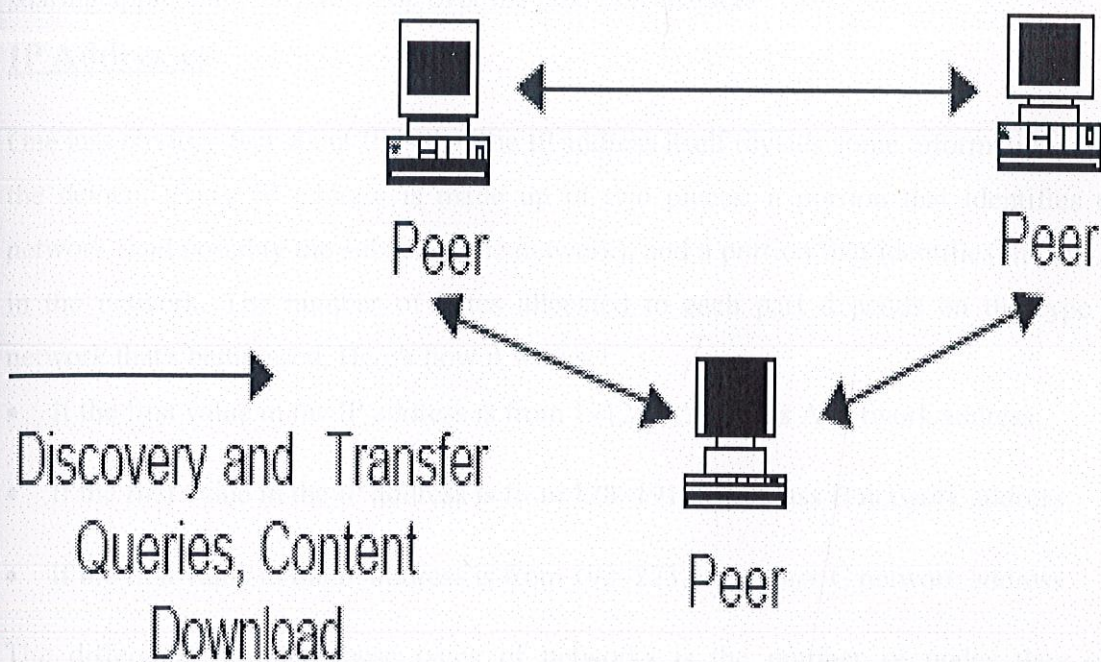


Fig 2.2.4: Pure Peer to Peer.

Peer-to-Peer with a Discovery Server

Networking Essentials

So far, we've used the Remoting infrastructure to communicate between applications. However, peer-to-peer applications often need to work at a lower level and take networking, sockets, and broadcasts into their own hands.

In this chapter, we'll cover the essentials of network programming with .NET. We'll start by reviewing the basics of physical networks and network protocols such as the Internet

Protocol (IP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP), and then consider the support that's built into the System.Net namespace. We'll also present sample applications that demonstrate how you can stream data across a network with a TCP or UDP connection. All of this is in preparation for the peer-to-peer file-sharing application we'll develop over the next two chapters.

IP Addresses

One less obvious fact about IP is that the IP address itself reveals some information about the device. Every IP address is made up of two pieces: a portion that identifies the network (and possibly the subnet of the network), and a portion that identifies the device in the network. The number of bytes allocated to each part depends on the type of network that's being used. Here's how it works:

- If the first value in the IP address is from 1–126, it's a class A network address.
- If the first value in the IP address is from 128–191, it's a class B network address.
- If the first value in the IP address is from 192–223, it's a class C network address.

The difference between these types of networks is the number of nodes they can accommodate. Class A addresses are used for extremely large networks that can accommodate over 16 million nodes. With a class A network, the first byte in the IP address is used to define the network and the remaining three bytes identify the host. It's only possible to have 126 class A networks worldwide, so only extremely large companies such as AT&T, IBM, and HP have class A networks. Thus, in the IP address 120.24.0.10, the number 120 identifies the network and the remaining values identify the device.

Class B addresses use the first two bytes to describe the network. There can be about 16,000 class B networks worldwide, each with a maximum of 65,534 devices. Thus, in the IP address 150.24.0.10, the value 150.24 identifies the network, and the 0.10 identifies the device.

Finally, class C networks use the first three bytes to describe a network. That leaves only one byte to identify the device. As a result, class C networks can hold only 254 devices.

Most companies that request an IP address will be assigned a class C IP address. If more devices are required, multiple class C networks can be used.

Note that this list leaves out some valid IP addresses because they have special meanings. Here's a summary of special IP addresses:

- 127.0.0.0 is a loopback address that always refers to the local network.
- 127.0.0.1 is a loopback address that refers to the current device.
- IP addresses that start with a number from 224–239 are used for multicasting.
- IP addresses that start with 240–255 are reserved for testing purposes.

In fact, there are already more devices connected to the Internet than there are available IP addresses. To compensate for this problem, devices that aren't connected to the Internet (or access the Internet through a gateway computer) can be given private IP addresses. Private IP addresses aren't globally unique. They're just unique within a network. All classes of networks reserve some values for private IP addresses, as follows:

- In a class A network, any address beginning with 10 is private.
- In a class B network, any address beginning with 172.16–172.31 is private.
- In a class C network, any address beginning with 192.168.0–192.168.255 is private.

Of course, a computer that's sheltered from the Internet doesn't need to use a private IP address—just about any IP address would do. Unfortunately, computers without an IP address can be difficult or impossible to contact from another network. This is one of the headaches of peer-to-peer programming.

Transmission Control Protocol and User Datagram Protocol

Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are two higher-level protocols that depend on IP. When you program an application, you won't create IP datagrams directly. Instead, you'll send information using TCP or UDP.

TCP is a connection-oriented protocol that has built-in flow control, error correction, and sequencing. Thanks to these features, you won't need to worry about resending information if a data collision occurs. You also won't have to worry about resolving any

one of the numerous possible network problems that could occur as information is segmented into packets and then transported and reassembled in its proper sequence at another computer. As a result, TCP is a fairly complex protocol with a certain amount of overhead built-in. However, it's also the favorite of most network programmers, and it's the protocol we'll use to transfer files with the application developed in the next two chapters.

UDP is a connectionless protocol for transferring data. It doesn't guarantee that messages will be received in sequence, that messages won't be lost, or that only one copy of a given message will be received. As a result, UDP is quite fast, but it requires a significant amount of work from the application programmer if you need to send important data. One reason UDP might be used in a peer-to-peer application is to support peer discovery. This is because UDP allows you to send messages to multiple nodes on the network at once, without necessarily knowing their IP address. This is possible through broadcasting and multicasting, two technologies introduced later in this chapter.

Ports

Both TCP and UDP introduce the concept of ports. Port numbers don't correspond to anything physical—they're simply a method for differentiating different application endpoints on the same computer. For example, if you're running a web server, your computer will respond to requests on port 80. Another application might use port 8000. Ports map connections to applications.

Port numbers are divided into three categories:

- Ports from 0–1023 are well-known system ports. They should only be used by a privileged system process (for example, part of the Windows operating system), not your application code.
- Ports from 1024–49151 are registered user ports. Your server applications can use one of these ports, although you may want to check that your choice doesn't conflict with a registered port number for an application that could be used on your server.

- Ports from 49152–65525 are dynamic ports. They're often used for ports that are allocated at runtime (for example, a local port a client might use when contacting a server).

Remember, every transmission over TCP or UDP involves two port numbers: one at the server end and one at the client end. The server port is generally the more important one. It's fixed in advance, and the server usually listens to it continuously. The client port is used to receive data sent from the server, and it can be chosen dynamically when the connection is initiated. A combination of port number and IP address makes an endpoint, or socket.

4.4 Networking In .NET

The .NET Framework includes two namespaces designed for network programming: System.Net and System.Net.Sockets. The System.Net namespace includes several classes that won't interest peer-to-peer programmers, including abstract base classes and types used to set Windows authentication credentials. However, there are also several noteworthy types:

- IPAddress represents a numeric IP address.
- IPEndPoint represents a combination of an IPAddress and port number. Taken together, these constitute a socket endpoint.
- Dns provides shared helper methods that allow you to resolve domain names (for example, you can convert a domain name into a number IP address, and vice versa).
- IPHostEntry associates a DNS host name with an array of IPAddress objects. Usually, you'll only be interested in retrieving the first IP address (in fact, in most cases there will only be one associated IP address).

- The `FileWebRequest` and `FileWebResponse` classes are useful when downloading a file from a URI (such as `file://ComputerName/ShareName/FileName`). However, we won't use these classes in this book.

The `System.Net.Sockets` class includes the types you'll need for socket programming with TCP or UDP. This namespace holds the most important functionality for the peer-to-peer programmer, including the following class types:

- `TcpListener` is used on the server side to listen for connections.
- `TcpClient` is used on the server and client side to transfer information over a TCP connection. Usually, you'll transmit data by reading and writing to the stream returned from `TcpClient.GetStream()`.
- `UdpClient` is used on the server and client to transfer information over a UDP connection, using methods such as `Send()` and `Receive()`.
- `Socket` represents the Berkeley socket used by both TCP and UDP. You can communicate using this socket directly, but it's usually easier to use the higher-level `TcpClient` and `UdpClient` classes.
- `SocketException` represents any error that occurs at the operating system level when attempting to establish a socket connection or send a message.
- `NetworkStream` is used with TCP connections. It allows you to send and receive data using a convention .NET stream, which is quite handy.

Network Streams

In .NET, you can send data over a TCP connection using the `NetworkStream` class, which follows the standard .NET streaming model. That means that you can write data to a `NetworkStream` in the same way that you would write bytes to a file. (It also means you can chain a `CryptoStream` onto your network stream for automatic encryption.)

The `NetworkStream` class differs slightly from other .NET streams because it represents a buffer of data that's just about to be sent or has just been received. On the sender's side, the buffer is emptied as data is sent across the network. On the recipient's side, the buffer

is emptied as data is read into the application. Because of this behavior, the `NetworkStream` class is not seekable, which means you cannot use the `Seek()` method or access the `Length` or `Position` properties.

In addition, the `NetworkStream` class adds a few useful properties:

- `Writable` and `Readable` indicate whether the `NetworkStream` supports write and read operations, respectively.
- `Socket` contains a reference to the underlying socket that's being used for data transmission.
- `DataAvailable` is a Boolean flag that's set to `True` when there's incoming data in the stream that you have not yet read.

The `Write()` and `Read()` methods allow you to copy byte arrays to and from the `NetworkStream`, but to simplify life you'll probably use the `BinaryWriter` and `BinaryReader` classes that are defined in the `System.IO` namespace. These classes can wrap any stream, and automatically convert common .NET types (such as strings, integers, and dates) into an array of bytes.

One good rule of thumb is to use the same approach for writing to a file as you do when reading it. For example, if you use the `BinaryWriter` to write data, use the `BinaryReader` to retrieve it, instead of the `NetworkStream.Read()` methods. This prevents you from introducing problems if you don't decode data the same way you encode it. For example, by default the `BinaryWriter` encodes data to binary using UTF-8 encoding. If you use `Unicode` to decode it, a problem could occur.

Communicating with TCP

TCP connections require a three-stage handshaking mechanism:

1. First, the server must enter listening mode by performing a passive open. At this point, the server will be idle, waiting for an incoming request.
2. A client can then use the IP address and port number to perform an active open. The server will respond with an acknowledgment message in a predetermined format that incorporates the client sequence number.

3. Finally, the client will respond to the acknowledgment. At this point, the connection is ready to transmit data in either direction.

In .NET, you perform the passive open with the server by using the `TcpListener.Start()` method. This method initializes the port, sets up the underlying socket, and begins listening, although it doesn't block your code. After this point, you can call the `Pending()` method to determine if any connection requests have been received. `Pending()` examines the underlying socket and returns `True` if there's a connection request. You can also call `AcceptTcpClient()` at any point to retrieve the connection request, or block the application until a connection request is received.

Communicating with UDP

In UDP, no connection needs to be created. As a result, there's no differentiation between client and server, and no listener class. Data can be sent immediately once the `UdpClient` object is created. However, you cannot use the `NetworkStream` to send messages with UDP. Instead, you must write binary data directly using the `Send()` and `Receive()` methods of the `UdpClient` class. Every time you use the `Send()` method, you specify three parameters: a byte array, the length of the byte array, and the `EndPoint` for the remote computer where the message will be sent.

The Discovery Service

A discovery service has one key task: to map peer identifiers to peer connectivity information. The peer identifier might be a unique user name or a dynamically generated identifier such as a globally unique identifier (GUID). The connectivity information includes all the details needed for another peer to create a direct connection. Typically, this includes an IP address and port number, although this information could be wrapped up in a higher-level construct. For example, the coordination server that we used in the Remoting chat application stores a proxy (technically, that encapsulates the IP address and port number as well as other details such as the remote class type and version).

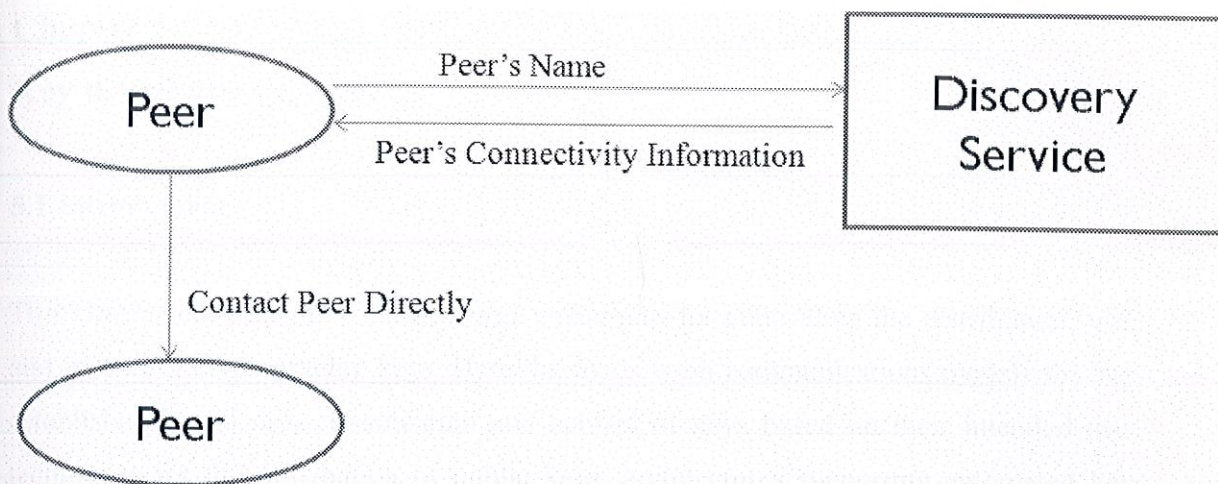


Fig 2.2.5: Basics of Discovery Service

In addition, a discovery service might provide information about the resources a peer provides. For example, in the file-sharing application, a peer creates a query based on a file name or keyword. The server then responds with a list of peers that can satisfy that request. In order to provide this higher-level service, the discovery service needs to store a catalog of peer information. This makes the system more dependent on its central component, and it limits the ways that you can search, because the central component must expect the types of searches and have all the required catalogs. However, if your searches are easy to categorize, this approach greatly improves performance and reduces network bandwidth.

Chapter 5

Key distribution Center

5.1 Introduction

This chapter considers key management techniques for controlling the distribution, use, and update of cryptographic keys. Here the focus is on communications models for key establishment and use, classification and control of keys based on their intended use, techniques for the distribution of public keys, architectures supporting automated key updates in distributed systems, and the roles of trusted third parties. Systems providing cryptographic services require techniques for initialization and key distribution as well as protocols to support on-line update of keying material, key backup/recovery, revocation, and for managing certificates in certificate-based systems. This chapter examines techniques related to these issues.

5.2 Background and basic concepts

A keying relationship is the state wherein communicating entities share common data (keying material) to facilitate cryptographic techniques. This data may include public or secret keys, initialization values, and additional non-secret parameters.

Key management is the set of techniques and procedures supporting the establishment and maintenance of keying relationships between authorized parties.

Key management encompasses techniques and procedures supporting:

1. Initialization of system users within a domain;
2. Generation, distribution, and installation of keying material;
3. Controlling the use of keying material;
4. Update, revocation, and destruction of keying material; and
5. Storage, backup/recovery, and archival of keying material.

5.2.1 Classifying keys by algorithm type and intended use

The terminology of Table 5.1 is used in reference to keying material. A symmetric cryptographic system is a system involving two transformations – one for the originator and one for the recipient – both of which make use of either the same secret key (symmetric key) or two keys easily computed from each other. An asymmetric cryptographic system is a system involving two related transformations – one defined by a public key (the public transformation), and another defined by a private key (the private transformation) – with the property that it is computationally infeasible to determine the private transformation from the public transformation.

Term	Meaning
private key, public key	paired keys in an asymmetric cryptographic system
symmetric key	key in a symmetric (single-key) cryptographic system
secret	adjective used to describe private or symmetric key

Table 5.1: Private, public, symmetric, and secret keys.

Table 5.2 indicates various types of algorithms commonly used to achieve the specified cryptographic objectives. Keys associated with these algorithms may be correspondingly classified, for the purpose of controlling key usage. The classification given requires specification of both the type of algorithm (e.g., encryption vs. signature) and the intended use (e.g., confidentiality vs. entity authentication).

5.2.2 Key management objectives, threats, and policy

Key management plays a fundamental role in cryptography as the basis for securing cryptographic techniques providing confidentiality, entity authentication, data origin authentication, data integrity, and digital signatures. The goal of a good cryptographic design is to reduce more complex problems to the proper management and safe-keeping of a small number of cryptographic keys, ultimately secured through trust in hardware or software by physical isolation or procedural controls. Reliance on physical and procedural security (e.g., secured rooms with isolated equipment), tamper-resistant

hardware, and trust in a large number of individuals is minimized by concentrating trust in a small number of easily monitored, controlled, and trustworthy elements.

Keying relationships in a communications environment involve at least two parties (a sender and a receiver) in real-time. In a storage environment, there may be only a single party, which stores and retrieves data at distinct points in time.

The objective of key management is to maintain keying relationships and keying material in a manner which counters relevant threats, such as:

1. compromise of confidentiality of secret keys.
2. compromise of authenticity of secret or public keys. Authenticity requirements include knowledge or verifiability of the true identity of the party a key is shared or associated with.
3. unauthorized use of secret or public keys. Examples include using a key which is no longer valid, or for other than an intended purpose.

In practice, an additional objective is conformance to a relevant security policy.

Security policy and key management

Key management is usually provided within the context of a specific security policy. A security policy explicitly or implicitly defines the threats a system is intended to address. The policy may affect the stringency of cryptographic requirements, depending on the susceptibility of the environment in question to various types of attack. Security policies typically also specify:

1. practices and procedures to be followed in carrying out technical and administrative aspects of key management, both automated and manual;
2. the responsibilities and accountability of each party involved; and
3. the types of records (audit trail information) to be kept, to support subsequent reports or reviews of security-related events.

5.2.3 Simple key establishment models

The following key distribution problem motivates more efficient key establishment models.

The n^2 key distribution problem

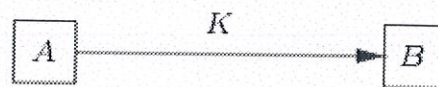
In a system with n users involving symmetric-key techniques, if each pair of users may potentially need to communicate securely, then each pair must share a distinct secret key. In this case, each party must have $n - 1$ secret keys; the overall number of keys in the system, which may need to be centrally backed up, is then $n(n - 1)/2$, or approximately n^2 . As the size of a system increases, this number becomes unacceptably large.

In systems based on symmetric-key techniques, the solution is to use centralized key servers: a star-like or spoked-wheel network is set up, with a trusted third party at the center or hub of communications (see Remark 5.3). This addresses the n^2 key distribution problem, at the cost of the requirement of an on-line trusted server, and additional communications with it. Public-key techniques offer an alternate solution.

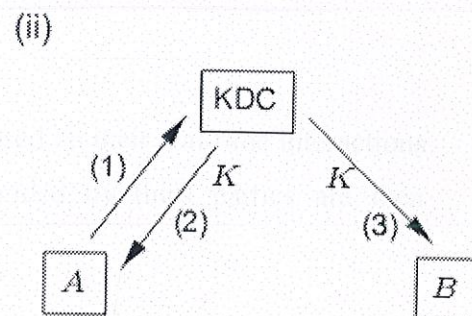
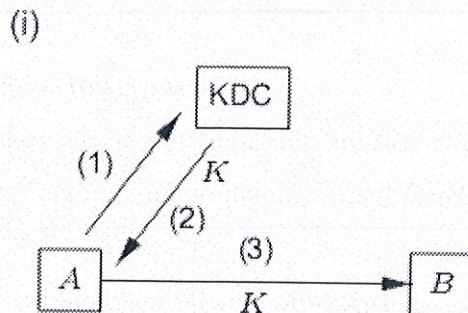
Point-to-point and centralized key management

Point-to-point communications and centralized key management, using key distribution centers or key translation centers, are examples of simple key distribution (communications) models relevant to symmetric-key systems. Here “simple” implies involving at most one third party. These are illustrated in Figure 5.1 and described below, where K_{XY} denotes a symmetric key shared by X and Y .

(a) Point-to-point key distribution



(b) Key distribution center (KDC)



(c) Key translation center (KTC)

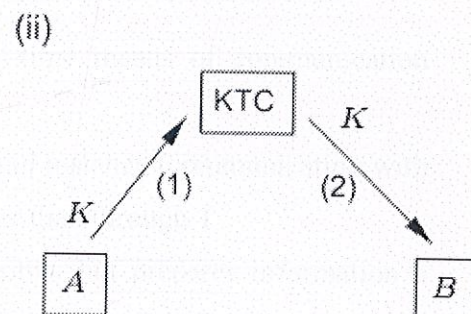
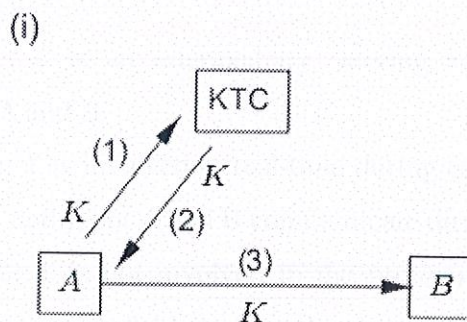


Figure 5.1: Simple key distribution models (symmetric-key).

1. point-to-point mechanisms. These involve two parties communicating directly.
2. key distribution centers (KDCs). KDCs are used to distribute keys between users which share distinct keys with the KDC, but not with each other. A basic KDC protocol proceeds as follows. Upon request from A to share a key with B, the KDC T generates or otherwise acquires a key K, then sends it encrypted under K_{AT} to A, along with a copy of K (for B) encrypted under K_{BT} . Alternatively, T may communicate K (secured under K_{BT}) to B directly.

3. key translation centers (KTCs). The assumptions and objectives of KTCs are as for KDCs above, but here one of the parties (e.g., A) supplies the session key rather than the trusted center. A basic KTC protocol proceeds as follows. A sends a key K to the KTC T encrypted under K_{AT} . The KTC deciphers and re-enciphers K under K_{BT} , then returns this to A (to relay to B) or sends it to B directly.

KDCs provide centralized key generation, while KTCs allow distributed key generation. Both are centralized techniques in that they involve an on-line trusted server.

5.2.4 Roles of third parties

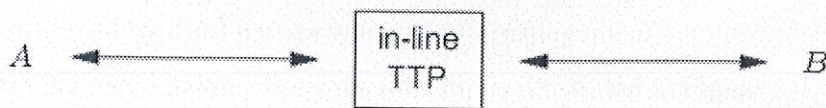
Below, trusted third parties (TTPs) are first classified based on their real-time interactions with other entities. Key management functions provided by third parties are then discussed.

(i) In-line, on-line, and off-line third parties

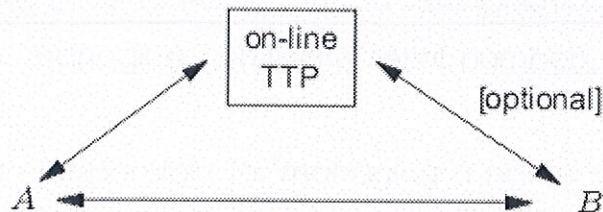
From a communications viewpoint, three categories of third parties T can be distinguished based on relative location to and interaction with the communicating parties A and B :

1. in-line: T is an intermediary, serving as the real-time means of communication between A and B.
2. on-line: T is involved in real-time during each protocol instance (communicating with A or B or both), but A and B communicate directly rather than through T .
3. off-line: T is not involved in the protocol in real-time, but prepares information a priori, which is available to A or B or both and used during protocol execution.

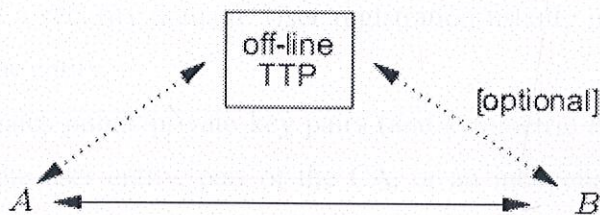
(a) in-line



(b) on-line



(c) off-line



..... communications carried out prior to protocol run

Figure 5.2: In-line, on-line, and off-line third parties.

In-line third parties are of particular interest when A and B belong to different security domains or cannot otherwise interact directly due to non-interoperable security mechanisms. Examples of an in-line third party include a KDC or KTC which provides the communications path between A and B, as in Figure 5.1(b)(ii) or (c)(ii). Parts (b)(i) and (c)(i) illustrate examples of on-line third parties which are not in-line. An example of an off-line third party is a certification authority producing public-key certificates and placing them in a public directory; here, the directory may be an on-line third party, but the certification authority is not.

(ii) Third party functions related to public-key certificates

Potential roles played by third parties within a key management system involving public-key certificates are listed below. Their relationship is illustrated in Figure 5.3.

1. certification authority (CA) – responsible for establishing and vouching for the authenticity of public keys. In certificate-based systems (x5.4.2), this includes binding public keys to distinguished names through signed certificates, managing certificate serial numbers, and certificate revocation.

2. name server – responsible for managing a name space of unique user names (e.g., unique relative to a CA).

3. registration authority – responsible for authorizing entities, distinguished by unique names, as members of a security domain. User registration usually involves associating keying material with the entity.

4. key generator – creates public/private key pairs (and symmetric keys or passwords). This may be part of the user entity, part of the CA, or an independent trusted system component.

5. certificate directory – a certificate database or server accessible for read-access by users. The CA may supply certificates to (and maintain) the database, or users may manage their own database entries (under appropriate access control).

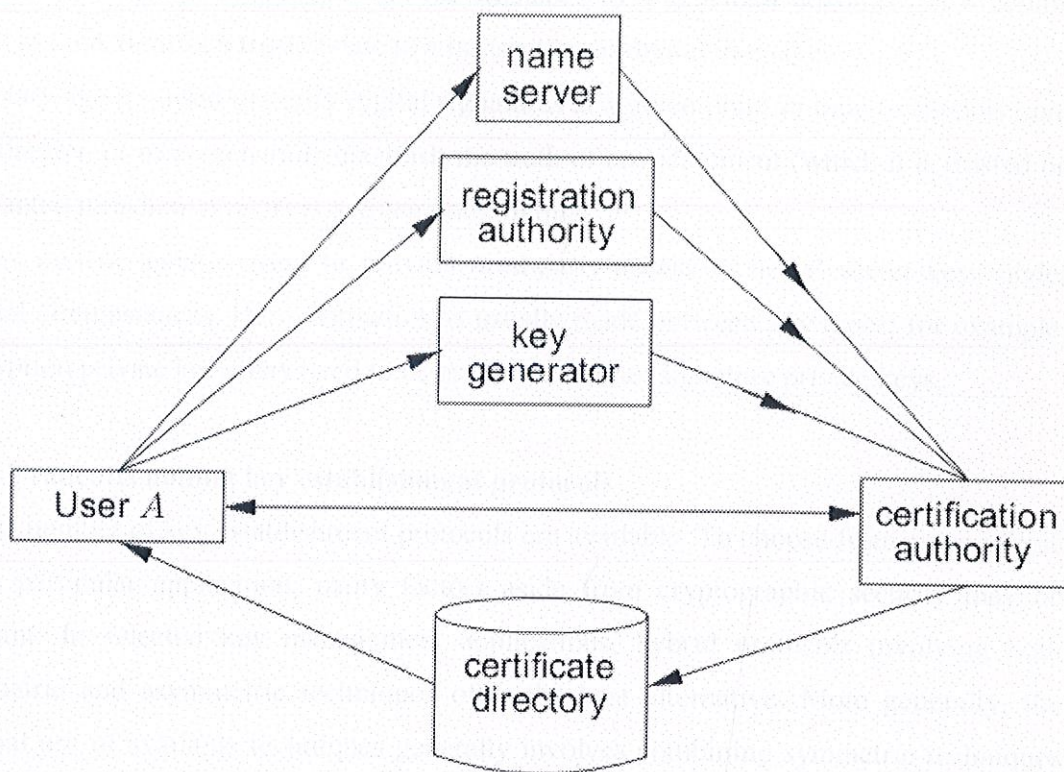


Figure 5.3: Third party services related to public-key certification.

(iii) Other basic third party functions

Additional basic functions a trusted third party may provide include:

1. key server (authentication server) – facilitates key establishment between other parties, including for entity authentication. Examples include KDCs and KTCs.
2. key management facility – provides a number of services including storage and archival of keys, audit collection and reporting tools, and (in conjunction with a certification authority or CA) enforcement of life cycle requirements including updating and revoking keys. The associated key server or certification authority may provide a record (audit trail) of all events related to key generation and update, certificate generation and revocation, etc.

(iv) Advanced third party functions

Advanced service roles which may be provided by trusted third parties, include:

1. timestamp agent – used to assert the existence of a specified document at a certain point in time, or affix a trusted date to a transaction or digital message.
2. notary agent – used to verify digital signatures at a given point in time to support non-repudiation, or more generally establish the truth of any statement (which it is trusted on or granted jurisdiction over) at a given point in time.
3. key escrow agent – used to provide third-party access to users' secret keys under special circumstances. Here distinction is usually made between key types; for example, encryption private keys may need to be escrowed but not signature private keys.

5.2.5 Tradeoffs among key establishment protocols

A vast number of key establishment protocols are available. To choose from among these for a particular application, many factors aside from cryptographic security may be relevant. In selected key management applications, hybrid protocols involving both symmetric and asymmetric techniques offer the best alternative. More generally, the optimal use of available techniques generally involves combining symmetric techniques for bulk encryption and data integrity with public-key techniques for signatures and key management.

Public-key vs. symmetric-key techniques (in key management)

Primary advantages offered by public-key (vs. symmetric-key) techniques for applications related to key management include:

1. simplified key management. To encrypt data for another party, only the encryption public key of that party need be obtained. This simplifies key management as only authenticity of public keys is required, not their secrecy. Table 5.3 illustrates the case for encryption keys. The situation is analogous for other types of public-key pairs, e.g., signature key pairs.
2. on-line trusted server not required. Public-key techniques allow a trusted on-line server to be replaced by a trusted off-line server plus any means for delivering authentic public keys (e.g., public-key certificates and a public database provided by an untrusted on-line server). For applications where an on-line trusted server is not mandatory, this may make the system more amenable to scaling, to support very large numbers of users.

3. enhanced functionality. Public-key cryptography offers functionality which typically cannot be provided cost-effectively by symmetric techniques (without additional on-line trusted third parties or customized secure hardware). The most notable such features are non-repudiation of digital signatures, and true (single-source) data origin authentication.

	Symmetric keys		Asymmetric keys	
	secrecy	authenticity	secrecy	authenticity
encryption key	yes	yes	no	yes
decryption key	yes	yes	yes	yes

Table 5.3: Key protection requirements: symmetric-key vs. public-key systems.

Figure 5.4 compares key management for symmetric-key and public-key encryption. The pairwise secure channel in Figure 5.4(a) is often a trusted server with which each party communicates. The pairwise authentic channel in Figure 5.4(b) may be replaced by a public directory through which public keys are available via certificates; the public key in this case is typically used to encrypt a symmetric data key.

5.3 Techniques for distributing confidential keys

Various techniques and protocols are available to distribute cryptographic keys whose confidentiality must be preserved (both private keys and symmetric keys). These include the use of key layering and symmetric-key certificates.

5.3.1 Key layering and cryptoperiods

Table 5.2 may be used to classify keys based on usage. The class "confidentiality" may be sub-classified on the nature of the information being protected: user data vs keying material. This suggests a natural key layering as follows:

1. master keys – keys at the highest level in the hierarchy, in that they themselves are not cryptographically protected. They are distributed manually or initially installed and protected by procedural controls and physical or electronic isolation.

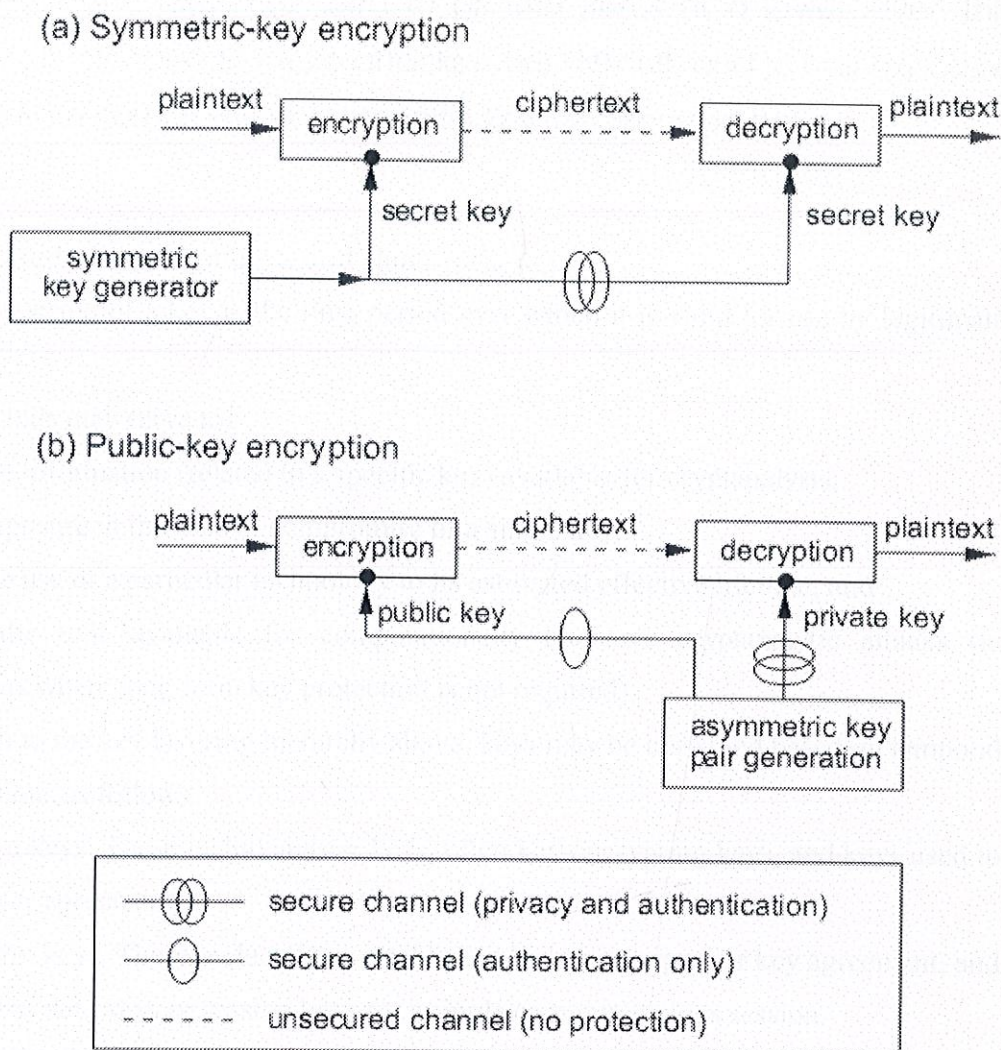


Figure 5.4: Key management: symmetric-key vs. public-key encryption.

2. key-encrypting keys – symmetric keys or encryption public keys used for key transport or storage of other keys. These may also be called key-transport keys, and may themselves be secured under other keys.
3. data keys – used to provide cryptographic operations on user data (e.g., encryption, authentication). These are generally short-term symmetric keys; however, asymmetric signature private keys may also be considered data keys, and these are usually longer-term keys.

The keys at one layer are used to protect items at a lower level. This constraint is intended to make attacks more difficult, and to limit exposure resulting from compromise of a specific key, as discussed below.

Cryptoperiods, long-term keys, and short-term keys

The cryptoperiod of a key is the time period over which it is valid for use by legitimate parties.

Cryptoperiods may serve to:

1. limit the information (related to a specific key) available for cryptanalysis;
2. limit exposure in the case of compromise of a single key;
3. limit the use of a particular technology to its estimated effective lifetime; and
4. limit the time available for computationally intensive cryptanalytic attacks (in applications where long-term key protection is not required).

In addition to the key layering hierarchy above, keys may be classified based on temporal considerations as follows.

1. long-term keys. These include master keys, often key-encrypting keys, and keys used to facilitate key agreement.
2. short-term keys. These include keys established by key transport or key agreement, and often used as data keys or session keys for a single communications session.

In general, communications applications involve short-term keys, while data storage applications require longer-term keys. Long-term keys typically protect short-term keys. Diffie-Hellman keys are an exception in some cases. Cryptoperiods limit the use of keys to fixed periods, after which they must be replaced.

5.3.2 Key translation centers and symmetric-key certificates

Further to centralized key management discussed in 5.2.3, this section considers techniques involving key translation centers, including use of symmetric-key certificates.

(i) Key translation centers

A key translation center (KTC) *T* is a trusted server which allows two parties *A* and *B*, which do not directly share keying material, to establish secure communications through

use of long-term keys K_{AT} and K_{BT} they respectively share with T . A may send a confidential message M to B using Protocol. If M is a key K , this provides a key transfer protocol; thus, KTCs provide translation of keys or messages.

Protocol Message translation protocol using a KTC

SUMMARY: A interacts with a trusted server (KTC) T and party B .

RESULT: A transfers a secret message M (or session key) to B . See Note 5.5.

1. Notation. E is a symmetric encryption algorithm. M may be a session key K .
2. One-time setup. A and T share key K_{AT} . Similarly B and T share K_{BT} .
3. Protocol messages.

$$A \rightarrow T : A, E_{K_{AT}}(B, M) \quad (1)$$

$$A \leftarrow T : E_{K_{BT}}(M, A) \quad (2)$$

$$A \rightarrow B : E_{K_{BT}}(M, A) \quad (3)$$

4. Protocol actions.

- (a) A encrypts M (along with the identifier of the intended recipient) under K_{AT} , and sends this to T with its own identifier (to allow T to look up K_{AT}).
- (b) Upon decrypting the message, T determines it is intended for B , looks up the key (K_{BT}) of the indicated recipient, and re-encrypts M for B .
- (c) T returns the translated message for A to send to (or post in a public site for) B ; alternatively, T may send the response to B directly.

Only one of A and B need communicate with T . As an alternative to the protocol as given, A may send the first message to B directly, which B would then relay to T for translation, with T responding directly to B .

(ii) Symmetric-key certificates

Symmetric-key certificates provide a means for a KTC to avoid the requirement of either maintaining a secure database of user secrets (or duplicating such a database for multiple servers), or retrieving such keys from a database upon translation requests.

As before, associated with each party B is a key K_{BT} shared with T , which is now embedded in a symmetric-key certificate $E_{KT}(K_{BT}, B)$ encrypted under a symmetric master key K_T known only to T . (A lifetime parameter L could also be included in the certificate as a validity period.) The certificate serves as a memo from T to itself (who

alone can open it), and is given to B so that B may subsequently present it back to T precisely when required to access B's symmetric key K_{BT} for message translation. Rather than storing all user keys, T now need securely store only K_T .

Symmetric-key certificates may be used in Protocol 5.12 by changing only the first message as below, where $SCert_A = E_{K_T}(K_{AT}, A)$, $SCert_B = E_{K_T}(K_{BT}, B)$:

$$A \rightarrow T : SCert_A, EK_{AT}(B, M), SCert_B (1)$$

A public database may be established with an entry specifying the name of each user and its corresponding symmetric-key certificate. To construct message (1), A retrieves B's symmetric-key certificate and includes this along with its own. T carries out the translation as before, retrieving K_{AT} and K_{BT} from these certificates, but now also verifies that A's intended recipient B as specified in $E_{K_{AT}}(B, M)$ matches the identifier in the supplied certificate $SCert_B$.

5.4 Key life cycle issues

Key management is simplest when all cryptographic keys are fixed for all time. Cryptoperiods necessitate the update of keys. This imposes additional requirements, e.g., on certification authorities which maintain and update user keys. The set of stages through which a key progresses during its existence, referred to as the life cycle of keys, is discussed in this section.

5.4.1 Lifetime protection requirements

Controls are necessary to protect keys both during usage and storage. Regarding long-term storage of keys, the duration of protection required depends on the cryptographic function (e.g., encryption, signature, data origin authentication/integrity) and the time-sensitivity of the data in question.

Security impact of dependencies in key updates

Keying material should be updated prior to cryptoperiod expiry. Update involves use of existing keying material to establish new keying material, through appropriate key establishment protocols and key layering.

To limit exposure in case of compromise of either long term secret keys or past session keys, dependencies among keying material should be avoided. For example, securing a new session key by encrypting it under the old session key is not recommended (since compromise of the old key compromises the new). See x12.2.3 regarding perfect forward secrecy and known-key attacks.

Lifetime storage requirements for various types of keys

Stored secret keys must be secured so as to provide both confidentiality and authenticity. Stored public keys must be secured such that their authenticity is verifiable. Confidentiality and authenticity guarantees, respectively countering the threats of disclosure and modification, may be provided by cryptographic techniques, procedural (trust-based) techniques, or physical protection (tamper-resistant hardware). Signature verification public keys may require archival to allow signature verification at future points in time, including possibly after the private key ceases to be used. Some applications may require that signature private keys neither be backed up nor archived: such keys revealed to any party other than the owner potentially invalidates the property of non repudiation. Note here that loss (without compromise) of a signature private key may be addressed by creation of a new key, and is non-critical as such a private key is not needed for access to past transactions; similarly, public encryption keys need not be archived. On the other hand, decryption private keys may require archival, since past information encrypted thereunder might otherwise be lost.

Keys used for entity authentication need not be backed up or archived. All secret keys used for encryption or data origin authentication should remain secret for as long as the data secured thereunder requires continued protection (the protection lifetime), and backup or archival is required to prevent loss of this data or verifiability should the key be lost.

5.4.2 Key management life cycle

Except in simple systems where secret keys remain fixed for all time, cryptoperiods associated with keys require that keys be updated periodically. Key update necessitates additional procedures and protocols, often including communications with third parties in

public-key systems. The sequence of states which keyingmaterial progresses through over its lifetime is called the key management life cycle. Life cycle stages, as illustrated, may include:

1. user registration – an entity becomes an authorized member of a security domain. This involves acquisition, or creation and exchange, of initial keyingmaterial such as shared passwords or PINs by a secure, one-time technique (e.g., personal exchange, registered mail, trusted courier).

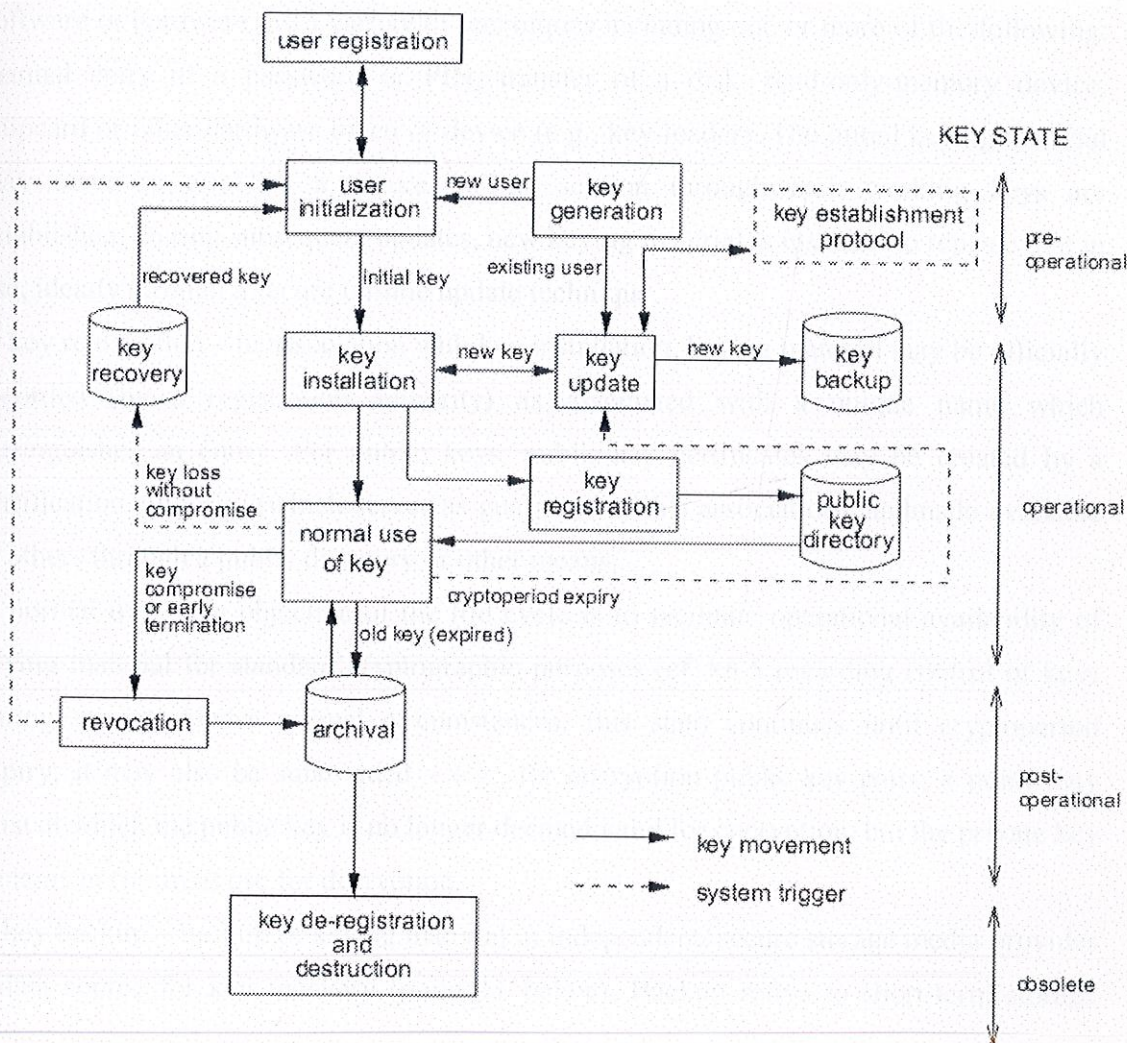


Figure 5.5: Key management life cycle.

2. user initialization – an entity initializes its cryptographic application (e.g., installs and initializes software or hardware), involving use or installation of initial keying material obtained during user registration.
3. key generation – generation of cryptographic keys should include measures to ensure appropriate properties for the intended application or algorithm and randomness in the sense of being predictable (to adversaries) with negligible probability. An entity may generate its own keys, or acquire keys from a trusted system component.
4. key installation – keying material is installed for operational use within an entity's software or hardware, by a variety of techniques including one or more of the following: manual entry of a password or PIN, transfer of a disk, read-only-memory device, chipcard or other hardware token or device (e.g., key-loader). The initial keying material may serve to establish a secure on-line session through which working keys are established. During subsequent updates, new keying material is installed to replace that in use, ideally through a secure on-line update technique.
5. key registration – in association with key installation, keying material may be officially recorded (by a registration authority) as associated with a unique name which distinguishes an entity. For public keys, public-key certificates may be created by a certification authority (which serves as guarantor of this association), and made available to others through a public directory or other means.
6. normal use – the objective of the life cycle is to facilitate operational availability of keying material for standard cryptographic purposes (cf. x5.5 regarding control of keys during usage). Under normal circumstances, this state continues until cryptoperiod expiry; it may also be subdivided – e.g., for encryption public-key pairs, a point may exist at which the public key is no longer deemed valid for encryption, but the private key remains in (normal) use for decryption.
7. key backup – backup of keying material in independent, secure storage media provides a data source for key recovery (point 11 below). Backup refers to short-term storage during operational use.
8. key update – prior to cryptoperiod expiry, operational keying material is replaced by new material. This may involve some combination of key generation, key derivation, execution of two-party key establishment protocols, or communications with a trusted

third party. For public keys, update and registration of new keys typically involves secure communications protocols with certification authorities.

9. archival – keying material no longer in normal use may be archived to provide a source for key retrieval under special circumstances (e.g., settling disputes involving repudiation). Archival refers to off-line long-term storage of post-operational keys.

10. key de-registration and destruction – once there are no further requirements for the value of a key or maintaining its association with an entity, the key is de-registered (removed from all official records of existing keys), and all copies of the key are destroyed. In the case of secret keys, all traces are securely erased.

11. key recovery – if keying material is lost in a manner free of compromise (e.g., due to equipment failure or forgotten passwords), it may be possible to restore the material from a secure backup copy.

12. key revocation – it may be necessary to remove keys from operational use prior to their originally scheduled expiry, for reasons including key compromise. For public keys distributed by certificates, this involves revoking certificates.

Of the above stages, all are regularly scheduled, except key recovery and key revocation which arise under special situations.

Key states within life cycle

The typical events involving keying material over the lifetime of the key define stages of the life cycle. These may be grouped to define a smaller set of states for cryptographic keys, related to their availability for use. One classification of key states is as follows :

1. pre-operational. The key is not yet available for normal cryptographic operations.
2. operational. The key is available, and in normal use.
3. post-operational. The key is no longer in normal use, but off-line access to it is possible for special purposes.
4. obsolete. The key is no longer available. All records of the key value are deleted.

System initialization and key installation

Key management systems require an initial keying relationship to provide an initial secure channel and optionally support the establishment of subsequent working keys (long-term and short-term) by automated techniques. The initialization process typically involves non-cryptographic one-time procedures such as transfer of keying material in

person, by trusted courier, or over other trusted channels. The security of a properly architected system is reduced to the security of keying material, and ultimately to the security of initial key installation. For this reason, initial key installation may involve dual or split control, requiring co-operation of two or more independent trustworthy parties.

5.5 KDC – The Implementation

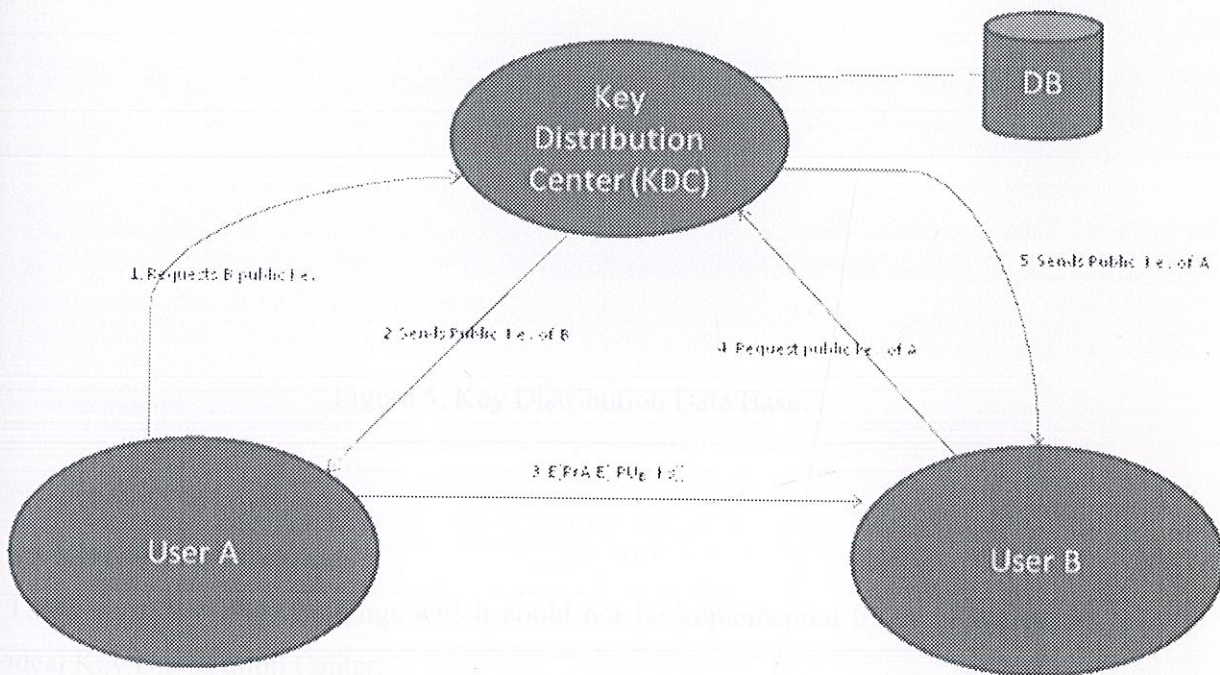


Figure 5. Key Distribution Center.

KDC Data-Base

User ID	Public Key	Private Key	Timestamp
Bharat	Xml in byte[]	Xml in byte[]	datetime()

Figure 5. Key Distribution Data Base.

5.6 KDC – Short Comings

There were few short comings which could not be implemented by us as compared to ideal Key Distribution Center.

1. Nonce was not appended in the steps.
2. KDC does not digitally sign the message it sends.

Chapter 6

Source Code

6.1 Main Screen Coding

Main Screen Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace TestApp
{
    public partial class Form1 : Form
    {
        string path;

        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            System.Diagnostics.Process.Start("TextEncryption\\TextEncryption.exe");
            Application.Exit();
        }
    }
}
```

```

    }

    private void button2_Click(object sender, EventArgs e)
    {
        System.Diagnostics.Process.Start("DESAnalysis\\DESAnalysis.exe");
        Application.Exit();
    }

    private void button3_Click(object sender, EventArgs e)
    {
        System.Diagnostics.Process.Start("KDC\\KDC Server\\KDC
Server.exe");
        Application.Exit();
    }

    private void button4_Click(object sender, EventArgs e)
    {
        System.Diagnostics.Process.Start("KDC\\KDC Client\\KDC
Client.exe");
        Application.Exit();
    }

    private void button5_Click(object sender, EventArgs e)
    {
        System.Diagnostics.Process.Start("FileEncryption\\FileEncryption.exe");
        Application.Exit();
    }

    private void button6_Click(object sender, EventArgs e)
    {
        ServerSettings a = new ServerSettings();
        a.Show();
    }
}

```

6.2 Server Setting Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace TestApp
{
    public partial class ServerSettings : Form
    {
        public ServerSettings()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            string server = servername.Text;

            FileStream a = new FileStream("settings.txt",
            FileMode.Create, FileAccess.Write);
            StreamWriter op = new StreamWriter(a);

            op.WriteLine(server);
            op.Close();
            a.Close();

            this.Close();
        }
    }
}
```

6.3 Text Encryption Coding

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;
using System.IO;

namespace TextEncryption
{
    public partial class TextEncrypt : Form
    {
        //for asymmetric ciphers
        RSAParameters rsaparams;
        DSAParameters dsaparams;
        byte[] rsacipherbyte;
        byte[] rsasignaturebytes;
        byte[] dsasignaturebytes;

        //for symmetric cipher
        CipherMode mode;
        PaddingMode padding;
        byte[] key;
        byte[] IV;
        byte[] cipherbytes;

        public TextEncrypt()
        {
            InitializeComponent();
        }

        private void RijndaelButton_CheckedChanged(object sender,
        EventArgs e)
        {

```

```

        if (RijndaelButton.Checked == true)
        {
            ZerosButton.Enabled = false;
            NoneButton.Enabled = false;
            PKCS7Button.Checked = true;
        }
        else
        {
            ZerosButton.Enabled = true;
        }

        KeyTextBox.Text = "";
        Encryptbutton.Enabled = true;
        PlainTextBox.Text = "";
        CipherTextBox1.Text = "";
        DecryptTextBox.Text = "";
        Decryptbutton.Enabled = false;
    }

    private void EstablishMode()
    {
        if (ECBButton.Checked == true)
            mode = CipherMode.ECB;
        if (CBCButton.Checked == true)
            mode = CipherMode.CBC;
        if (CFBButton.Checked == true)
            mode = CipherMode.CFB;
        if (CTSButton.Checked == true)
            mode = CipherMode.CTS;
    }

    private void EstablishPadding()
    {
        if (PKCS7Button.Checked == true)
            padding = PaddingMode.PKCS7;
        if (ZerosButton.Checked == true)
            padding = PaddingMode.Zeros;
        if (NoneButton.Checked == true)

```

```

        padding = PaddingMode.None;
    }

    private void TextEncrypt_Load(object sender, EventArgs e)
    {
    }

    SymmetricAlgorithm CreateSymmetricAlgorithm()
    {
        if (DESButton.Checked == true)
            return DES.Create();
        if (RijndaelButton.Checked == true)
            return Rijndael.Create();
        if (RC2Button.Checked == true)
            return RC2.Create();
        if (TDESButton.Checked == true)
            return TripleDES.Create();
        return null;
    }

    private void Keybutton_Click(object sender, EventArgs e)
    {
        Encryptbutton.Enabled = true;

        if (DESButton.Checked == true)
        {
            string t = Convert.ToString(KeyTextBox.Text);
            if (t.Length == 8)
                key = Encoding.UTF8.GetBytes(KeyTextBox.Text);
            else
                MessageBox.Show("Please Enter \"64\" Bit Key (8 Chars)");
        }

        if (TDESButton.Checked == true)
        {
            string t = Convert.ToString(KeyTextBox.Text);
            if (t.Length == 24)

```

```

        key = Encoding.UTF8.GetBytes(KeyTextBox.Text);
    else
        MessageBox.Show("Please Enter \"192\" Bit Key (24
Chars)");
    }

    if (RijndaelButton.Checked == true)
    {
        string t = Encoding.ToString(KeyTextBox.Text);
        if (t.Length == 32)
            key = Encoding.UTF8.GetBytes(KeyTextBox.Text);
        else
            MessageBox.Show("Please Enter \"256\" Bit Key (32
Chars)");
    }

    if (RC2Button.Checked == true)
    {
        string t = Encoding.ToString(KeyTextBox.Text);
        if (t.Length == 16)
            key = Encoding.UTF8.GetBytes(KeyTextBox.Text);
        else
            MessageBox.Show("Please Enter \"128\" Bit Key (16
Chars)");
    }
}

private void Encryptbutton_Click(object sender, EventArgs e)
{
    CipherTextBox1.Text = "";

    EstablishMode();
    EstablishPadding();

    SymmetricAlgorithm sa = CreateSymmetricAlgorithm();

    sa.GenerateIV();
    if (KeyTextBox.Text == "")

```

```

        sa.GenerateKey();
    else
        sa.Key = key;

    key = sa.Key;
    IV = sa.IV;
    sa.Mode = mode;
    sa.Padding = padding;

    MemoryStream ms = new MemoryStream();
    CryptoStream cs = new CryptoStream(ms,
sa.CreateEncryptor(), CryptoStreamMode.Write);

    byte[] plaintext =
Encoding.UTF8.GetBytes(PlainTextBox.Text);
    cs.Write(plaintext, 0, plaintext.Length);
    cs.Close();

    cipherbytes = ms.ToArray();
    ms.Close();

    CipherTextBox1.Text = Convert.ToBase64String(cipherbytes);

    Decryptbutton.Enabled = true;
    button2.Enabled = true;
}

private void Decryptbutton_Click(object sender, EventArgs e)
{
    DecryptTextBox.Text = "";

    EstablishMode();
    EstablishPadding();

    SymmetricAlgorithm sa = CreateSymmetricAlgorithm();

    sa.Key = key;
    sa.IV = IV;

```

```

        sa.Mode = mode;
        sa.Padding = padding;

        MemoryStream ms = new MemoryStream(cipherbytes);
        CryptoStream cs = new CryptoStream(ms,
sa.CreateDecryptor(), CryptoStreamMode.Read);

        byte[] plainbyte = new Byte[cipherbytes.Length];
        cs.Read(plainbyte, 0, cipherbytes.Length);
        cs.Close();
        ms.Close();

        char[] a = Encoding.UTF8.GetChars(plainbyte);
        for (int q = 0; q < a.Length; q++)
            DecryptTextBox.Text += Convert.ToString(a[q]);
    }

    private void KeyTextBox_TextChanged(object sender, EventArgs e)
    {
        if (KeyTextBox.Text != "")
            Encryptbutton.Enabled = false;
        else
            Encryptbutton.Enabled = true;
    }

    private void TDESButton_CheckedChanged(object sender, EventArgs
e)
    {
        KeyTextBox.Text = "";
        Encryptbutton.Enabled = true;
        PlainTextBox.Text = "";
        CipherTextBox1.Text = "";
        DecryptTextBox.Text = "";
        Decryptbutton.Enabled = false;
    }

    private void RC2Button_CheckedChanged(object sender, EventArgs
e)

```

```

{
    KeyTextBox.Text = "";
    Encryptbutton.Enabled = true;
    PlainTextBox.Text = "";
    CipherTextBox1.Text = "";
    DecryptTextBox.Text = "";
    Decryptbutton.Enabled = false;
}

```

e) private void DESButton_CheckedChanged(object sender, EventArgs e)

```

{
    KeyTextBox.Text = "";
    Encryptbutton.Enabled = true;
    PlainTextBox.Text = "";
    CipherTextBox1.Text = "";
    DecryptTextBox.Text = "";
    Decryptbutton.Enabled = false;
}

```

private void newparameters_Click(object sender, EventArgs e)

```

{
    RSACiphertextbox.Text = "";
    RSAPlaintextbox.Text = "";
    RSADecrypttextbox.Text = "";
    linkLabell.LinkVisited = false;

```

```

    RSACryptoServiceProvider rsa = new
    RSACryptoServiceProvider();

```

```

    //using rsa.ToXmlString(true)
    //provide public and private RSA param
    StreamWriter writer = new

```

```

    StreamWriter("TextEncryption\\PublicPrivateKey.xml");
    string publicPrivateKeyXML = rsa.ToXmlString(true);
    writer.Write(publicPrivateKeyXML);
    writer.Close();

```

```

        //provide public only RSA params
        writer = new
StreamWriter("TextEncryption\\PublicOnlyKey.xml");
        string publicOnlyKeyXML = rsa.ToXmlString(false);
        writer.Write(publicOnlyKeyXML);
        writer.Close();

        pvaluebox.Text = publicPrivateKeyXML;
    }

    private void RSAEncryptbutton_Click(object sender, EventArgs e)
    {
        RSADecrypttextbox.Text = "";
        //using xml
        RSACryptoServiceProvider rsa = new
RSACryptoServiceProvider();

        //public only RSA parameters for encrypt
        StreamReader reader = new
StreamReader("TextEncryption\\PublicOnlyKey.xml");
        string publicOnlyKeyXML = reader.ReadToEnd();
        rsa.FromXmlString(publicOnlyKeyXML);
        reader.Close();

        //read plaintext, encrypt it to ciphertext
        byte[] plainbytes =
Encoding.UTF8.GetBytes(RSAPlaintextbox.Text);
        rsacipherbyte = rsa.Encrypt(plainbytes, false); //FOAEP
needs high encryption pack

        RSACiphertextbox.Text =
Convert.ToBase64String(rsacipherbyte);

        RSADecryptButton.Enabled = true;
        button4.Enabled = true;
    }

    private void RSADecryptButton_Click(object sender, EventArgs e)

```

```

{
    RSADecrypttextbox.Text = "";
    //using xml
    RSACryptoServiceProvider rsa = new
    RSACryptoServiceProvider();

    //public and private RSA parameters for encrypt
    StreamReader reader = new
    StreamReader("TextEncryption\\PublicPrivateKey.xml");
    string publicPrivateKeyXML = reader.ReadToEnd();
    rsa.FromXmlString(publicPrivateKeyXML);
    reader.Close();

    //read ciphertext, decrypt it to plaintext
    byte[] plaintext = rsa.Decrypt(rsacipherbyte, false);
    //FOAEP needs high encryption pack

    char[] a = Encoding.UTF8.GetChars(plaintext);
    for (int q = 0; q < a.Length; q++)
        RSADecrypttextbox.Text += Convert.ToString(a[q]);
}

private void linkLabel1_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
{
    linkLabel1.LinkVisited = true;

    System.Diagnostics.Process.Start("TextEncryption\\PublicPrivateKey.xml"
);
}

private void RSASignButton_Click(object sender, EventArgs e)
{
    byte[] messagebytes =
    Encoding.UTF8.GetBytes(RSAMessageBox.Text);

    //create digest of original message using SHA1

```

```

    SHA1 sha1 = new SHA1CryptoServiceProvider();
    byte[] hashbytes = sha1.ComputeHash(messagebytes);

    RSACryptoServiceProvider rsa = new
    RSACryptoServiceProvider();
    rsasignaturebytes = rsa.SignHash(hashbytes,
    "1.3.14.3.2.26");
    rsaparams = rsa.ExportParameters(false);

    //save the RSA details in an XML File
    StreamWriter w = new
    StreamWriter("TextEncryption\\RSADetails.xml");
    string val = rsa.ToXmlString(true);
    w.Write(val);
    w.Close();

    //display the obtained codes in text form
    char[] a = Encoding.UTF8.GetChars(hashbytes);
    for (int q = 0; q < a.Length; q++)
        RSAHashMessageDigest.Text += Convert.ToString(a[q]);

    char[] b = Encoding.UTF8.GetChars(rsasignaturebytes);
    for (int q = 0; q < b.Length; q++)
        RSAEncryptedMessageDigest.Text +=
    Convert.ToString(b[q]);

    RSAVerifyButton.Enabled = true;
}

private void RSAVerifyButton_Click(object sender, EventArgs e)
{
    byte[] messagebytes =
    Encoding.UTF8.GetBytes(RSAMessageBox.Text);

    //create digest of original message using SHA1
    SHA1 sha1 = new SHA1CryptoServiceProvider();
    byte[] hashbytes = sha1.ComputeHash(messagebytes);

```

```

//create RSA object using parameters from signing
PSACryptoServiceProvider rsa = new
RSACryptoServiceProvider();
rsa.ImportParameters(rsaparams);

//do verification on hash using OID for SHA-1
bool match = rsa.VerifyHash(hashbytes, "1.3.14.3.2.26",
rsasignaturebytes);

if (match == true)
    MessageBox.Show("RSA Digital Signature Verified");
else
    MessageBox.Show("RSA Signature Not Verified");
}

private void linkLabel2_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
{
    linkLabel2.LinkVisited = true;

    System.Diagnostics.Process.Start("TextEncryption\\RSADetails.xml");
}

private void DSASignButton_Click(object sender, EventArgs e)
{
    byte[] messagebytes =
Encoding.UTF8.GetBytes(DSAMessageBox.Text);

    //create digest of original message using SHA1
    SHA1 sha1 = new SHA1CryptoServiceProvider();
    byte[] hashbytes = sha1.ComputeHash(messagebytes);

    DSACryptoServiceProvider dsa = new
DSACryptoServiceProvider();
    dsasignaturebytes = dsa.SignHash(hashbytes,
"1.3.14.3.2.26");
    dsaparams = dsa.ExportParameters(false);

```

```

        //save the RSA details in an XML File
        StreamWriter w = new
StreamWriter("TextEncryption\\DSADetails.xml");
        string val = dsa.ToXmlString(true);
        w.Write(val);
        w.Close();

        //display the obtained codes in text form
        char[] a = Encoding.UTF8.GetChars(hashbytes);
        for (int q = 0; q < a.Length; q++)
            DSAHashMessageDigest.Text += Convert.ToString(a[q]);

        char[] b = Encoding.UTF8.GetChars(dsasignaturebytes);
        for (int q = 0; q < b.Length; q++)
            DSAEncryptedMessageDigest.Text +=
Convert.ToString(b[q]);

        DSAVerifyButton.Enabled = true;
    }

    private void DSAVerifyButton_Click(object sender, EventArgs e)
    {
        byte[] messagebytes =
Encoding.UTF8.GetBytes(DSAMessageBox.Text);

        //create digest of original message using SHA1
        SHA1 sha1 = new SHA1CryptoServiceProvider();
        byte[] hashbytes = sha1.ComputeHash(messagebytes);

        //create RSA object using parameters from signing
        DSACryptoServiceProvider dsa = new
DSACryptoServiceProvider();
        dsa.ImportParameters(dsaparams);

        //do verification on hash using OID for SHA-1
        bool match = dsa.VerifyHash(hashbytes, "1.3.14.3.2.26",
dsasignaturebytes);
    }

```

```

        if (match == true)
            MessageBox.Show("DSA Digital Signature Verified");
        else
            MessageBox.Show("DSA Signature Not Verified");
    }

    private void linkLabel3_LinkClicked(object sender,
    LinkLabelLinkClickedEventArgs e)
    {
        linkLabel3.LinkVisited = true;

        System.Diagnostics.Process.Start("TextEncryption\\DSADetails.xml");
    }

    private void TextEncrypt_FormClosed(object sender,
    FormClosedEventArgs e)
    {
        System.Diagnostics.Process.Start("TestApp.exe");
    }

    private void button1_Click(object sender, EventArgs e)
    {
        OpenFileDialog openfile = new OpenFileDialog();
        openfile.RestoreDirectory = true;
        DialogResult result = openfile.ShowDialog();

        if ( result == DialogResult.Cancel )
            return;

        string filename = openfile.FileName;
        FileStream a = new FileStream(filename,
        FileMode.OpenOrCreate, FileAccess.Read);

        int len = (int)a.Length;

        byte[] r = new byte[len];

```

```

        a.Read(r, 0, len);
        char[] t = Encoding.UTF8.GetChars(r);
        PlainTextBox.Text = "";
        for (int q = 0; q < t.Length; q++)
            PlainTextBox.Text += Convert.ToString(t[q]);
        a.Close();
    }

```

```

private void button2_Click(object sender, EventArgs e)
{

```

```

    SaveFileDialog savefile = new SaveFileDialog();
    savefile.RestoreDirectory = true;
    DialogResult result = savefile.ShowDialog();

```

```

    if (result == DialogResult.Cancel)
        return;

```

```

    string filename = savefile.FileName;

```

```

    FileStream a = new FileStream(filename, FileMode.Truncate,
    FileAccess.Write);

```

```

    StreamWriter op = new StreamWriter(a);
    op.WriteLine(CipherTextBox1.Text);
    op.Close();
    a.Close();
}

```

```

private void button3_Click(object sender, EventArgs e)
{

```

```

    OpenFileDialog openfile = new OpenFileDialog();
    openfile.RestoreDirectory = true;
    DialogResult result = openfile.ShowDialog();

```

```

    if (result == DialogResult.Cancel)
        return;

```

```

    string filename = openfile.FileName;

```

```
FileStream a = new FileStream(filename,
    FileMode.OpenOrCreate, FileAccess.Read);
```

```
int len = (int)a.Length;
```

```
byte[] r = new byte[len];
```

```
a.Read(r, 0, len);
```

```
char[] t = Encoding.UTF8.GetChars(r);
```

```
RSAPlainTextbox.Text = "";
```

```
for (int q = 0; q < t.Length; q++)
```

```
    RSAPlainTextbox.Text += Convert.ToString(t[q]);
```

```
a.Close();
```

```
}
```

```
private void button4_Click(object sender, EventArgs e)
```

```
{
```

```
    SaveFileDialog savefile = new SaveFileDialog();
```

```
    savefile.RestoreDirectory = true;
```

```
    DialogResult result = savefile.ShowDialog();
```

```
    if (result == DialogResult.Cancel)
```

```
        return;
```

```
    string filename = savefile.FileName;
```

```
    FileStream a = new FileStream(filename, FileMode.Truncate,
    FileAccess.Write);
```

```
    StreamWriter op = new StreamWriter(a);
```

```
    op.WriteLine(RSACiphertextbox.Text);
```

```
    op.Close();
```

```
    a.Close();
```

```
}
```

```
}
```

```
}
```

6.4 DES Crypta -Analysis Coding

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;
using System.IO;

namespace DESAnalysis
{
    public partial class DESAnalysis : Form
    {
        byte[] IV;
        byte[] key;
        byte[] cipherbytes;
        const int MAXVAL = 50000;

        public DESAnalysis()
        {
            InitializeComponent();
        }

        private void Encryptbutton_Click(object sender, EventArgs e)
        {
            string t = Convert.ToString(KeyTextBox.Text);
            if (t.Length == 8)
            {
                byte[] key = Encoding.UTF8.GetBytes(KeyTextBox.Text);

                CipherTextBox.Text = "";
            }
        }
    }
}
```

```

CipherMode mode = CipherMode.CFB;
PaddingMode padding = PaddingMode.Zeros;

SymmetricAlgorithm sa = DES.Create();

sa.GenerateIV();
IV = sa.IV;
sa.Key = key;
sa.Mode = mode;
sa.Padding = padding;

MemoryStream ms = new MemoryStream();
CryptoStream cs = new CryptoStream(ms,
sa.CreateEncryptor(), CryptoStreamMode.Write);

byte[] plaintext =
Encoding.UTF8.GetBytes(PlainTextBox.Text);
cs.Write(plaintext, 0, plaintext.Length);
cs.Close();

cipherbytes = ms.ToArray();
ms.Close();

CipherTextBox.Text =
Convert.ToBase64String(cipherbytes);
SuppliedCiphertextBox.Text = CipherTextBox.Text;
}
else
    MessageBox.Show("Please Enter \"64\" Bit Key (8
Chars)");
}

private void CryptaButton_Click(object sender, EventArgs e)
{
    int q = 0;
    progressBar1.Value = 0;
    progressBar1.Maximum = MAXVAL;
    progressBar1.Minimum = 0;

```

```

for (q = 0; q <= MAXVAL; q++)
{
    progressBar1.Increment(1);
    string temp = Convert.ToString(q);
    int len = temp.Length;

    string t1 = "";
    for (int w = 0; w < (8 - len); w++)
        t1 += "0";

    t1 += temp;
    key = Encoding.UTF8.GetBytes(t1);
    int res = decrypt(t1);

    if (res == 1)
        break;
}

private int decrypt(string value)
{
    SymmetricAlgorithm sa = DES.Create();
    CipherMode mode = CipherMode.CFB;
    PaddingMode padding = PaddingMode.Zeros;

    sa.Key = key;
    sa.IV = IV;
    sa.Mode = mode;
    sa.Padding = padding;

    MemoryStream ms = new MemoryStream(cipherbytes);
    CryptoStream cs = new CryptoStream(ms,
sa.CreateDecryptor(), CryptoStreamMode.Read);

    byte[] plainbyte = new Byte[cipherbytes.Length];
    int ptl = cs.Read(plainbyte, 0, cipherbytes.Length);
    byte[] plntxt = new byte[ptl];

```

```

        Array.Copy(plainbyte, 0, plntxt, 0, ptl);

        string recoveredtext = "";
        char[] original = PlainTextBox.Text.ToCharArray();

        char[] a = Encoding.ASCII.GetChars(plntxt, 0,
        PlainTextBox.Text.Length);
        for (int q = 0; q < a.Length; q++)
            recoveredtext += Convert.ToString(a[q]);

        int l1 = recoveredtext.Length;
        int l2 = original.Length;

        int counter = 0;
        for (int e = 0; e < l2; e++)
        {
            if (recoveredtext[e] != original[e])
                counter = 1;
        }
        if (counter == 0)
        {
            RecoveredPlaintextBox.Text = recoveredtext;
            ObtainedKeyTextBox.Text = value;
            return 1;
        }
        else
            return 0;
    }

    private void DESAnalysis_FormClosed(object sender,
    FormClosedEventArgs e)
    {
        System.Diagnostics.Process.Start("TestApp.exe");
    }
}

```

6.5 File Encryption Coding

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.Security.Cryptography;

namespace FileEncryption
{
    public partial class Form1 : Form
    {
        string filename1;
        string filename2;
        string filename3;
        string filename4;

        byte[] cipherbytes;
        byte[] key;
        byte[] r;
        byte[] r1;

        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            OpenFileDialog od = new OpenFileDialog();
```

```

        od.RestoreDirectory = true;
        od.ShowDialog();
        filename1 = od.FileName;
        origianlfilebox.Text = filename1;

        //entering the .enc extension to the file to be saved
        filename2 += filename1 + ".enc";

        encryptionfilebox.Text = filename2;
    }

    SymmetricAlgorithm CreateSymmetricAlgorithm()
    {
        if (DESButton.Checked == true)
            return DES.Create();
        if (RijndaelButton.Checked == true)
            return Rijndael.Create();
        if (RC2Button.Checked == true)
            return RC2.Create();
        if (TDESButton.Checked == true)
            return TripleDES.Create();
        return null;
    }

    void checkkey1()
    {
        if (DESButton.Checked == true)
        {
            string t = Convert.ToString(KeyTextBox.Text);
            if (t.Length == 8)
                key = Encoding.UTF8.GetBytes(KeyTextBox.Text);
            else
                MessageBox.Show("Please Enter \"64\" Bit Key (8
Chars)");
        }

        if (TDESButton.Checked == true)
        {

```

```

        string t = Convert.ToString(KeyTextBox.Text);
        if (t.Length == 24)
            key = Encoding.UTF8.GetBytes(KeyTextBox.Text);
        else
            MessageBox.Show("Please Enter \"192\" Bit Key (24
Chars)");
    }

    if (RijndaelButton.Checked == true)
    {
        string t = Convert.ToString(KeyTextBox.Text);
        if (t.Length == 32)
            key = Encoding.UTF8.GetBytes(KeyTextBox.Text);
        else
            MessageBox.Show("Please Enter \"256\" Bit Key (32
Chars)");
    }

    if (RC2Button.Checked == true)
    {
        string t = Convert.ToString(KeyTextBox.Text);
        if (t.Length == 16)
            key = Encoding.UTF8.GetBytes(KeyTextBox.Text);
        else
            MessageBox.Show("Please Enter \"128\" Bit Key (16
Chars)");
    }
}

private void Encryptbutton_Click(object sender, EventArgs e)
{
    if (KeyTextBox.Text == "")
        MessageBox.Show("Please Enter Password For Encryption
Of File");
    else
    {
        checkkey1();
    }
}

```

```

        FileStream a1 = new FileStream(filename1,
        FileMode.Open, FileAccess.ReadWrite);
        int len = (int)a1.Length;
        r = new byte[len];
        a1.Read(r, 0, len);
        a1.Close();

        SymmetricAlgorithm sa = CreateSymmetricAlgorithm();
        sa.Key = key;
        sa.Mode = CipherMode.ECB;
        sa.Padding = PaddingMode.PKCS7;

        MemoryStream ms = new MemoryStream();
        CryptoStream cs = new CryptoStream(ms,
        sa.CreateEncryptor(), CryptoStreamMode.Write);

        cs.Write(r, 0, r.Length);
        cs.Close();
        cipherbytes = ms.ToArray();
        ms.Close();

        FileStream a2 = new FileStream(filename2,
        FileMode.OpenOrCreate, FileAccess.ReadWrite);
        a2.Write(cipherbytes, 0, cipherbytes.Length);
        a2.Close();

        MessageBox.Show("File Encrypted Successfully");
    }
}

void checkkey2()
{
    if (DESButton2.Checked == true)
    {
        string t = Convert.ToString(decryptpassbox.Text);
        if (t.Length == 8)
            key = Encoding.UTF8.GetBytes(decryptpassbox.Text);
        else
    }
}

```

```

        MessageBox.Show("Please Enter \"64\" Bit Key (8
Chars)");
    }

    if (TDESButton2.Checked == true)
    {
        string t = Convert.ToString(decryptpassbox.Text);
        if (t.Length == 24)
            key = Encoding.UTF8.GetBytes(decryptpassbox.Text);
        else
            MessageBox.Show("Please Enter \"192\" Bit Key (24
Chars)");
    }

    if (RijndaelButton2.Checked == true)
    {
        string t = Convert.ToString(decryptpassbox.Text);
        if (t.Length == 32)
            key = Encoding.UTF8.GetBytes(decryptpassbox.Text);
        else
            MessageBox.Show("Please Enter \"256\" Bit Key (32
Chars)");
    }

    if (RC2Button2.Checked == true)
    {
        string t = Convert.ToString(decryptpassbox.Text);
        if (t.Length == 16)
            key = Encoding.UTF8.GetBytes(decryptpassbox.Text);
        else
            MessageBox.Show("Please Enter \"128\" Bit Key (16
Chars)");
    }
}

private void button5_Click(object sender, EventArgs e)
{
    OpenFileDialog od = new OpenFileDialog();

```

```

        od.RestoreDirectory = true;
        od.ShowDialog();
        filename3 = od.FileName;
        decryptencryptedbox.Text = filename3;

        //removing the .enc extension to the file to be saved
        filename4 = "";
        int len = filename3.Length;

        for (int y = 0; y < (len - 4); y++)
            filename4 += filename3[y];

        decryptoriginalbox.Text = filename4;
    }

    private void button3_Click(object sender, EventArgs e)
    {
        if (decryptpassbox.Text == "")
            MessageBox.Show("Please Enter Password For Encryption  
Of File");
        else
        {
            checkkey2();

            FileStream a1 = new FileStream(filename3,
            FileMode.Open, FileAccess.ReadWrite);

            r1 = new byte[a1.Length];
            a1.Seek(0, SeekOrigin.Begin);
            a1.Read(r1, 0, r1.Length);
            a1.Close();

            try
            {
                SymmetricAlgorithm sa = CreateSymmetricAlgorithm();
                sa.Key = key;
                sa.Mode = CipherMode.ECB;
                sa.Padding = PaddingMode.PKCS7;
            }
        }
    }

```

```
MemoryStream ms = new MemoryStream(r1);  
CryptoStream cs = new CryptoStream(ms,  
sa.CreateDecryptor(), CryptoStreamMode.Read);
```

```
byte[] plainbyte = new Byte[r1.Length];  
cs.Read(plainbyte, 0, r1.Length);  
cs.Close();  
ms.Close();
```

```
FileStream a2 = new FileStream(filename4,  
FileMode.OpenOrCreate, FileAccess.ReadWrite);  
a2.Write(plainbyte, 0, plainbyte.Length);  
a2.Close();
```

```
MessageBox.Show("File Decrypted Successfully");
```

```
}  
catch (Exception)
```

```
{  
    MessageBox.Show("Decryption Failed !!");
```

```
}  
}
```

```
private void Form1_FormClosed(object sender,  
FormClosedEventArgs e)
```

```
{  
    System.Diagnostics.Process.Start("TestApp.exe");
```

```
}  
}
```

6.6 KDC Server Side Coding :

6.6.1 Login Page Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;
using System.IO;

namespace KDC_Server
{
    public partial class Form1 : Form
    {
        string servername;

        public Form1()
        {
            InitializeComponent();
        }

        private void loginbutton_Click(object sender, EventArgs e)
        {
            string username = loginidbox.Text;
            byte[] mesg = Encoding.UTF8.GetBytes(passwordbox.Text);
            SHA1 sha1 = new SHA1CryptoServiceProvider();
            byte[] hash = sha1.ComputeHash(mesg);

            KDCDataSetTableAdapters.LoginAdminTableAdapter a = new
            KDC_Server.KDCDataSetTableAdapters.LoginAdminTableAdapter();
```

```

int counter = (int)a.VerifyLogin(loginidbox.Text, hash);

if (counter == 1)
{
    Mainpage a1 = new Mainpage(this, username);
    a1.Show();
    this.Hide();
}
else
    MessageBox.Show("Wrong Username/Password");
}

private void linkLabel1_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
{
    StandAloneServer a = new StandAloneServer();
    a.Show();
    this.Hide();
}

private void Form1_FormClosing(object sender,
FormClosingEventArgs e)
{
    Application.Exit();
    System.Environment.Exit(System.Environment.ExitCode);
}

private void Form1_Load(object sender, EventArgs e)
{
    FileStream a = new FileStream("settings.txt",
    FileMode.OpenOrCreate, FileAccess.Read);
    int len = (int)a.Length;

    byte[] r = new byte[len];
    a.Read(r, 0, len);
    char[] t = Encoding.UTF8.GetChars(r);

    for (int q = 0; q < t.Length; q++)

```

```
servername += Convert.ToString(t[q]);  
a.Close();
```

```
KDC_Server.Properties.Settings.Default["KDCConnectionString"] = "Data  
Source=" + servername + ";Initial Catalog=KDC;User  
ID=bharat123;Password=asd";  
}  
}  
}
```

6.6.2 Main Page Code :

```
using System;  
using System.Collections.Generic;  
using System.ComponentModel;  
using System.Data;  
using System.Drawing;  
using System.Text;  
using System.Windows.Forms;  
  
namespace KDC_Server  
{  
    public partial class Mainpage : Form  
    {
```

```

Form1 login;
string loginname;

public Mainpage(Form1 a, string name)
{
    InitializeComponent();
    login = a;
    toolStripMenuItem2.Text = name;
    loginname = name;
}

private void Mainpage_FormClosing(object sender,
FormClosingEventArgs e)
{
    login.Show();
}

private void exitToolStripMenuItem_Click(object sender,
EventArgs e)
{
    this.Close();
    Application.Exit();
    System.Environment.Exit(System.Environment.ExitCode);
}

private void newUserToolStripMenuItem1_Click(object sender,
EventArgs e)
{
    NewUser a = new NewUser();
    a.MdiParent = this;
    a.Show();
}

private void newUserToolStripMenuItem_Click(object sender,
EventArgs e)
{
    ChangePassword a = new ChangePassword();
    a.MdiParent = this;
}

```

```

        a.Show();
    }

    private void runServerToolStripMenuItem_Click(object sender,
EventArgs e)
    {
        KDCServer a = new KDCServer();
        a.MdiParent = this;
        a.WindowState = FormWindowState.Maximized;
        a.Show();
    }

    private void changeKeyPairsToolStripMenuItem_Click(object
sender, EventArgs e)
    {
        ChangeKeyPairs a = new ChangeKeyPairs(loginname);
        a.MdiParent = this;
        a.Show();
    }

    private void requestsToolStripMenuItem_Click(object sender,
EventArgs e)
    {
        UpdateKeys a = new UpdateKeys();
        a.MdiParent = this;
        a.Show();
    }
}
}

```

6.6.3 Change Key Pairs Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;
using System.IO;

namespace KDC_Server
{
    public partial class ChangeKeyPairs : Form
    {
        RSACryptoServiceProvider rsa;
        string loginname;

        public ChangeKeyPairs(string login)
        {
            InitializeComponent();
            loginname = login;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            rsa = new RSACryptoServiceProvider();

            string publicPrivateKeyXML = rsa.ToXmlString(true);

            //provide public only RSA params
            string publicOnlyKeyXML = rsa.ToXmlString(false);

            parameterdeatils.Text = publicPrivateKeyXML;
            button2.Enabled = true;
        }
    }
}
```

```

    }

    private void button2_Click(object sender, EventArgs e)
    {
        //writing the key pairs to xml file

        StreamWriter writer = new StreamWriter("KDC\\KDC
Server\\KDCServerPublicPrivateKey.xml");
        string publicPrivateKeyXML = rsa.ToXmlString(true);
        writer.Write(publicPrivateKeyXML);
        writer.Close();

        //provide public only RSA params
        writer = new StreamWriter("KDC\\KDC
Server\\KDCServerPublicOnlyKey.xml");
        string publicOnlyKeyXML = rsa.ToXmlString(false);
        writer.Write(publicOnlyKeyXML);
        writer.Close();

        //writin the data to the db (update table)
        KDCDataSetTableAdapters.UpdateRequestsTableAdapter a = new
KDC_Server.KDCDataSetTableAdapters.UpdateRequestsTableAdapter();
        byte[] data = Encoding.UTF8.GetBytes(publicOnlyKeyXML);
        byte[] data2 = Encoding.UTF8.GetBytes(publicPrivateKeyXML);
        a.InsertData(loginname, data, data2, DateTime.Today);

        MessageBox.Show("Update Registered");
    }
}
}

```

6.6.4 Change Password Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;

namespace KDC_Server
{
    public partial class ChangePassword : Form
    {
        public ChangePassword()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (loginnamebox.Text == "")
                MessageBox.Show("Please Enter Username");
            else
            {
                if (oldpassbox.Text == "")
                    MessageBox.Show("Please Enter Old Password");
            }
        }
    }
}
```

```

else
{
    if (passbox1.Text != passbox2.Text)
        MessageBox.Show("Please Confirm The Entered
Passwords");
    else
    {
        if (radioButton1.Checked == true) //admin
        {
            KDCDataSetTableAdapters.LoginAdminTableAdapter al = new
            KDC_Server.KDCDataSetTableAdapters.LoginAdminTableAdapter();
            byte[] pass =
            Encoding.UTF8.GetBytes(oldpassbox.Text);
            SHA1 sha1 = new
            SHA1CryptoServiceProvider();
            byte[] hash = sha1.ComputeHash(pass);
            int count =
            (int)al.VerifyLogin(loginnamebox.Text, hash);

            if (count == 1)
            {
                pass =
                Encoding.UTF8.GetBytes(passbox1.Text);
                hash = sha1.ComputeHash(pass);
                al.ChangePassword(hash,
                loginnamebox.Text);

                MessageBox.Show("Password Changed");
            }
            else
                MessageBox.Show("Wrong
Username/Password");
        }
        else if (radioButton2.Checked == true) //client
        {

```


6.6.6 New User Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;

namespace KDC_Server
{
    public partial class NewUser : Form
    {
        public NewUser()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (usernamebox.Text == "")
                MessageBox.Show("Please Enter Username");
            else
            {
                if (loginnamebox.Text == "")
                    MessageBox.Show("Please Enter Login ID");
                else
                {
                    if (passbox1.Text != passbox2.Text)
                        MessageBox.Show("Please Confirm The Password(s)
Entered");
                    else
                    {
                        if (radioButton1.Checked == true) //admin
                        {
```

```

KDCDataSetTableAdapters.LoginAdminTableAdapter a = new
KDC_Server.KDCDataSetTableAdapters.LoginAdminTableAdapter();
    byte[] pass =
Encoding.UTF8.GetBytes(passbox1.Text);
    SHA1 sha1 = new
SHA1CryptoServiceProvider();
    byte[] hash = sha1.ComputeHash(pass);

    int count =
(int)a.CheckLogin(loginnamebox.Text);

    if (count == 0)
    {
        a.InsertAdmin(loginnamebox.Text, hash,
usernamebox.Text);

        MessageBox.Show("User Created");
    }
    else
        MessageBox.Show("Login Name Taken,
Enter Again");
    }
    else if (radioButton2.Checked == true) //client
account
    {
KDCDataSetTableAdapters.LoginClientTableAdapter b = new
KDC_Server.KDCDataSetTableAdapters.LoginClientTableAdapter();
        byte[] pass =
Encoding.UTF8.GetBytes(passbox1.Text);
        SHA1 sha1 = new
SHA1CryptoServiceProvider();
        byte[] hash = sha1.ComputeHash(pass);

        int count =
(int)b.CheckLogin(loginnamebox.Text);

        if (count == 0)

```

```

        {
            b.InsertClient(loginnamebox.Text, hash,
usernamebox.Text);

            MessageBox.Show("User Created");
        }
        else
            MessageBox.Show("Login Name Taken,
Enter Again");
    }
}

}

}

private void linkLabel1_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
{
    if (radioButton1.Checked == true) //admin
    {
        KDCDataSetTableAdapters.LoginAdminTableAdapter a = new
KDC_Server.KDCDataSetTableAdapters.LoginAdminTableAdapter();

        int count = (int)a.CheckLogin(loginnamebox.Text);

        if (count == 0)
            MessageBox.Show("Login Name Available");
        else
            MessageBox.Show("Login Name Taken");
    }
    else if (radioButton2.Checked == true) //client account
    {
        KDCDataSetTableAdapters.LoginClientTableAdapter b = new
KDC_Server.KDCDataSetTableAdapters.LoginClientTableAdapter();

        int count = (int)b.CheckLogin(loginnamebox.Text);

        if (count == 0)
            MessageBox.Show("Login Name Available");
    }
}

```

```

        else
            MessageBox.Show("Login Name Taken");
    }
}
}

```

6.6.7 Update Key Code :

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace KDC_Server
{
    public partial class UpdateKeys : Form
    {
        public UpdateKeys()
        {
            InitializeComponent();
        }

        private void UpdateKeys_Load(object sender, EventArgs e)
        {
            // TODO: This line of code loads data into the
            'KDCDataSet.UpdateRequests' table. You can move, or remove it, as
            needed.

            this.updateRequestsTableAdapter.Fill(this.kDCDataSet.UpdateRequests);
        }
    }
}

```

```

    }

    private void button1_Click(object sender, EventArgs e)
    {
        int counter = 0;

        for (int q = 0; q < dataGridView1.Rows.Count; q++)
        {
            if (dataGridView1.Rows[q].Cells[0].Value != null)
            {
                string temp =
                    dataGridView1.Rows[q].Cells[0].Value.ToString();
                if (temp == "True")
                {
                    KDCDataSetTableAdapters.KeyDirectoryTableAdapter a = new
                    KDC_Server.KDCDataSetTableAdapters.KeyDirectoryTableAdapter();

                    string login =
                        dataGridView1.Rows[q].Cells[1].Value.ToString();
                    int count = (int)a.CheckLogin(login);

                    if (count == 1)
                    {
                        //change exsisting value

                        a.UpdateKeyPairs((byte[])dataGridView1.Rows[q].Cells[2].Value,
                            (byte[])dataGridView1.Rows[q].Cells[3].Value, (DateTime)dataGridView1.Rows[q].Cells[4].Value, login);
                    }
                    else if (count == 0)
                    {
                        //add new user if not present
                        a.InsertUser(login,
                            (byte[])dataGridView1.Rows[q].Cells[2].Value,
                            (byte[])dataGridView1.Rows[q].Cells[3].Value,
                            (DateTime)dataGridView1.Rows[q].Cells[4].Value);
                    }
                }
            }
        }
    }

```

```

        counter = 1;

KDCDataSetTableAdapters.UpdateRequestsTableAdapter b = new
KDC_Server.KDCDataSetTableAdapters.UpdateRequestsTableAdapter();
        b.DeleteValue(login);
    }
}

if (counter != 0)
    MessageBox.Show("Selected Value(s) Updated");

this.updateRequestsTableAdapter.Fill(this.kDCDataSet.UpdateRequests);
}

private void fillToolStripButton_Click(object sender, EventArgs
e)
{
    try
    {
        this.updateRequestsTableAdapter.Fill(this.kDCDataSet.UpdateRequests);
    }
    catch (System.Exception ex)
    {
        System.Windows.Forms.MessageBox.Show(ex.Message);
    }
}
}
}

```

6.7 KDC Server Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.IO;
using System.Net;
using System.Net.Sockets;

namespace KDC_Server
{
    public partial class KDCServer : Form
    {
        string ipadd;
        int counterv = 0;

        public KDCServer()
        {
            InitializeComponent();
            //ipadd = "172.16.8.237";

            IPHostEntry HosyEntry =
            Dns.GetHostEntry((Dns.GetHostName()));
            if (HosyEntry.AddressList.Length > 0)
            {
                foreach (IPAddress ip in HosyEntry.AddressList)
                {
                    ipadd = ip.ToString();
                }
            }
        }
    }
}
```

```

private Socket connection; // Socket for accepting a connection
private Thread readThread; // Thread for processing incoming
messages

private NetworkStream socketStream; // network data stream
private BinaryReader reader;
private BinaryWriter writer;
private TcpListener listener;

private void KDCServer_Load(object sender, EventArgs e)
{
    readThread = new Thread(new ThreadStart(RunServer));
    readThread.Start();
}

private void KDCServer_FormClosing(object sender,
FormClosingEventArgs e)
{
    //System.Environment.Exit(System.Environment.ExitCode);
    closeconnections();
}

private delegate void DisplayDelegate(string message);

private void DisplayMessage(string message)
{
    // if modifying displayTextBox is not thread safe
    if (displayTextBox.InvokeRequired)
    {
        // use inherited method Invoke to execute
        DisplayMessage
        // via a delegate
        Invoke(new DisplayDelegate(DisplayMessage), new
object[] { message });
    } // end if
    else // OK to modify displayTextBox in current thread
        displayTextBox.Text += message;
    } // end method DisplayMessage

```

```

private delegate void DisableInputDelegate(bool value);

private void DisableInput(bool value)
{
    // if modifying inputTextBox is not thread safe
    if (inputTextBox.InvokeRequired)
    {
        // use inherited method Invoke to execute DisableInput
        // via a delegate
        Invoke(new DisableInputDelegate(DisableInput), new
object[] { value });
    } // end if
    else // OK to modify inputTextBox in current thread
        inputTextBox.ReadOnly = value;
} // end method DisableInput

private void inputTextBox_KeyDown(object sender, KeyEventArgs
e)
{
    // send the text to the client
    try
    {
        if (e.KeyCode == Keys.Enter && inputTextBox.ReadOnly ==
false)
        {
            // if the user at the server signaled termination
            // sever the connection to the client
            if (inputTextBox.Text == "TERMINATE")
                connection.Close();
            inputTextBox.Clear(); // clear the user's input
        } // end if
    } // end try
    catch (SocketException)
    {
        displayTextBox.Text += "\nError writing object";
    } // end catch
} // end method inputTextBox_KeyDown

```

```

public void getkey(string theReply)
{
    KDCDataSetTableAdapters.KeyDirectoryTableAdapter a = new
KDC_Server.KDCDataSetTableAdapters.KeyDirectoryTableAdapter();
    byte[] publickey = (byte[])a.GetPublicKey(theReply);

    char[] temp = Encoding.UTF8.GetChars(publickey);
    string key = "";

    for (int q = 0; q < temp.Length; q++)
        key += temp[q];

    writer.Write(key);

    displayTextBox.Text += "\r\nSERVER>>> Public Key Of " +
theReply + " Send To Client";
}

```

```

public void RunServer()
{
    int counter = 1;

    // wait for a client connection and display the text
    // that the client sends
    try
    {
        // Step 1: create TcpListener
        IPAddress local = IPAddress.Parse(ipadd);
        listener = new TcpListener(local, 50000);

        // Step 2: TcpListener waits for connection request
        listener.Start();

        // Step 3: establish connection upon client request
        while (true)
        {
            DisplayMessage("Waiting for connection\r\n");

```

```

        // accept an incoming connection
        connection = listener.AcceptSocket();
        counterv = 1;

        // create NetworkStream object associated with
socket
        socketStream = new NetworkStream(connection);

        // create objects for transferring data across
stream
        writer = new BinaryWriter(socketStream);
        reader = new BinaryReader(socketStream);

        DisplayMessage("Connection " + counter + "
received.\r\n");

        // inform client that connection was successful
        //writer.Write("SERVER>>> Connection successful");

        DisableInput(false); // enable inputTextBox

        string theReply = "";

        // Step 4: read string data sent from client
        do
        {
            try
            {
                // read the string sent to the server
                theReply = reader.ReadString();
            }
            catch { }
        } while (theReply.Length == 0);

        KDCDataSetTableAdapters.KeyDirectoryTableAdapter a = new
KDC_Server.KDCDataSetTableAdapters.KeyDirectoryTableAdapter();
        int count = (int)a.CheckLogin(theReply);

        if (count == 1)

```

```

        {
            //get the user public key
            getkey(theReply);
        }
        else
        {
            if (theReply != "TERMINATE")
            {
                writer.Write("Invalid User Please
Enter Correct Username");

                displayTextBox.Text +=
"\r\nSERVER>>> Invalid User ..Enter Correct Username";
            }
        }
    } // end try
    catch (Exception)
    {
        // handle exception if error reading data
        break;
    } // end catch
} while (theReply != "TERMINATE" &&
connection.Connected);

DisplayMessage("\r\nUser terminated
connection\r\n");

// Step 5: close connection
writer.Close();
reader.Close();
socketStream.Close();
connection.Close();
listener.Stop();

DisableInput(true); // disable InputTextBox
counter++;
} // end while
} // end try

```

```

        catch (Exception error)
        {
            MessageBox.Show(error.ToString());
        } // end catch
    }
    // end method RunServer

private void closeconnections()
{
    if (counterv == 1)
    {
        writer.Close();
        reader.Close();
        socketStream.Close();
        connection.Close();
        listener.Stop();
    }
    else
    {
        listener.Stop();
    }
}
}

```

6.8 KDC Stand Alone Server Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.IO;
using System.Net;
using System.Net.Sockets;

namespace KDC_Server
{
    public partial class StandAloneServer : Form
    {
        string ipadd;

        public StandAloneServer()
        {
            InitializeComponent();
            //ipadd = "172.16.8.237";

            IPHostEntry HosyEntry =
            Dns.GetHostEntry((Dns.GetHostName()));
            if (HosyEntry.AddressList.Length > 0)
            {
                foreach (IPAddress ip in HosyEntry.AddressList)
                {
                    ipadd = ip.ToString();
                }
            }
        }
    }
}
```

```

    }
}

private Socket connection; // Socket for accepting a connection
private Thread readThread; // Thread for processing incoming
messages

private NetworkStream socketStream; // network data stream
private BinaryReader reader;
private BinaryWriter writer;

private void StandAloneServer_Load(object sender, EventArgs e)
{
    readThread = new Thread(new ThreadStart(RunServer));
    readThread.Start();
}

private void StandAloneServer_FormClosing(object sender,
FormClosingEventArgs e)
{
    System.Environment.Exit(System.Environment.ExitCode);
    Application.Exit();
}

private delegate void DisplayDelegate(string message);

private void DisplayMessage(string message)
{
    // if modifying displayTextBox is not thread safe
    if (displayTextBox.InvokeRequired)
    {
        // use inherited method Invoke to execute
        DisplayMessage
        // via a delegate
        Invoke(new DisplayDelegate(DisplayMessage), new
object[] { message });
    } // end if
    else // OK to modify displayTextBox in current thread
        displayTextBox.Text += message;
}

```

```

} // end of method InputTextBox_KeyPress

private delegate void DisableInputDelegate(bool value);

private void DisableInput(bool value)
{
    // If modifying InputTextBox is not thread safe
    if (inputTextBox.InvokeRequired)
    {
        // use inherited method Invoke to execute DisableInput
        // via a delegate
        Invoke(new DisableInputDelegate(DisableInput), new
object[] { value });
    } // end if
    else // it is safe to modify InputTextBox in current thread
        inputTextBox.ReadOnly = value;
} // end method DisableInput

private void inputTextBox_KeyDown(object sender, KeyEventArgs
e)
{
    // send the text to the client

    try
    {
        if (e.KeyCode == Keys.Enter && inputTextBox.ReadOnly ==
false)
        {
            // if the user at the server terminal has entered
            // sever the connection to the client
            if (inputTextBox.Text == "TERMINATE")
                connection.Close();
            inputTextBox.Clear(); // clear the user's input
        } // end if
    } // end try
    catch (IOException)
    {
        displayTextBox.Text += "AnError writing object";
    }
}

```

```

        } // end of while loop

    } // end of while loop (input-output)

    public void getkey(string theReply)
    {
        KDCDataSetTableAdapters.KeyInReplyDataSet a = new
        KDC_Server.KDCDataSetTableAdapters.KeyInReplyDataSet();
        byte[] publickey = (byte[])a.GetPublicKey(theReply);

        char[] temp = Encoding.UTF8.GetChars(publickey);
        string key = "";

        for (int q = 0; q < temp.Length; q++)
            key += temp[q];

        writer.Write(key);

        displayTextBox.Text += "\r\nSERVER>>> Public Key Of " +
        theReply + " Send To Client";
    }

    public void RunServer()
    {
        TcpListener listener;
        int counter = 1;

        // wait for a client connection
        // that the client exists
        try
        {
            // Step 1: create TcpListener
            IPAddress local = IPAddress.Parse(ipadd);
            listener = new TcpListener(local, 50000);

            // Step 2: TcpListener waits for connection request
            listener.Start();
        }
    }

```

```

// Step 3: establish connection, open a data stream
while (true)
{
    DisplayMessage("Waiting for connection\r\n");

    // create an listening socket
    connection = listener.AcceptSocket();

    // create NetworkStream object associated with
    socket

    socketStream = new NetworkStream(connection);

    // create objects for transforming data streams
    stream

    writer = new BinaryWriter(socketStream);
    reader = new BinaryReader(socketStream);

    DisplayMessage("Connection " + counter + "
received.\r\n");

    // Inform client that connection was successful!!
    //writer.WriteLine("OKNetwork Connection successful");

    DisableInput(false); // enable Input Box

    string theReply = "";

    // Step 4: read all the data sent from client
    do
    {
        try
        {
            // read the string sent to the server
            theReply = reader.ReadString();
        }
    }
}

KDCDataSetTableAdapters.KeyDirectoryTableAdapter a = new
KDC_Server.KDCDataSetTableAdapters.KeyDirectoryTableAdapter();

```

```

        int count = (int)a.CheckLogin(theReply);

        if (count == 1)
        {
            //get the user password
            getkey(theReply);
        }
        else
        {
            if (theReply != "TERMINATE")
            {
                writer.Write("Invalid User Please,
Enter Correct Username");

                displayTextBox.Text +=
"\r\nSERVER>>> Invalid User ..Enter Correct Username";
            }
        }

    } // end try
    catch (IOException)
    {
        // handle connection error, if any
        break;
    } // end catch
} while (theReply != "TERMINATE" &&
connection.Connected);

DisplayMessage("\r\nUser terminated
connection\r\n");

// Step 5: close connection
writer.Close();
reader.Close();
socketStream.Close();
connection.Close();

DisableInput(true); // disable input until user
counter++;

```

```

        } // end while
    } // end try
    catch (UnauthorizedAccessException error)
    {
        MessageBox.Show(error.ToString());
    } // end catch
}
// end Authentication
}
}

```

6.9 KDC Client Side Coding :

6.9.1 Login Page Code :

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;
using System.IO;

namespace KDC_Client
{
    public partial class Form1 : Form
    {
        string servername;

        public Form1()
    }
}

```

```

{
    InitializeComponent();
}

private void loginbutton_Click(object sender, EventArgs e)
{
    string username = loginidbox.Text;
    byte[] mesg = Encoding.UTF8.GetBytes(passwordbox.Text);
    SHA1 sha1 = new SHA1CryptoServiceProvider();
    byte[] hash = sha1.ComputeHash(mesg);

    KDCDataSetTableAdapters.KDCDataSetTableAdapter a = new
    KDC_Client.KDCDataSetTableAdapters.KDCDataSetTableAdapter();
    int counter = (int)a.VerifyLogin(loginidbox.Text, hash);

    if (counter == 1)
    {
        //loading the xml file for the request to server
        KDCDataSetTableAdapters.KDCDataSetTableAdapter b =
        new KDC_Client.KDCDataSetTableAdapters.KDCDataSetTableAdapter();

        int count = (int)b.CheckUser(username);

        if (count == 1)
        {
            byte[] data1 = (byte[])b.GetUserKey(username);
            byte[] data2 = (byte[])b.GetPrivateKey(username);

            char[] keytemp = Encoding.UTF8.GetChars(data1);
            string keyxml = "";
            for (int y = 0; y < keytemp.Length; y++)
                keyxml += keytemp[y].ToString(keytemp[y]);

            StreamWriter writer = new StreamWriter("KDC\\KDC
Client\\KDCPublicOnlyKey.xml");
            string publicOnlyKeyXML = keyxml;
            writer.Write(publicOnlyKeyXML);
        }
    }
}

```

```

writer.Close();

char[] keytempl = Encoding.UTF8.GetChars(data2);
string keyxml1 = "";
for (int y1 = 0; y1 < keytempl.Length; y1++)
    keyxml1 += "<key>" + Encoding.UTF8.GetString(keytempl[y1]);

StreamWriter writer1 = new StreamWriter("KDC\\KDC
Client\\KDCPublicPrivateKey.xml");
string publicOnlyKeyXML1 = keyxml1;
writer1.Write(publicOnlyKeyXML1);
writer1.Close();
}
else
{
    //creating existing xml file if it exists
    publiconlykeyfile = new
    StreamWriter("KDC\\KDC Client\\KDCPublicOnlyKey.xml", true);
    writer1.Write();
    publiconlykeyfile.Close();
    publicprivatekeyfile = new
    StreamWriter("KDC\\KDC Client\\KDCPublicPrivateKey.xml", true);
    writer1.Write();
    publicprivatekeyfile.Close();
}

//open new form
Mainpage a1 = new Mainpage(this, username);
a1.Show();
this.Hide();
}
else
    MessageBox.Show("Wrong Username/Password");

//setting username = "long";

Mainpage a = new Mainpage(this, username);
a.Show();

```

```

        this.Hide();
    }

    private void Form1_FormClosing(object sender,
    FormClosingEventArgs e)
    {
        Application.Exit();
        System.Environment.Exit(System.Environment.ExitCode);
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        FileStream a = new FileStream("settings.txt",
        FileMode.OpenOrCreate, FileAccess.Read);
        int len = (int)a.Length;

        byte[] r = new byte[len];
        a.Read(r, 0, len);
        char[] t = Encoding.UTF8.GetChars(r);

        for (int q = 0; q < t.Length; q++)
            servername += Char.ToString(t[q]);
        a.Close();

        KDC_Client.Properties.Settings.Default["KDCConnectionString"] = "Data
        Source=" + servername + ";Initial Catalog=KDC;User
        ID=bharat123;Password=asd";
    }
}

```

6.9.2 Main Page Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace KDC_Client
{
    public partial class Mainpage : Form
    {
        Form login;
        string loginname;

        public Mainpage(Form a, string name)
        {
            InitializeComponent();
            login = a;
            toolStripMenuItem2.Text = name;
            loginname = name;
        }

        private void Mainpage_FormClosing(object sender,
        FormClosingEventArgs e)
        {
            login.Show();
        }

        private void chaToolStripMenuItem_Click(object sender,
        EventArgs e)
        {
            ChatForm a = new ChatForm();
            a.MdiParent = this;
            a.Show();
        }
    }
}
```

```

    }

    private void changeKeyPairsToolStripMenuItem_Click(object
sender, EventArgs e)
    {
        ChangeKeyPairs a = new ChangeKeyPairs(loginname);
        a.MdiParent = this;
        a.Show();
    }

    private void exitToolStripMenuItem_Click(object sender,
EventArgs e)
    {
        this.Close();
        Application.Exit();
        System.Environment.Exit(System.Environment.ExitCode);
    }

    private void runClientToolStripMenuItem_Click(object sender,
EventArgs e)
    {
        Client a = new Client(this);
        a.MdiParent = this;
        a.Show();
    }

    private void clientServerToolStripMenuItem_Click(object sender,
EventArgs e)
    {
        ClientServer a = new ClientServer(this);
        a.MdiParent = this;
        a.Show();
    }

    private void clientConfigurationToolStripMenuItem_Click(object
sender, EventArgs e)
    {
        ClientConfiguration a = new ClientConfiguration();

```

```

        a.MdiParent = this;
        a.Show();
    }
}

```

6.9.3 Change Key Pairs Code :

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;
using System.IO;

namespace KDC_Client
{
    public partial class ChangeKeyPairs : Form
    {
        RSA rsa;
        string loginname;
        string path;

        public ChangeKeyPairs(string login)
        {
            InitializeComponent();
            loginname = login;
        }
    }
}

```

```

    }

    private void button1_Click(object sender, EventArgs e)
    {
        rsa = new RSACryptoServiceProvider();

        string publicPrivateKeyXML = rsa.ToXmlString(true);

        //provide public only RSA paramter
        string publicOnlyKeyXML = rsa.ToXmlString(false);

        parameterdetails.Text = publicPrivateKeyXML;
        button2.Enabled = true;
    }

    private void button2_Click(object sender, EventArgs e)
    {
        //writing the key pair to the file

        StreamWriter writer = new StreamWriter("KDC\\KDC
Client\\KDCPublicPrivateKey.xml");

        string publicPrivateKeyXML = rsa.ToXmlString(true);
        writer.Write(publicPrivateKeyXML);
        writer.Close();

        //provide public only RSA paramter
        writer = new StreamWriter("KDC\\KDC
Client\\KDCPublicOnlyKey.xml");

        string publicOnlyKeyXML = rsa.ToXmlString(false);
        writer.Write(publicOnlyKeyXML);
        writer.Close();

        //write the data to the db (update table)
        KDCDataSetTableAdapters.UpdateDataSetTableAdapters a = new
KDC_Client.KDCDataSetTableAdapters.UpdateDataSetTableAdapters();
        byte[] data = Encoding.UTF8.GetBytes(publicOnlyKeyXML);
        byte[] data2 = Encoding.UTF8.GetBytes(publicPrivateKeyXML);
        a.InsertData(loginname, data, data2, DateTime.Today);
    }

```

```

        MessageBox.Show("Update Registered");
    }
}

```

6.9.4 Change Password Code :

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Security.Cryptography;

namespace KDC_Client
{
    public partial class ChangePassword : Form
    {
        public ChangePassword()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (loginnamebox.Text == "")
                MessageBox.Show("Please Enter Username");
        }
    }
}

```

```

else
{
    if (oldpassbox.Text == "")
        MessageBox.Show("Please Enter Old Password");
    else
    {
        if (passbox1.Text != passbox2.Text)
            MessageBox.Show("Please Confirm The Entered Passwords");
        else
        {
            KDCDataSetTableAdapters.LoginDataSetTableAdapters
al = new KDC_Client.KDCDataSetTableAdapters.LoginDataSetTableAdapters();
            byte[] pass =
Encoding.UTF8.GetBytes(oldpassbox.Text);
            SHA1 sha1 = new SHA1CryptoServiceProvider();
            byte[] hash = sha1.ComputeHash(pass);
            int count =
(int)al.VerifyLogin(loginnamebox.Text, hash);

            if (count == 1)
            {
                pass =
Encoding.UTF8.GetBytes(passbox1.Text);
                hash = sha1.ComputeHash(pass);
                al.ChangePassword(hash, loginnamebox.Text);

                MessageBox.Show("Password Changed");
            }
            else
                MessageBox.Show("Wrong Username/Password");
        }
    }
}
}
}
}
}

```

6.9.5 Session Key Connect Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Security.Cryptography;

namespace KDC_Client
{
    public partial class KDC_Client : Form
    {
        private TcpClient client;
        private NetworkStream output; // stream for receiving data
        private BinaryWriter writer; // facilitates writing to the
stream
        private BinaryReader reader; // facilitates reading from the
stream
        private Thread readThread; // Thread for processing incoming
messages
        string ipadd;
        private string message = "";
        string path;

        string parent;
        byte[] key;
```

```

byte[] data1;
byte[] data2;
string datatosend;
System.Net.Sockets.Socket sa;

public ClientConnect(MainForm mainpage)
{
    parent = mainpage;
    InitializeComponent();
}

private void ClientConnect_Load(object sender, EventArgs e)
{
}

private void ClientConnect_FormClosing(object sender,
FormClosingEventArgs e)
{
    //writer.WriteLine("TERMINATE");
    closeconnections();
}

public void RunClient()
{
    // Insert any initialization code here, before the try
    try
    {
        //DisplayMessage("Attempting connection\r\n");

        // Step 1: create TcpClient and connect to server
        client = new TcpClient();
        client.Connect(ipadd, 50001);

        // Step 2: get NetworkStream associated with TcpClient
        output = client.GetStream();

        // Insert any initialization code here, before the try
    }
}

```

```

writer = new StreamWriter(output);
reader = new StreamReader(output);

// Step 2: connecting to the server
do
{
    // Step 3: processing data
    try
    {
        // read message from server
        message = reader.ReadString();
        if (message == "Connection successful")
            groupBox2.Enabled = true;
        else
            MessageBox.Show("No Server Found");
    } // end try
    catch (Exception)
    {
        // handle exception if error in reading server
        data

//System.Environment.Exit(System.Environment.ExitCode);
    } // end catch
} while (message != "TERMINATE");

// Step 4: disconnecting
//disconnecting()

//Application.Exit();
} // end try
catch (Exception error)
{
    // handle exception if error in establishing connection
    MessageBox.Show(error.ToString(), "Connection Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
    //System.Environment.Exit(System.Environment.ExitCode);
} // end catch

```

```

} // end of CloseConnections()

private void CloseConnections()
{
    writer.Close();
    reader.Close();
    output.Close();
    client.Close();
}

private void ConfigureClient_Click(object sender, EventArgs e)
{
    ipadd = textBox1.Text;
    readThread = new Thread(new ThreadStart(RunClient));
    readThread.Start();
    groupBox3.Enabled = true;
}

private void connecttokdc_Click(object sender, EventArgs e)
{
    KDCClient a = new KDCClient();
    a.MdiParent = parent;
    a.Show();
    groupBox1.Enabled = true;
}

SymmetricAlgorithm CreateSymmetricAlgorithm()
{
    if (DESButton.Checked == true)
        return DES.Create();
    if (RijndaelButton.Checked == true)
        return Rijndael.Create();
    if (RC2Button.Checked == true)
        return RC2.Create();
    if (TDESButton.Checked == true)
        return TripleDES.Create();
    return null;
}

```

```

private void GenerateIV_Click(object sender, EventArgs e)
{
    sa = CreateSymmetricAlgorithm();
    sa.Mode = CipherMode.ECB;
    sa.Padding = PaddingMode.PKCS7;

    sa.GenerateKey();

    key = sa.Key;

    //data to be send
    string keyval = Convert.ToBase64String(key, 0, key.Length);

    datatosend = "Datasendby: ";
    datatosend += parent.toolStripMenuItem2.Text;
    datatosend += " Algorithm: ";

    if (DESButton.Checked == true)
        datatosend += "DES";
    if (RijndaelButton.Checked == true)
        datatosend += "Rijndael";
    if (RC2Button.Checked == true)
        datatosend += "RC2";
    if (TDESButton.Checked == true)
        datatosend += "TripleDES";

    datatosend += " Key: ";
    for (int q1 = 0; q1 < keyval.Length; q1++)
        datatosend += Convert.ToString(keyval[q1]);

    data1 = Encoding.UTF8.GetBytes(datatosend);
    groupBox5.Enabled = true;
}

```

```

private void EncryptUsingPrivate_Click(object sender, EventArgs e)
{

```

```

//encrypt using private key
groupBox6.Enabled = true;
}

private void EncryptUsingPublic_Click(object sender, EventArgs e)
{
    //encrypt using client's public key
    //create an RSA object for rsa = new
    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

    //public only RSA parameters for encrypt of client
    StreamReader reader = new StreamReader("KDC\\KDC
Client\\SavedPublicOnlyKey.xml");
    string publicOnlyKeyXML = reader.ReadToEnd();
    rsa.FromXmlString(publicOnlyKeyXML);
    reader.Close();

    //read plaintext, encrypt it to ciphertext
    data2 = rsa.Encrypt(data1, false); //if false, no padding
    //encryption pack

    EncryptUsingPrivate.Enabled = true;
}

private void button3_Click(object sender, EventArgs e)
{
    //send encrypted data to client
    string senddata = "";
    for (int q = 0; q < data2.Length; q++)
    {
        char t = Convert.ToChar(data2[q]);
        senddata += Convert.ToString(t);
    }

    writer.Write(senddata);
    NetworkStream wr = client.GetStream();
    wr.Write(data2, 0, data2.Length);
}

```

```

//hash datatosend to sha1
SHA1 sha1 = new SHA1CryptoServiceProvider();
byte[] msg = Encoding.UTF8.GetBytes(datatosend);
byte[] hashbytes = sha1.ComputeHash(msg);

//RSA Cryptosystem provider rsa = new
RSACryptoServiceProvider();

//read public key
TextReader reader = new StreamReader("KDC\\KDC
Client\\KDCPublicPrivateKey.xml");
string publicPrivateKeyXML = reader.ReadToEnd();
rsa.FromXmlString(publicPrivateKeyXML);

byte[] rsasignaturebytes = rsa.SignHash(hashbytes,
"1.3.14.3.2.26");

//signed hash value export
wr.Write(rsasignaturebytes, 0, rsasignaturebytes.Length);

//exchange messages
int algoval = 0;

if (DESButton.Checked == true)
    algoval = 1;
if (RijndaelButton.Checked == true)
    algoval = 2;
if (RC2Button.Checked == true)
    algoval = 3;
if (TDESButton.Checked == true)
    algoval = 4;

SessionKeyServer sks = new SessionKeyServer(algoval,
sa.Key);

sks.Show();
sks.MdiParent = parent;
this.Close();
}
}

```

}

6.9.6 Session Key Server Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Security.Cryptography;

namespace KDC_Client
{
    public partial class Client : Component
    {
        Message parent;
        string ipadd;
        public string ipadd2;
        int counterv = 0;

        public ClientServer(Message temp)
        {
            InitializeComponent();
            //ipadd = "192.16.8.227";

            parent = temp;

            {
                HosiEntry =
                Dns.GetHostEntry(Dns.GetHostName());
            }
        }
    }
}
```

```

        if (HosyEntry.AddressList.Length > 0)
        {
            foreach (IPAddress ip in HosyEntry.AddressList)
            {
                ipadd = ip.ToString();
            }
        }
    }

    private TcpListener listener;
    private Socket connection; // Socket for accepting a connection
    private Thread readThread; // Thread for processing incoming
messages
    private NetworkStream socketStream; // NetworkStream
    private StreamReader reader;
    private StreamWriter writer;
    string data="";
    byte[] databyte = new byte[128];
    byte[] datahash = new byte[128];
    private NetworkStream rd;

    private void ClientServer_Load(object sender, EventArgs e)
    {
        readThread = new Thread(new ThreadStart(RunServer));
        readThread.Start();
    }

    private void ClientServer_FormClosing(object sender,
        FormClosingEventArgs e)
    {
        // Disconnect client;
    }

    private delegate void DisplayMessage(string message);

    private void DisplayMessage(string message)
    {
        // Modifying displayText to be non-static
    }

```

```

        if (displayTextBox.InvokeRequired)
        {
            // use Invoke to get and display the message
            // OK to modify displayTextBox in current thread
            Invoke(new Action(() => { displayTextBox.Text += message; })
                , new object[] { message });
        } // end if
    } else // OK to modify displayTextBox in current thread
    {
        displayTextBox.Text += message;
    } // end method DisplayMessage
}

```

```

public void RunServer()
{
    int counter = 1;
    int count = 0;

    // wait for a client connection and display the log
    // that the server is running
    try
    {
        // Step 1: create TcpListener
        IPAddress local = IPAddress.Parse(ipaddr);
        listener = new TcpListener(local, 50001);

        // Step 2: TcpListener waits for connection request
        listener.Start();

        // Step 3: establish connection upon client request
        while (true)
        {
            DisplayMessage("Waiting for connection\r\n");

            // accept an incoming connection
            connection = listener.AcceptSocket();
            counter = 1;
            count = 0;
        }
    }
}

```

```

// create SocketStream object, associated with
socketStream = new SocketStream(connection);

// create delegate for Write method of Stream
writer = new StreamWriter(socketStream);
reader = new StreamReader(socketStream);
rd = socketStream;

DisplayMessage("Connection " + counter + "
received.\r\n");

// inform client that connection was successful
writer.Write("Connection successful");

string theReply = "";

// Step 4: read string data sent from client
do
{
    try
    {
        if (count == 0)
        {
            // read the string sent to the server
            theReply = reader.ReadString();

            displayTextBox.Text += theReply;
            data = theReply;
            rd.Read(databyte, 0, 128);
            count++;
        }
        else if (count == 1)
        {
            rd.Read(datahash, 0, 128);
            count++;
        }
    }
}

```

```

        } // end try
        catch (IOException)
        {
            // handle exception by terminating connection
            break;
        } // end catch
    } while (theReply != "TERMINATE" &&
connection.Connected);

```

```

        DisplayMessage("\r\nUser terminated
connection\r\n");

```

```

        // Step 5: close connection

```

```

        writer.Close();
        reader.Close();
        socketStream.Close();
        connection.Close();
        listener.Stop();

```

```

        counter++;

```

```

    } // end while

```

```

} // end try

```

```

catch (Exception error)

```

```

{

```

```

    MessageBox.Show(error.ToString());

```

```

} // end catch

```

```

}

```

```

// end method RunServer

```

```

private void closeconnections()

```

```

{

```

```

    if (counterv == 1)

```

```

    {

```

```

        writer.Close();

```

```

        reader.Close();

```

```

        socketStream.Close();

```

```

        connection.Close();

```

```

        listener.Stop();
    }
}

```

```

    }
    else
    {
        listener.Stop();
    }
}

private void ExtractDataButton_Click(object sender, EventArgs e)
{
    textBox1.Text = data;
}

private void DecryptButton_Click(object sender, EventArgs e)
{
    RSACryptServiceProvider rsa = new
    RSACryptServiceProvider();

    //public and private RSA parameters for encryption
    XmlReader reader = new XmlReader("KDC\\KDC
Client\\KDCPublicPrivateKey.xml");
    string publicPrivateKeyXML = reader.ReadToEnd();
    rsa.FromXmlString(publicPrivateKeyXML);
    reader.Close();

    //here: substitution, decrypt it to plaintext
    byte[] plaintext = rsa.Decrypt(databyte, false); //data
needs sign encryption pack

    char[] a = Encoding.UTF8.GetChars(plaintext);
    for (int q = 0; q < a.Length; q++)
        textBox2.Text += Convert.ToString(a[q]);

    //verification

    //get key from the XML for the user who sent the data
    string user = "";
    int counter = 12;

```

```

string temp = textBox2.Text;

while (temp[counter] != ' ')
{
    char x = temp[counter];
    user += x;
    counter++;
}

KDCDataSetTableAdapters.KeyDataSetTableAdapters b = new
KDC_Client.KDCDataSetTableAdapters.KeyDataSetTableAdapters();
byte[] key = (byte[])b.GetUserKey(user);

char[] keytemp = Encoding.UTF8.GetChars(key);
string keyxml = "";
for (int y = 0; y < keytemp.Length; y++)
    keyxml += Convert.ToString(keytemp[y]);

StreamWriter writer = new StreamWriter("KDC\\KDC
Client\\KDCSignPublicOnlyKey.xml");
string publicOnlyKeyXML = keyxml;
writer.Write(publicOnlyKeyXML);
writer.Close();

SHA1 sha1 = new SHA1CryptoServiceProvider();
byte[] msgghsh = Encoding.UTF8.GetBytes(textBox2.Text);

byte[] hashbytes = sha1.ComputeHash(msgghsh);

//create RSA object using parameters from signing
CAACryptorLib.Provider rsal = new
ProviderLib.CAACryptorLib.Provider();

StreamReader reader1 = new StreamReader("KDC\\KDC
Client\\KDCSignPublicOnlyKey.xml");
string publicPrivateKeyXML1 = reader1.ReadToEnd();
rsal.FromXmlString(publicPrivateKeyXML1);

```

```

//do verification on hash using sha1
bool match = rsa1.VerifyHash(hashbytes, "1.3.14.3.2.26",
datahash);

if (match == true)
    MessageBox.Show("Digital Signature Verified");
else
    MessageBox.Show("Signature Not Verified");
}

private void ExchangemessageButton_Click(object sender,
EventArgs e)
{
    //extract key and iv from the text form a message
    string obtaineddata = textBox2.Text;
    string[] ary = obtaineddata.Split(' ');
    string algo = ary[3].ToString();
    string key = ary[5].ToString();

    int algoval = 0;
    if(algo=="DES")
        algoval = 1;
    else if(algo=="Rijndael")
        algoval = 2;
    else if(algo=="RC2")
        algoval = 3;
    else if(algo=="TripleDES")
        algoval = 4;

    byte[] keyval = Convert.FromBase64String(key);

    //get ip address of the sender
    Client ip = new Client(this, parent, algoval, keyval);
    ip.MdiParent = parent;
    ip.Show();
    this.Close();
}
}

```

}

6.9.7 Get IP Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace KDC_Client
{
    public partial class KDC_Client : Form
    {
        ClientServer parent;
        MainPage pl;
        int algoval;
        byte[] keyval;

        public GetIP(ClientServer a, MainPage b, int algo, byte[] keyv)
        {
            InitializeComponent();
            parent = a;
            pl = b;
            algoval = algo;
            keyval = keyv;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            parent.ipadd2 = textBox1.Text;

            //start the message exchange
        }
    }
}
```

```

        skc = new KDCClient(textBox1.Text,
        algoval, keyval);
        skc.Show();
        skc.MdiParent = pl;

        this.Close();
    }
}

```

6.9.8 KDC Client Code :

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Security.Cryptography;

namespace KDC_Client
{
    public partial class KDCClient : Form
    {
        string ipadd;

        public KDCClient()
        {
            InitializeComponent();
            //ipadd = "172.16.1.251";

```

```

        a = new StreamReader("KDC\\KDC\\
Client\\config.txt", Encoding.UTF8);
        int len = (int)a.Length;

        byte[] r = new byte[len];
        a.Read(r, 0, len);
        char[] t = Encoding.UTF8.GetChars(r);

        for (int q = 0; q < t.Length; q++)
            ipadd += t[q].ToString(t[q]);
        a.Close();
    }

    private TcpClient client;
    private NetworkStream output; // stream for receiving data
    private StreamWriter writer; // facilitates writing to the
    stream
    private BinaryReader reader; // facilitates reading from the
    stream
    private Thread readThread; // thread for processing received
    message
    private string message = "";

    private void KDCClient_Load(object sender, EventArgs e)
    {
        readThread = new Thread(new ThreadStart(RunClient));
        readThread.Start();
    }

    private void KDCClient_FormClosing(object sender,
    FormClosingEventArgs e)
    {
        writer.Write("TERMINATE");
        closeconnections();
        //System.Environment.Exit(System.Environment.ExitCode);
    }

    private delegate void RunClient(string message);

```

```

private void DisplayMessage(string message)
{
    // is modifying displayTextbox in current thread?
    if (displayTextBox.InvokeRequired)
    {
        // use inherited method Invoke to call
        DisplayMessage
        // via a delegate
        Invoke(new Action(DisplayMessage), new
object[] { message });
    } // end if
    else // OK to modify displayTextbox in current thread
        displayTextBox.Text += message;
    } // end method DisplayMessage

private delegate void DisableInputDelegate(bool value);

private void DisableInput(bool value)
{
    if (inputTextBox.InvokeRequired)
    {
        // use inherited method Invoke to call DisableInput
        // via a delegate
        Invoke(new Action(DisableInput), new
object[] { value });
    } // end if
    else // OK to modify inputTextBox in current thread
        inputTextBox.ReadOnly = value;
    } // end method DisableInput

private void inputTextBox_KeyDown(object sender, KeyEventArgs
e)
{
    try
    {
        if (e.KeyCode == Keys.Enter && inputTextBox.ReadOnly ==
false)

```



```

// loop until server closes connection
do
{
    // Step 3: uncompress message
    try
    {
        // read message from server
        message = reader.ReadString();
        DisplayMessage("\r\n" + message);
    } // end try
    catch (Exception)
    {
        // handle exception if error in reading message
        data
        //System.Environment.Exit(System.Environment.ExitCode);
    } // end catch
} while (message != "TERMINATE");

// Step 4: close connection
closeconnections();

//Application.Exit();
} // end try
catch (Exception error)
{
    // handle exception if error in establishing connection
    MessageBox.Show(error.ToString(), "Connection Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
    //System.Environment.Exit(System.Environment.ExitCode);
} // end catch

} // end method RunClient

private void closeconnections()
{
    writer.Close();
    reader.Close();

```

```

        output.Close();
        client.Close();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        RSACryptoServiceProvider rsa = new
        RSACryptoServiceProvider();

        StreamWriter writer = new StreamWriter("KDC\\KDC
        Client\\SavedPublicOnlyKey.xml");

        string publicOnlyKeyXML = displayTextBox.Text;
        writer.Write(publicOnlyKeyXML);
        writer.Close();
        MessageBox.Show("File Saved !!");
    }
}

```

6.9.9 Session Key Chat Server Code :

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;

```

```

using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Security.Cryptography;

namespace KDC_Client
{
    public partial class KDC_Client : Form
    {
        byte[] key;
        int nk;
        int algo;
        SymmetricAlgorithm sa;
        int counter = 0;
        string ipadd;
        int mins;
        int secs;

        public SessionKeyServer(int algoval, byte[] key1)
        {
            InitializeComponent();

            IPHostEntry HosyEntry =
                Dns.GetHostEntry((Dns.GetHostName()));
            if (HosyEntry.AddressList.Length > 0)
            {
                foreach (IPAddress ip in HosyEntry.AddressList)
                {
                    ipadd = ip.ToString();
                }
            }

            if (algoval == 1)
                sa = DES.Create();
            else if (algoval == 2)

```

```

        sa = SHA1.Create();
    else if (algoval == 3)
        sa = MD5.Create();
    else if (algoval == 4)
        sa = SHA256.Create();

    sa.Mode = CipherMode.ECB;
    sa.Padding = PaddingMode.PKCS7;
    sa.Key = key1;

    key = key1;
    algo = algoval;
}

private TcpListener listener;
private Socket connection; // Socket for accepting a connection
private Thread readThread; // Thread for processing incoming
// data
private Timer timer;
private NetworkStream socketStream; // Network stream to read
private BinaryReader reader; // Data stream coming from the
stream
private BinaryWriter writer;
byte[] databyte = new byte[128];
private NetworkStream rd;

private void SessionKeyServer_Load(object sender, EventArgs e)
{
    readThread = new Thread(new ThreadStart(RunServer));
    readThread.Start();
}

private void SessionKeyServer_FormClosing(object sender,
// EventArgs e)
{
    //Close listener here();
}

```

```

private delegate void DisplayMessage(string message);

private void DisplayMessage(string message)
{
    // If modifying displayTextBox is not thread safe
    if (EncryptedIncommingMessageBox.InvokeRequired)
    {
        // Use the Invoke method to call the
        DisplayMessage
        // via a delegate
        Invoke(new DisplayMessage(DisplayMessage), new
        object[] { message });
    } // end if
    else // OK to modify displayTextBox in current thread
        EncryptedIncommingMessageBox.Text += message;
    } // end method DisplayMessage

public void countdown()
{
    // Getting settings from the file
    StreamReader a = new StreamReader("KDC\\KDC
Client\\timeout.txt", Encoding.UTF8, true, 1024, Encoding.UTF8, true);
    int len = (int)a.Length;

    byte[] r = new byte[len];
    a.Read(r, 0, len);
    char[] t = Encoding.UTF8.GetChars(r);

    string tempval = "";
    for (int q = 0; q < t.Length; q++)
        tempval += Char.ToString(t[q]);
    a.Close();

    string[] s = tempval.Split(' ');
    mins = Convert.ToInt16(s[0]);
    secs = Convert.ToInt16(s[1]);
}

```

```

//Main timer counts down
progressBar1.Maximum = (mins * 60) + secs + 1;
progressBar1.Value = progressBar1.Maximum;
DateTime future = DateTime.Now;
future = future.AddMinutes(mins);
future = future.AddSeconds(secs);

int counter = 0;
int diff;
int prev = 0;
while (counter == 0)
{
    diff = future - DateTime.Now;

    label5.Text = diff.ToString(diff.Minutes + " : " +
diff.Seconds);

    if (prev != diff.Seconds)
        progressBar1.Increment(-1);

    if (diff.Minutes == 0 && diff.Seconds == 0)
        counter = 1;
    prev = diff.Seconds;
}
}

```

```

public void RunServer()
{
    int counter = 1;
    int count = 0;

    // wait for a client connection and display the text
    // that the client sends
    try
    {
        // Step 1: create TcpListener
        IPAddress local = IPAddress.Parse(ipaddr);
        listener = new TcpListener(local, 50002);
    }
}

```

```
// Start the listener thread.
listener.Start();
```

```
// Step 3: establish connection with client.
while (true)
```

```
{
```

```
    // accept an incoming connection.
```

```
    connection = listener.AcceptSocket();
```

```
    counterv = 1;
```

```
    count = 0;
```

```
    // Set up timer.
```

```
    timer = new Timer(new TimerCallback(countdown));
```

```
    timer.Start();
```

```
    // Create NetworkStream object associated with
```

```
socket
```

```
    socketStream = new NetworkStream(connection);
```

```
    reader = new BinaryReader(socketStream);
```

```
    writer = new BinaryWriter(socketStream);
```

```
    rd = socketStream;
```

```
    string theReply = "";
```

```
    nk = 0;
```

```
    // Step 4: read incoming data from client.
do
```

```
do
```

```
{
```

```
    try
```

```
{
```

```
        theReply = reader.ReadString();
```

```
        if (theReply == "New Key")
```

```
{
```

```
            nk = 1;
```

```
            count = 0;
```

```
            OutgoingMessageBox.ReadOnly = true;
```

```
        }
```

```

else if (theReply == "End Session")
{
    MessageBox.Show("Session Has Been Ended
!!");

    //End of main;
}
else
{
    if (nk == 0)
    {
        EncryptedIncommingMessageBox.Text
        EncryptedIncommingMessageBox.Text

        databyte =
Convert.FromBase64String(theReply);
        decryptincommingmessage(databyte);

        if (count == 0)
        {
            OutgoingMessageBox.ReadOnly =
false;

            count++;
        }
    }
    else if (nk == 1)
    {
        databyte =
Convert.FromBase64String(theReply);
        decryptincommingmessage(databyte);
        nk = 0;
        count = 0;
        timer.Abort();
        timer = new Thread(new
ThreadStart(countdown));

        timer.Start();
    }
}

```

```

        .Show("New Key Obtained
!!");

    }

}

} // end try
catch (IOException)
{
    // handle exception if error location is
    // break;
} // end catch

} while (theReply != "End Session" &&
connection.Connected);

// Step 5: close connection
// reader.Close();
writer.Close();
socketStream.Close();
connection.Close();
listener.Stop();
*/

counter++;
this.Close();

} // end while
} // end try
catch (IOException error)
{
    MessageBox.Show(error.ToString());
} // end catch
}

// end method RunServer

private void closeconnections()
{
    if (counterv == 1)
    {
        socketStream.Close();
        reader.Close();
    }
}

```

```

        writer.Close();
        connection.Close();
        listener.Stop();
    }
    else
    {
        listener.Stop();
    }
}

private void encryptoutgoingmessage(string message)
{
    byte[] r = Encoding.UTF8.GetBytes(message);
    MemoryStream ms = new MemoryStream();
    CryptoStream cs = new CryptoStream(ms,
sa.CreateEncryptor(), CryptoStreamMode.Write);

    cs.Write(r, 0, r.Length);
    cs.Close();
    byte[] cipherbytes = ms.ToArray();
    ms.Close();

    //send encrypted data
    string temp = Convert.ToBase64String(cipherbytes);
    writer.Write(temp);

    //display encrypted message
    EncryptedOutgoingMessageBox.Text = temp;
}

private void decryptincomingmessage(byte[] r1)
{
    MemoryStream ms = new MemoryStream(r1);
    CryptoStream cs = new CryptoStream(ms,
sa.CreateDecryptor(), CryptoStreamMode.Read);

    byte[] plainbyte = new byte[r1.Length];
    cs.Read(plainbyte, 0, r1.Length);
}

```

```

cs.Close();
ms.Close();

char[] plaintext = Encoding.UTF8.GetChars(plainbyte);
string val = "";
for (int w = 0; w < plaintext.Length; w++)
    val += plaintext[w].ToString(plaintext[w]);

if (nk == 0)
{
    DecryptedIncomingMessageBox.Text += val;
    DecryptedIncomingMessageBox.Text += "\r\n";
}

if (nk == 1)
{
    string t1 = "";

    if (algo == 1)
    {
        for (int q = 0; q < 8; q++)
            t1 += val[q].ToString();
    }
    else if (algo == 2)
    {
        for (int q = 0; q < 32; q++)
            t1 += val[q].ToString();
    }
    else if (algo == 3)
    {
        for (int q = 0; q < 16; q++)
            t1 += val[q].ToString();
    }
    else if (algo == 4)
    {
        for (int q = 0; q < 24; q++)
            t1 += val[q].ToString();
    }
}

```

```

        byte[] ty = new byte[t1.Length];
        for (int z = 0; z < t1.Length; z++)
            ty[z] = Convert.ToByte(t1[z]);

        key = ty;
        sa.Key = key;
    }
}

private void OutgoingMessageBox_KeyDown(object sender,
KeyEventArgs e)
{
    try
    {
        if (e.KeyCode == Keys.Enter &&
OutgoingMessageBox.ReadOnly == false)
        {
            encryptoutgoinmessage(OutgoingMessageBox.Text);
            OutgoingMessageBox.Clear(); // clear the input
        }
    }
    catch (Exception error)
    {
        MessageBox.Show(error.ToString());
    }
}
}

```

6.9.10 Session Key Chat Client Code :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Security.Cryptography;

namespace KDC_Client
{
    public partial class SessionKeyClient : Form
    {
        string ipadd;
        byte[] key;
        byte[] tempkey;
        SymmetricAlgorithm sa;
        byte[] databyte = new byte[128];

        int newkeyparam;
        bool conti;
        bool val;
        int mins;
        int secs;
        int resettimer = 0;

        private TcpClient client;
        private NetworkStream output; // stream for receiving data
        private StreamWriter writer; // stream for sending data
    }
}
```

```

private Stream reader; // Stream for reading data from
stream
private Thread readThread; // Thread for reading data from
stream

```

```

private Timer timer;
private string message = "";

```

```

public SessionKeyClient(string ip, int algoval, byte[] key1)
{

```

```

    InitializeComponent();

```

```

    ipadd = ip;

```

```

    if (algoval == 1)

```

```

        sa = SHA.Create();

```

```

    else if (algoval == 2)

```

```

        sa = RIJNDAEL.Create();

```

```

    else if (algoval == 3)

```

```

        sa = DES.Create();

```

```

    else if (algoval == 4)

```

```

        sa = TRIPLEDES.Create();

```

```

    sa.Mode = CipherMode.ECB;

```

```

    sa.Padding = PaddingMode.PKCS7;

```

```

    sa.Key = key1;

```

```

    key = key1;

```

```

}

```

```

private void SessionKeyClient_Load(object sender, EventArgs e)
{

```

```

    readThread = new Thread(new ThreadStart(RunClient));

```

```

    readThread.Start();

```

```

}

```

```

private void SessionKeyClient_FormClosing(object sender,
FormClosingEventArgs e)

```

```

{

```

```

        writer.Write("End Session");
        closeconnections();
    }

    public void countdown()
    {
        do
        {
            //getting settings from the file
            StreamReader a = new StreamReader("KDC\\KDC
Client\\timeout.txt", Encoding.UTF8, true, 1024);
            int len = (int)a.Length;

            byte[] r = new byte[len];
            a.Read(r, 0, len);
            char[] t = Encoding.UTF8.GetChars(r);

            string tempval = "";
            for (int q = 0; q < t.Length; q++)
                tempval += Convert.ToString(t[q]);
            a.Close();

            string[] s = tempval.Split(' ');
            mins = Convert.ToInt16(s[0]);
            secs = Convert.ToInt16(s[1]);

            //main timer in mins & secs
            progressBar1.Maximum = (mins * 60) + secs + 1;
            progressBar1.Value = progressBar1.Maximum;
            DateTime future = DateTime.Now;
            future = future.AddMinutes(mins);
            future = future.AddSeconds(secs);

            int counter = 0;
            TimeSpan diff;
            int prev = 0;
            while (counter == 0)
            {

```

```

diff = future - DateTime.Now;

label5.Text = "Time Left: " + diff.ToString(diff.Minutes + " : "
+ diff.Seconds);

if (prev != diff.Seconds)
    progressBar1.Increment(-1);

if (diff.Minutes == 0 && diff.Seconds == 0)
    counter = 1;
prev = diff.Seconds;
}

conti = false;

DialogBox: res = MessageBox.Show("Session Expired !!
Press OK to get new Session Key, Cancel to Exit", "Warning",
MessageBoxButtons.OKCancel);

if (res == DialogResult.OK)
{
    writer.Write("New Key");
    sa.GenerateKey();
    tempkey = sa.Key;

    sa.Key = key;

    val = false;
    newkeyparam = 1;
    encryptkeyandsend();

    sa.Key = tempkey;
    key = tempkey;
    conti = true;
    resettimer = 1;
}
else if (res == DialogResult.Cancel)
{
    resettimer = 0;
    message = "TERMINATE";
    MessageBox.Show("End Of Session !!");
}

```

```

        this.Close();
    }
} while (resettimer == 1);
}

private bool checktimeout()
{
    if (conti == true)
        return false;
    else
        return true;
}

public void RunClient()
{
    // instantiate TcpClient for sending data to server
    try
    {
        // Replace IP address and port number
        // Step 1: create TcpClient and connect to server
        client = new TcpClient();
        client.Connect(ipadd, 50002);

        // Step 2: get NetworkStream associated with TcpClient
        output = client.GetStream();

        // get the starting time of communication
        conti = true;
        timer = new Timer(new TimerCallback(countdown));
        timer.Start();

        // create objects for writing and reading information
        writer = new BinaryWriter(output);
        reader = new BinaryReader(output);

        newkeyparam = 0;
        // loop until server signals termination
    }
}

```

```

do
{
    // Step 3: connect to server
    try
    {
        val = checktimeout();

        if (val == false) // connection timeout
        {
            // read message from server
            message = reader.ReadString();
            EncryptedIncommingMessageBox.Text +=
message;

            EncryptedIncommingMessageBox.Text +=
"\r\n";

            databyte =
Convert.FromBase64String(message);
            decryptincommingmessage(databyte);
        }
    } // end try
    catch (Exception)
    {
        // handle exception if error in reading server
data

//System.Environment.Exit(System.Environment.ExitCode);
    } // end catch
    } while (message != "TERMINATE");

    // Step 4: close connection
    //reader.Dispose();
    //writer.Close();
    reader.Close();
    output.Close();
    client.Close(); //

this.Close();

```

```

        //Application.Exit();
    } // end try
    catch (Exception error)
    {
        // handle exception if error is not handled in
        // previous Show(error.ToString(), "Connection Error",
        // MessageBoxButtons.OK, MessageBoxIcon.Error);
        // Write.WriteLine(error.ToString(), "Error");
    } // end catch

```

```

} // end method RunClient

```

```

private void closeconnections()

```

```

{
    writer.Close();
    reader.Close();
    output.Close();
    client.Close();
}

```

```

private void encryptoutgoimessage(string message)

```

```

{
    byte[] r = Encoding.UTF8.GetBytes(message);
    MemoryStream ms = new MemoryStream();
    CryptoStream cs = new CryptoStream(ms,
sa.CreateEncryptor(), CryptoStreamMode.Write);

    cs.Write(r, 0, r.Length);
    cs.Close();
    byte[] cipherbytes = ms.ToArray();
    ms.Close();

    //send encrypted data
    string temp = Convert.ToBase64String(cipherbytes);
    writer.Write(temp);

    if (newkeyparam == 0)

```

```

    {
        //Display decrypted message
        EncryptedOutgoingMessageBox.Text = temp;
    }
    else
        newkeyparam = 0;
}

private void encryptkeyandsend()
{
    string temp = "";
    for (int w = 0; w < tempkey.Length; w++)
    {
        char r1 = Convert.ToChar(tempkey[w]);
        temp += Convert.ToString(r1);
    }

    encryptoutgoinmessage(temp);
}

private void decryptincommingmessage(byte[] r1)
{
    MemoryStream ms = new MemoryStream(r1);
    CryptoStream cs = new CryptoStream(ms,
sa.CreateDecryptor(), CryptoStreamMode.Read);

    byte[] plainbyte = new byte[r1.Length];
    cs.Read(plainbyte, 0, r1.Length);
    cs.Close();
    ms.Close();

    char[] plaintext = Encoding.UTF8.GetChars(plainbyte);
    for (int w = 0; w < plaintext.Length; w++)
        DecryptedIncommingMessageBox.Text +=
Convert.ToString(plaintext[w]);
    DecryptedIncommingMessageBox.Text += "\n\n";
}

```

```

        private void OutgoingMessageBox_KeyDown(object sender,
        KeyEventArgs e)
        {
            try
            {
                if (e.KeyCode == Keys.Enter &&
                OutgoingMessageBox.ReadOnly == false)
                {
                    encryptoutgoinmessage(OutgoingMessageBox.Text);
                    OutgoingMessageBox.Clear(); // clear the message
                }
            }
            catch (Exception error)
            {
                MessageBox.Show(error.ToString());
            }
        }
    }
}

```

6.9.11 Settings Code :

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace KDC_Client

```

```

{
    public partial class Program : Form
    {
        public Settings()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            string ipadd = textBox1.Text;

            FileStream a = new FileStream("KDC\\KDC
Client\\config.txt", FileMode.Create, FileAccess.Write);
            StreamWriter op = new StreamWriter(a);

            op.WriteLine(ipadd);
            op.Close();
            a.Close();
        }

        private void button2_Click(object sender, EventArgs e)
        {
            string timeoutmin = textBox2.Text;
            string timeoutsec = textBox3.Text;

            string temp = timeoutmin + " " + timeoutsec;

            FileStream a = new FileStream("KDC\\KDC
Client\\timeout.txt", FileMode.Create, FileAccess.Write);
            StreamWriter op = new StreamWriter(a);

            op.WriteLine(temp);
            op.Close();
            a.Close();
        }
    }
}

```

Chapter 7

Installation And testing

7.1 Setup Guide:

1. First check for the minimum requirements is met:

Minimum Hardware Requirement:

128MB RAM

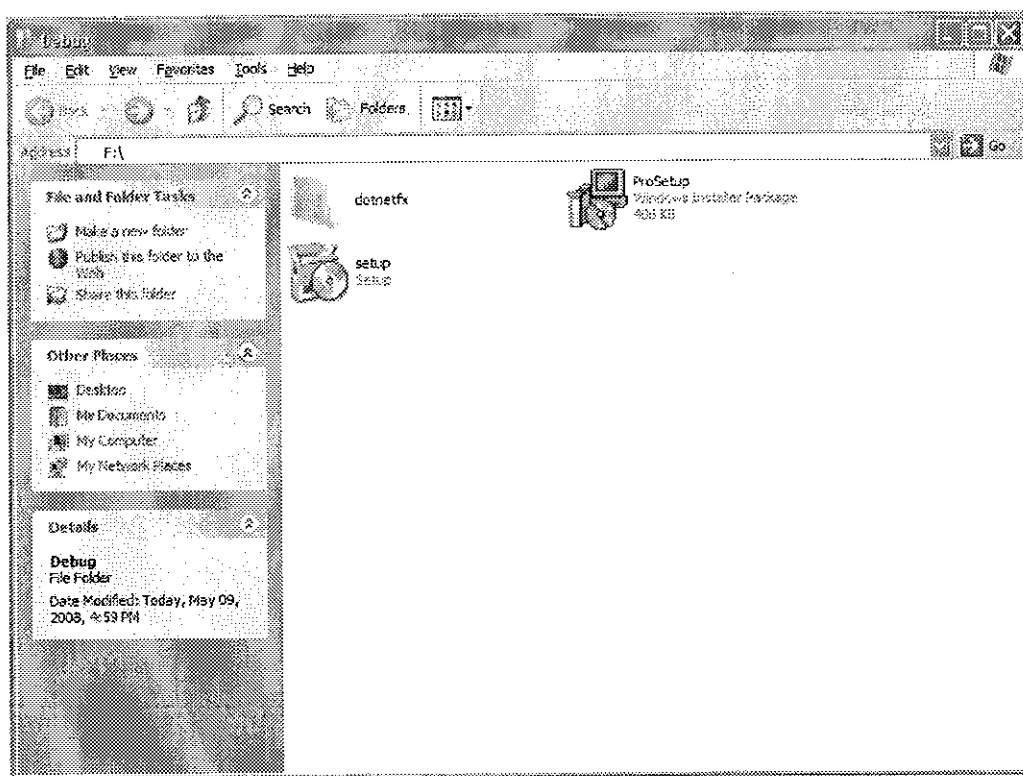
PIII processor

Minimum Software Requirement:

Windows 98, XP

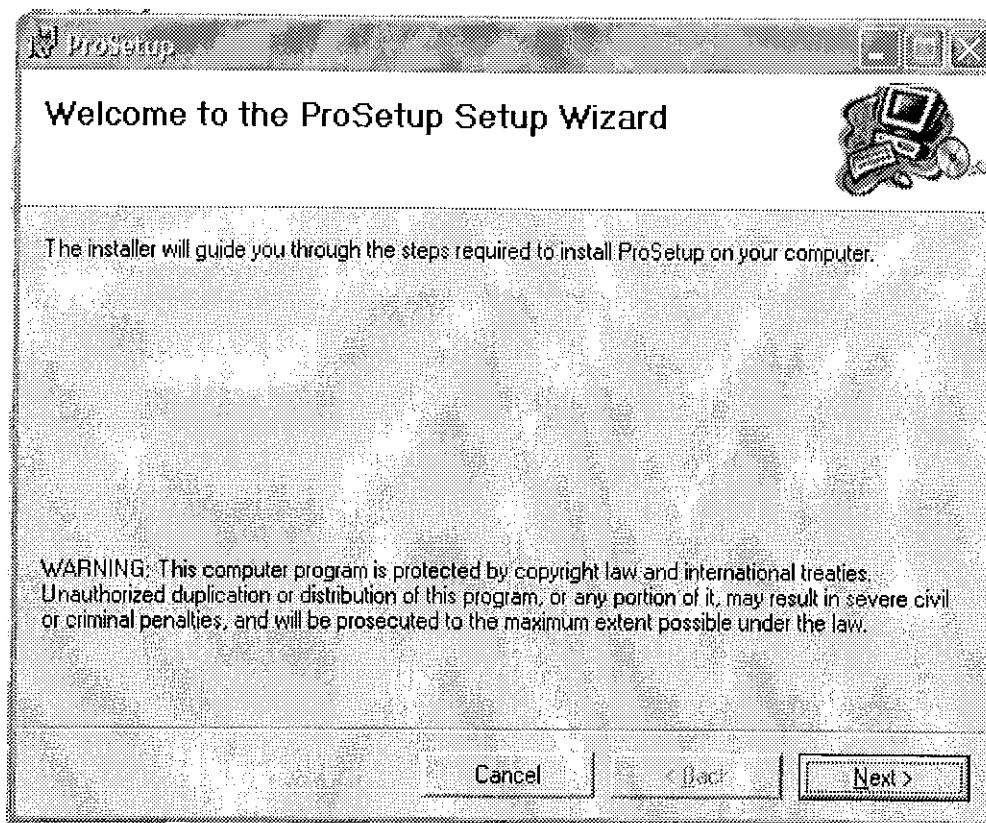
2. Insert the CD

3. Open the CD-ROM Drive location



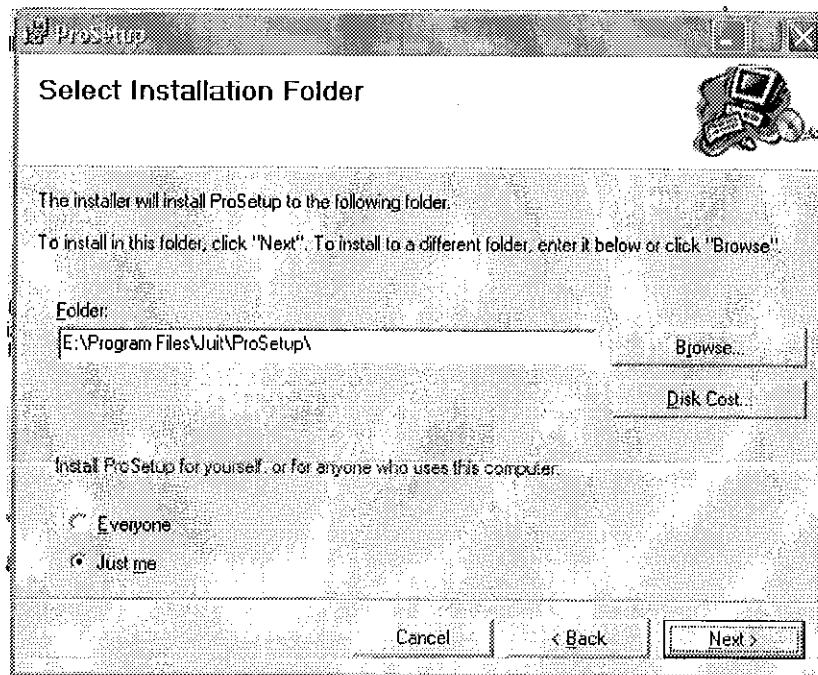
4. Double click the file names “setup”

5. The following window will appear



6. Press "Next" button

7. The following window appears



8. Enter the path of the folder where you want install the software.

Default path is "<windows drive letter>:\Program Files\Juit\ProSetup\"

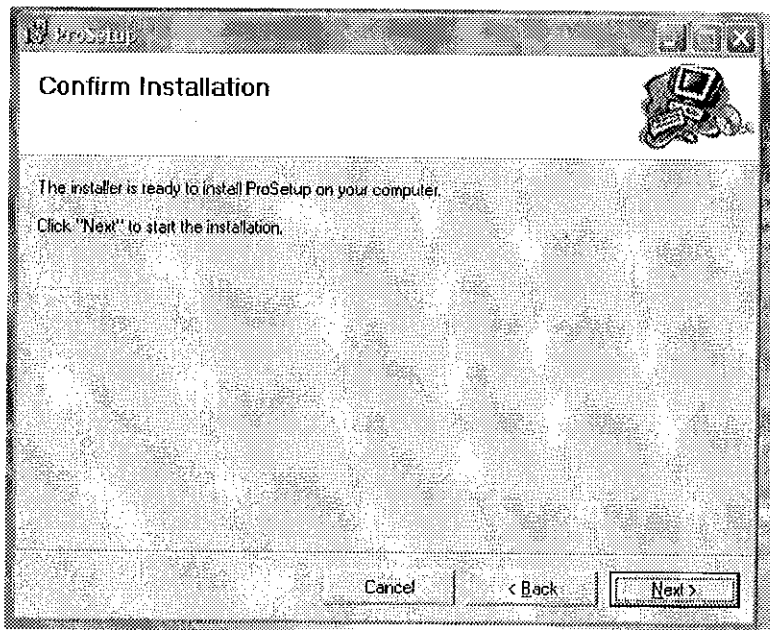
9. Select the user for which the software will be installed.

Everyone –installs the software for all users

Just me- install the software for the current user

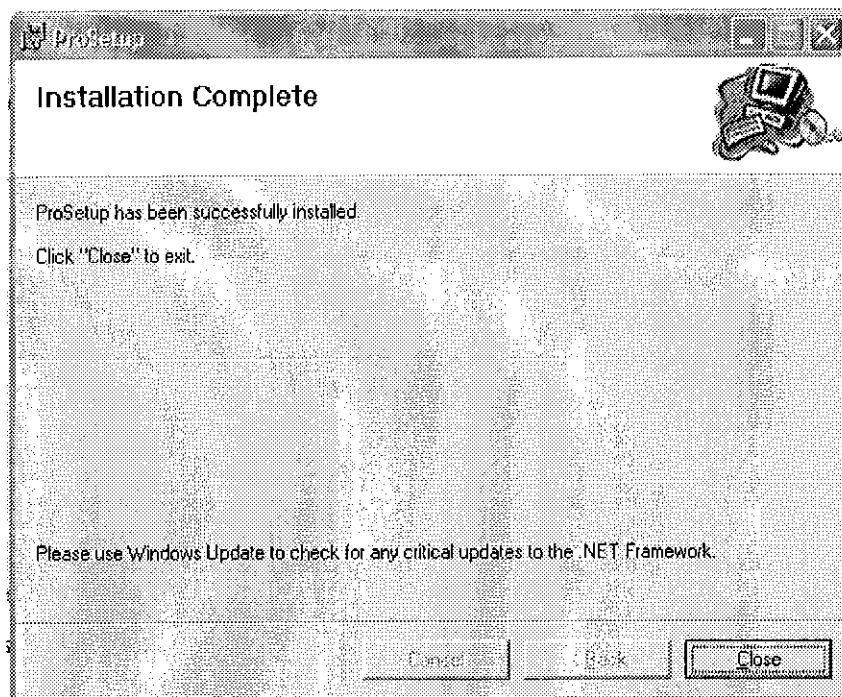
10. After enter the path and the user, Press "Next"

11. Confirmation window appears



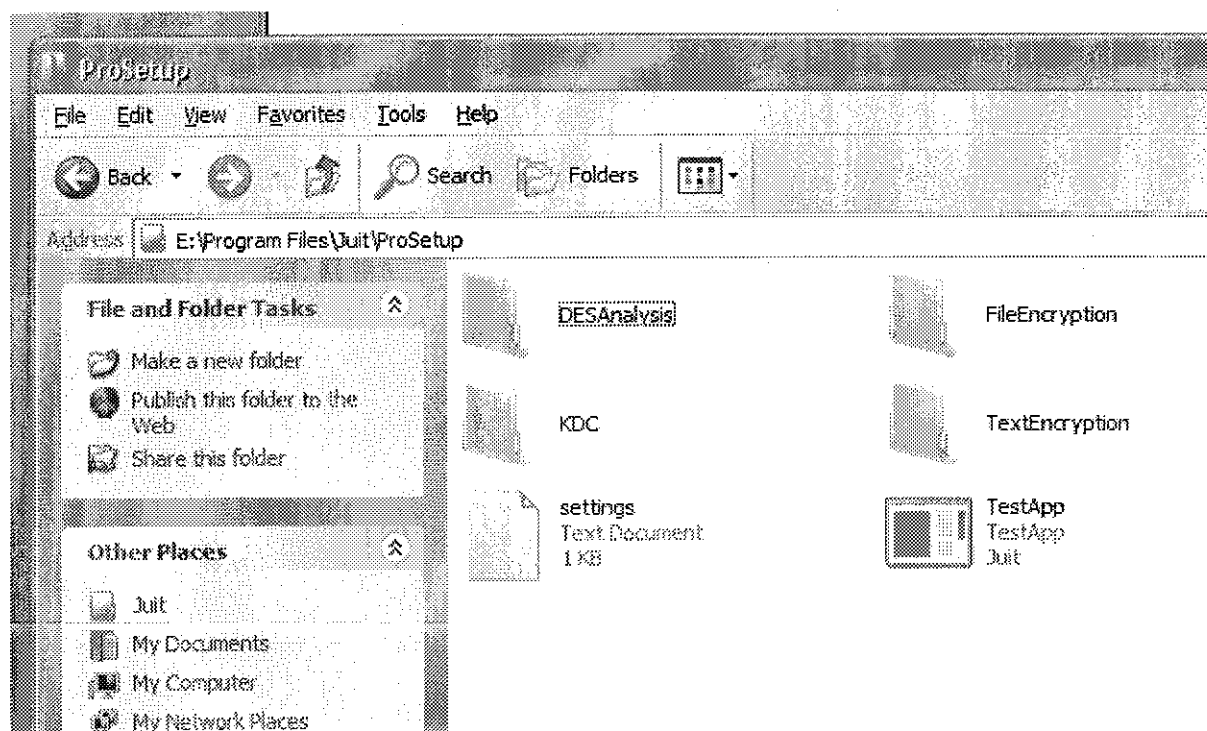
12. Press "Next" to confirm or "Back" to change previous settings.

13. When "Next" is pressed the installation completes itself automatically.



14. Press "Close" to finish installation.

15. Browse to the folder where software is installed.



16. Double click the "TestApp" file.

17. This brings to the main screen of the software.

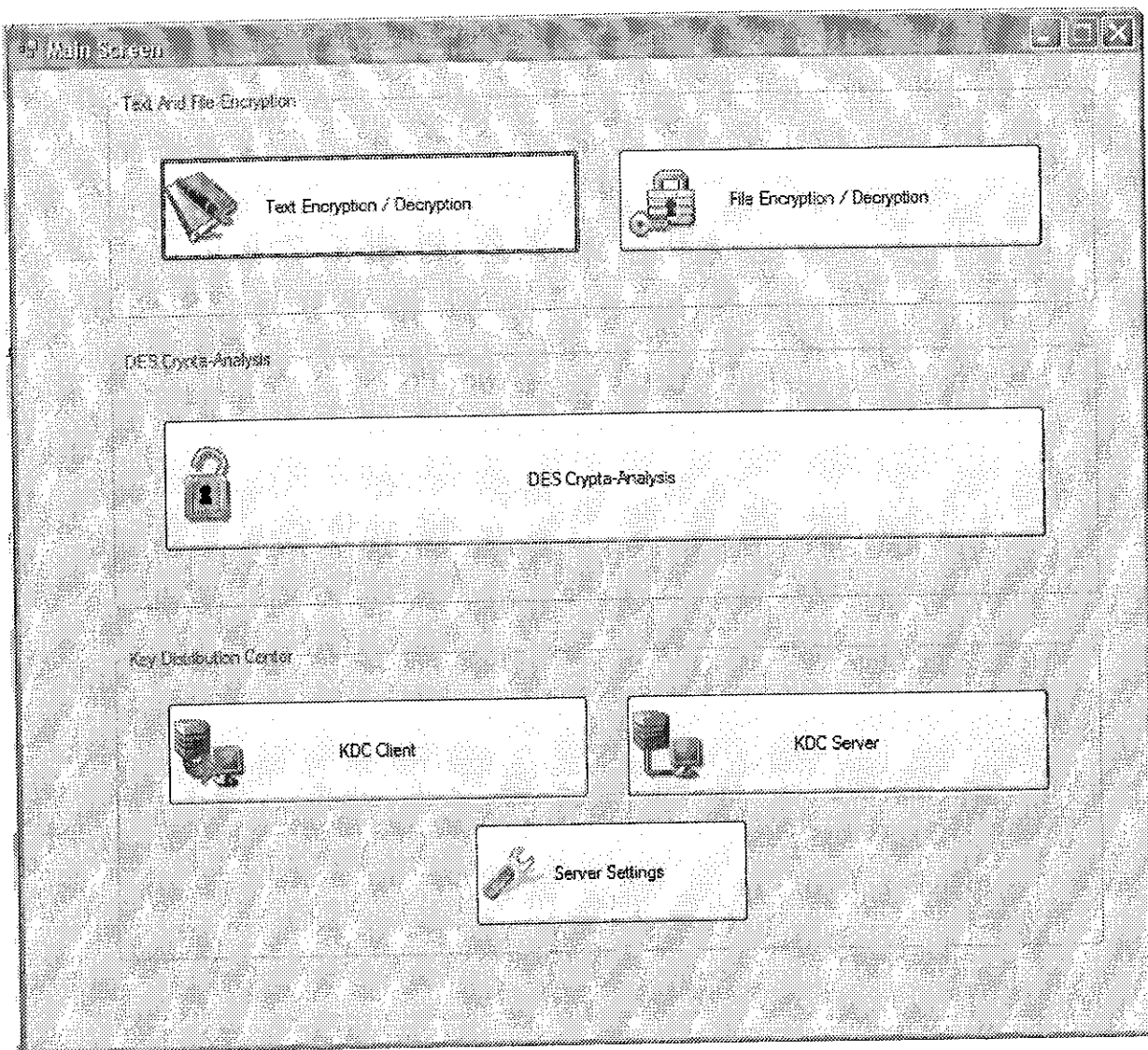


Figure Software Main Screen

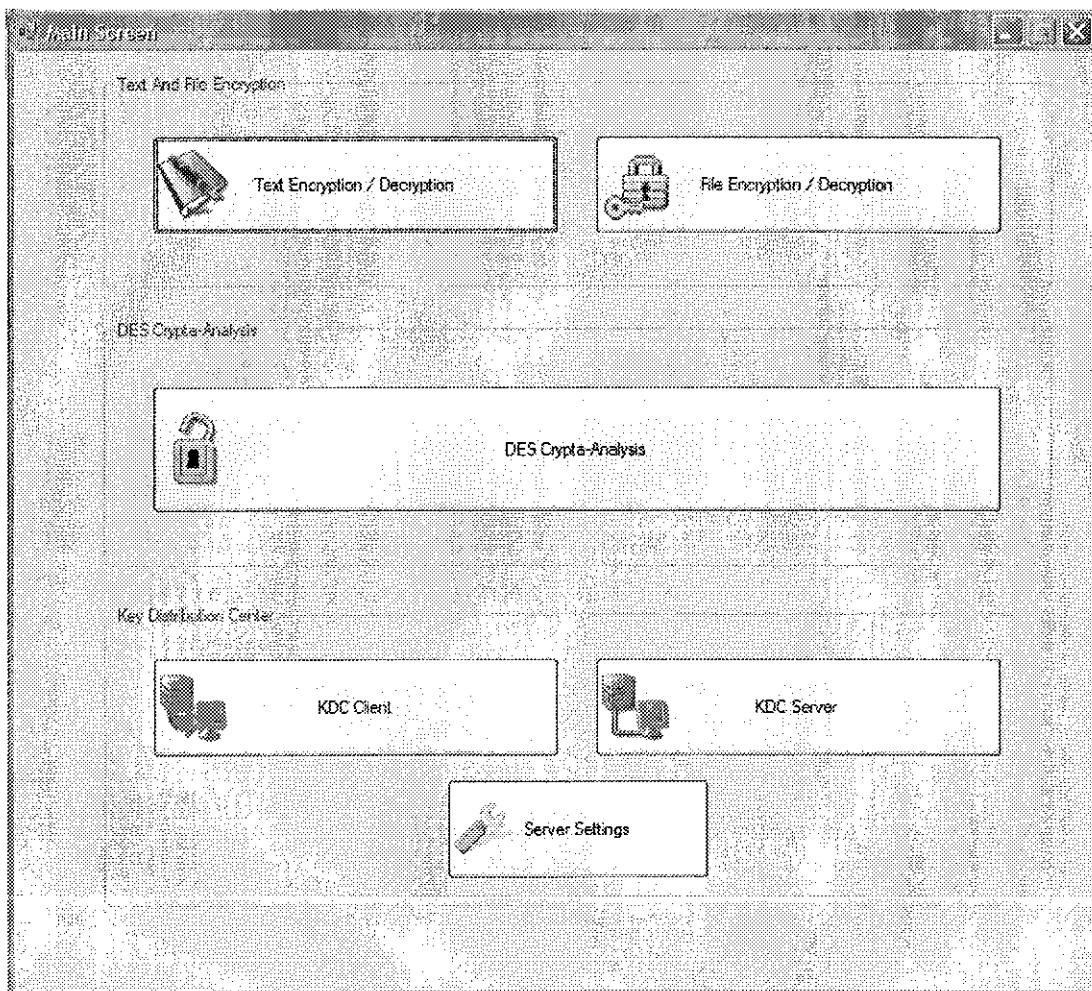
Source Code Compilation

- i. Browse to the "<cd-rom drive>:/code/TestApp/"
- ii. Open the TestApp.sln/TestApp.vcproj file in visual studio 2005.
- iii. Go build menu, and press "Build" to compile the code.

7.2 User Guide

The Main Screen

1. Text and File encryption
 - I. Text Encryption/Decryption
 - II. File Encryption/Decryption
1. DES Cryptanalysis
 - I. KDC Client
 - II. KDC Server
2. Key Distribution Center
 - III. Server Settings



Text Encryption using Symmetric Algorithm

The screenshot shows a 'Text Encryption' application window with three tabs: 'Symmetric Algorithms', 'Asymmetric Algorithms', and 'Digital Signatures'. The 'Symmetric Algorithms' tab is active, displaying a list of algorithms: DES (selected), Triple DES, Rijndael, and RC2. To the right, the 'Mode' section includes radio buttons for ECB (selected), CBC, CFB, OFB, and CTS. The 'Padding' section includes radio buttons for PKCS7 (selected), Zeros, and None. Below these, there is a 'Use User Key' button and a text input field with the instruction '(Please Leave The Space Blank To Generate A Random Key)'. The main interface features three text areas: 'Plaintext' with an 'Open TextFile' button below it, 'CipherText As Text String' with a 'Save Encrypted Message' button below it, and 'Recovered Plaintext'. To the right of the 'Plaintext' and 'CipherText' areas are 'Encrypt' and 'Decrypt' buttons respectively.

- i. Select any of the given symmetric algorithms.
- ii. Enter mode of operation.
- iii. Enter padding type.
- iv. Enter Key.
- v. Enter PlainText
- vi. Press "Encrypt" button.

Text Encryption using Asymmetric Algorithms

The screenshot shows a software interface for text encryption. At the top, there are three tabs: 'Symmetric Algorithms', 'Asymmetric Algorithms' (which is selected), and 'Digital Signatures'. Below the tabs is a 'New Parameters' button. The main area is titled 'Parameters Details' and contains an XML editor with the following content:

```
<RSAKeyValue><Modulus>
0VyTMAyYwYVWqW6LaCq+jlyWhgBHhtbuTc+oDFPtv95sEZ9obzC5FN68MUlvMdoSan2P3ewvwXvH9BBov
0KnPGWlt+3KeNWuQg17aQ844kHpFrde+
0XqJcVZnGq7HF4QkUBJUEg/aYFGjNh0vj277hnoBybgMck2uYVWzwns=</Modulus><Exponent>AQAB
</Exponent><P>+hE64zzMEbvrVH5DJ3Zt+MScq7p/
+kKQmNI9T02p65ESS69oUZTpZuJxPUCkz154VqQ7ggZUEroMhqznoVZw==</P><Q>
lIQfCPleeY2kD//Z58lohw+jfqNWpyLU2xC3c65voD31roSTPUiKzi5KPINLJMXN9rDY4F1d9QkDeuVVAJzQ
==</Q><DP>
mhpylsU9sRQQYwJoWfVw5MNvg/McyxMounjMxUOS4eS3ur2uuK8MwR8azeAydR8Oege41qdxChUPYM
RS2rasw==</DP><DQ>+xWSHv2wq53cwLA5NtUus+
19sCGPCr3oq58KkxSMAQtJrUe2coilDCs3rARLALJAiSIVlyRshPL8dlTmM4Q==</DQ><inverseQ>
```

Below the XML editor is a 'View The XML File' button. The interface then has three main sections:

- Plaintext:** A large text input area with an 'Open File' button below it and an 'Encrypt' button to the right.
- Ciphertext As Text:** A large text input area with a 'Save Encrypted Message' button below it.
- Recovered Plaintext:** A large text input area with a 'Decrypt' button to the right.

- i. Press "New Parameters" button to generate new parameters.
- ii. Enter Plaintext
- iii. Press "Encrypt" button
- iv. Press "Save Encrypted Message" to save the encrypted message.
- v. Press "Decrypt" button to decrypt.

Digital Signature

The screenshot shows a window titled "Text Encryption" with three tabs: "Symmetric Algorithms", "Asymmetric Algorithms", and "Digital Signatures". The "Digital Signatures" tab is selected. It contains two main sections: "Using RSA" and "Using DSA". Each section has three text input fields: "Message", "Hashed Message", and "Digital Signature". Below the "Message" field in the "Using RSA" section are buttons for "Sign", "Verify", and "RSA Parameter Details". Similarly, below the "Message" field in the "Using DSA" section are buttons for "Sign", "Verify", and "DSA Parameter Details".

- i. Enter the message to be signed.
- ii. Press "Sign" button.
- iii. The Hashed message and Digital Signature will be displayed.
- iv. Press "Verify" button to verify the message.
- v. Press "RSA Parameter Details" or "DSA Parameter Details" to view the parameters saved in a xml file which can be opened in Internet Explorer.

File Ecryption

File Encryption

Encrypt File Decrypt File

Choose Symmetric Algorithms

☒ DES ☐ Triple DES ☐ Rijndael ☐ RC2

Enter Encryption Password

Enter The Encryption Password :

Browse Files

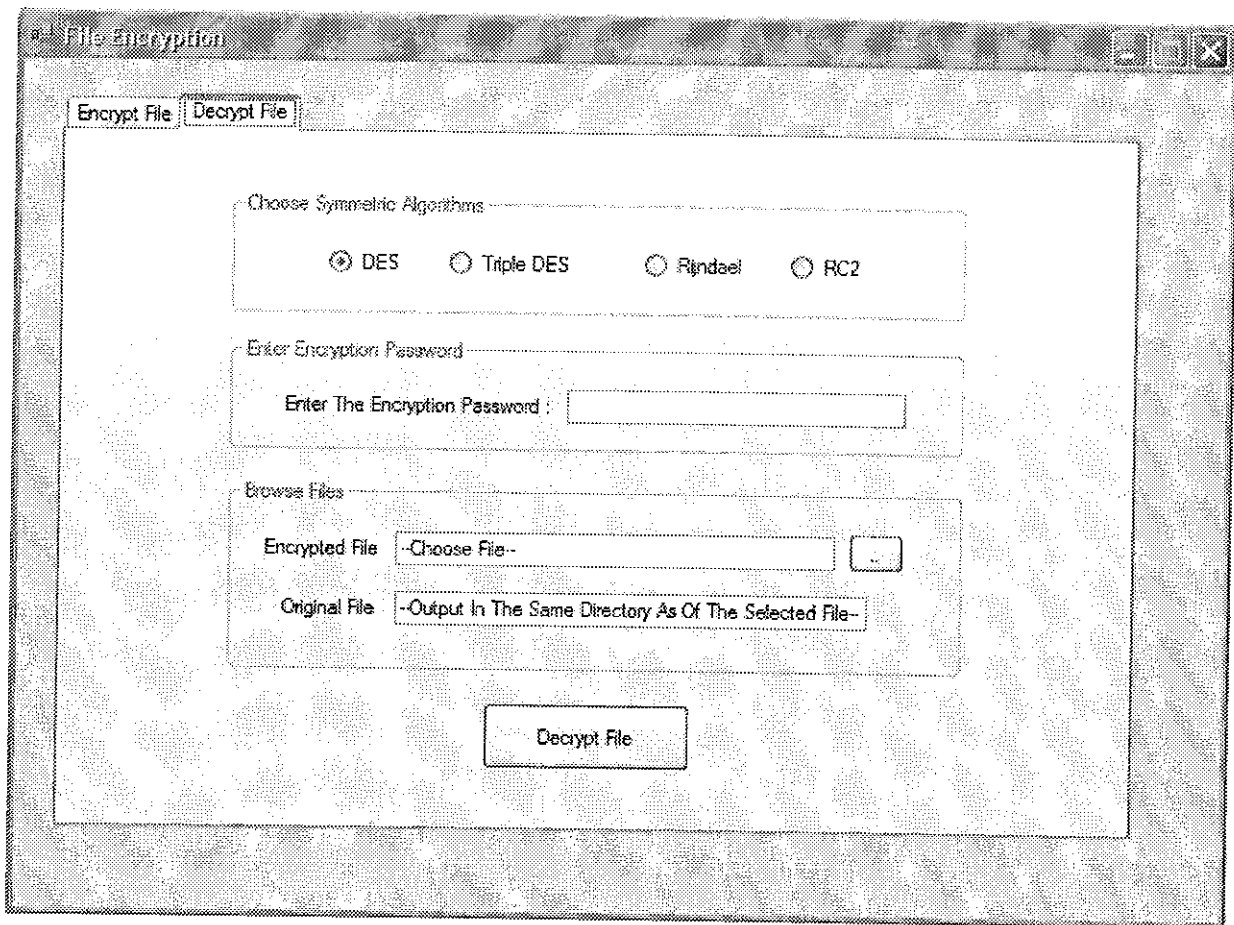
Original File -Choose File-

Encrypted File -Output In The Same Directory As Of The Selected File-

Encrypt File

- i. Choose the algorithm by selecting the radio button.
- ii. Enter the key.
- iii. Choose the file by pressing the "Browse" button.
- iv. Enter the destination location to save the encrypted file.
- v. Press "Encrypt File" button to encrypt the file.

File Decryption



- i. Choose the algorithm by selecting the radio button.
- ii. Enter the key.
- iii. Choose the file by pressing the "Browse" button.
- iv. Enter the source location of the encrypted file.
- v. Press "Decrypt File" button to decrypt the file.

DES Cryptanalysis

The screenshot shows a window titled "DES Cryptanalysis" with a standard Windows-style title bar. The window is divided into two main sections: "DES Encryption" and "Crypta-Analysis".

DES Encryption Section:

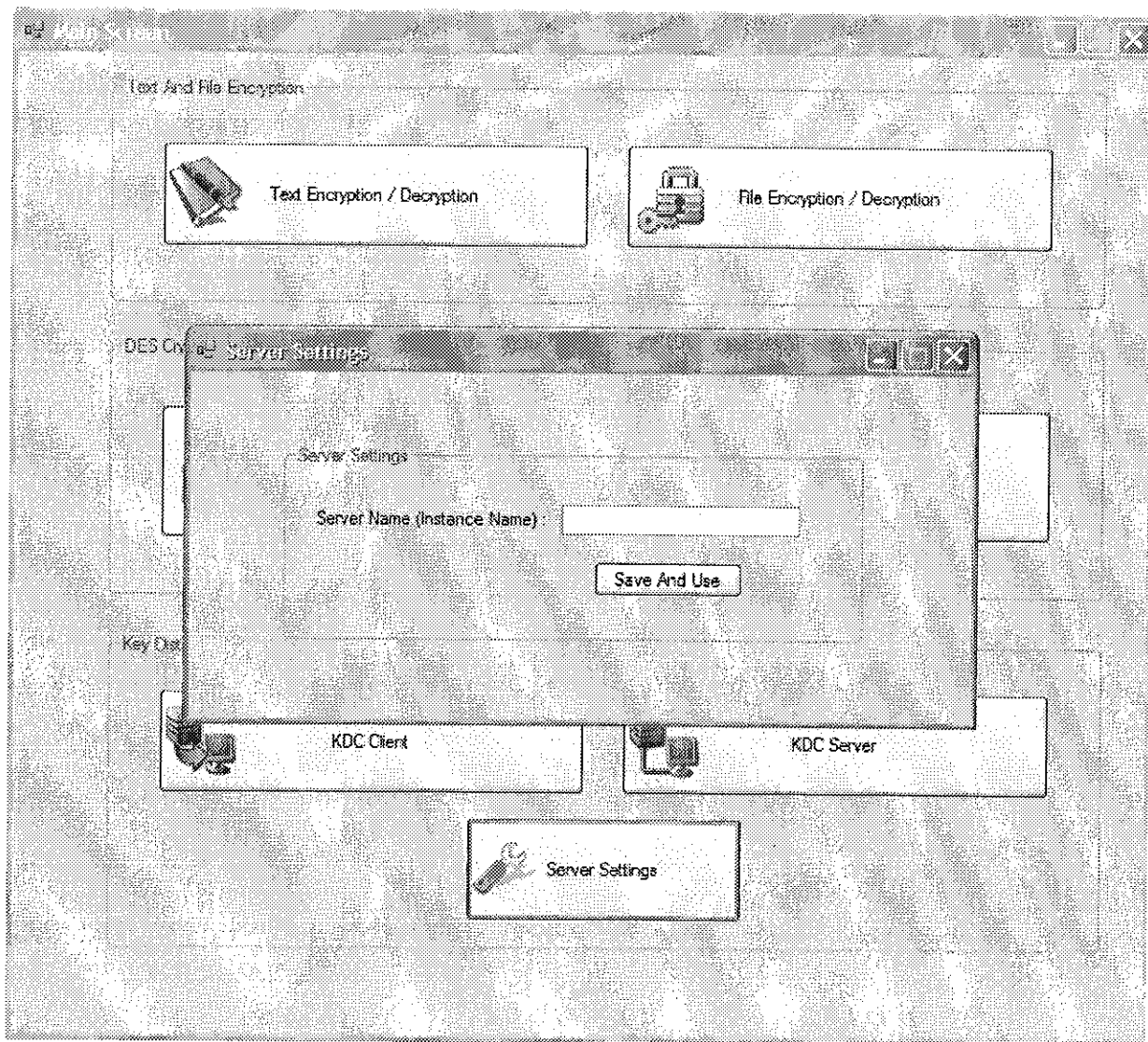
- At the top, there is a label "Enter Key" followed by a text input field. To the right of the field is the instruction "(Please Enter Numeric Key Only)".
- Below the key input is a label "Plaintext" followed by a larger text input field.
- To the right of the plaintext field is a button labeled "Encrypt".
- Below the plaintext field is a label "Cipher Text As Text String" followed by another large text input field.

Crypta-Analysis Section:

- At the top of this section is a label "Supplied Ciphertext" followed by a large text input field.
- To the right of the ciphertext field is a button labeled "Start Crypta-Analysis".
- Below the ciphertext field is a label "Obtained Key" followed by a text input field.
- Below the key field is a label "Recovered Plaintext" followed by a large text input field.

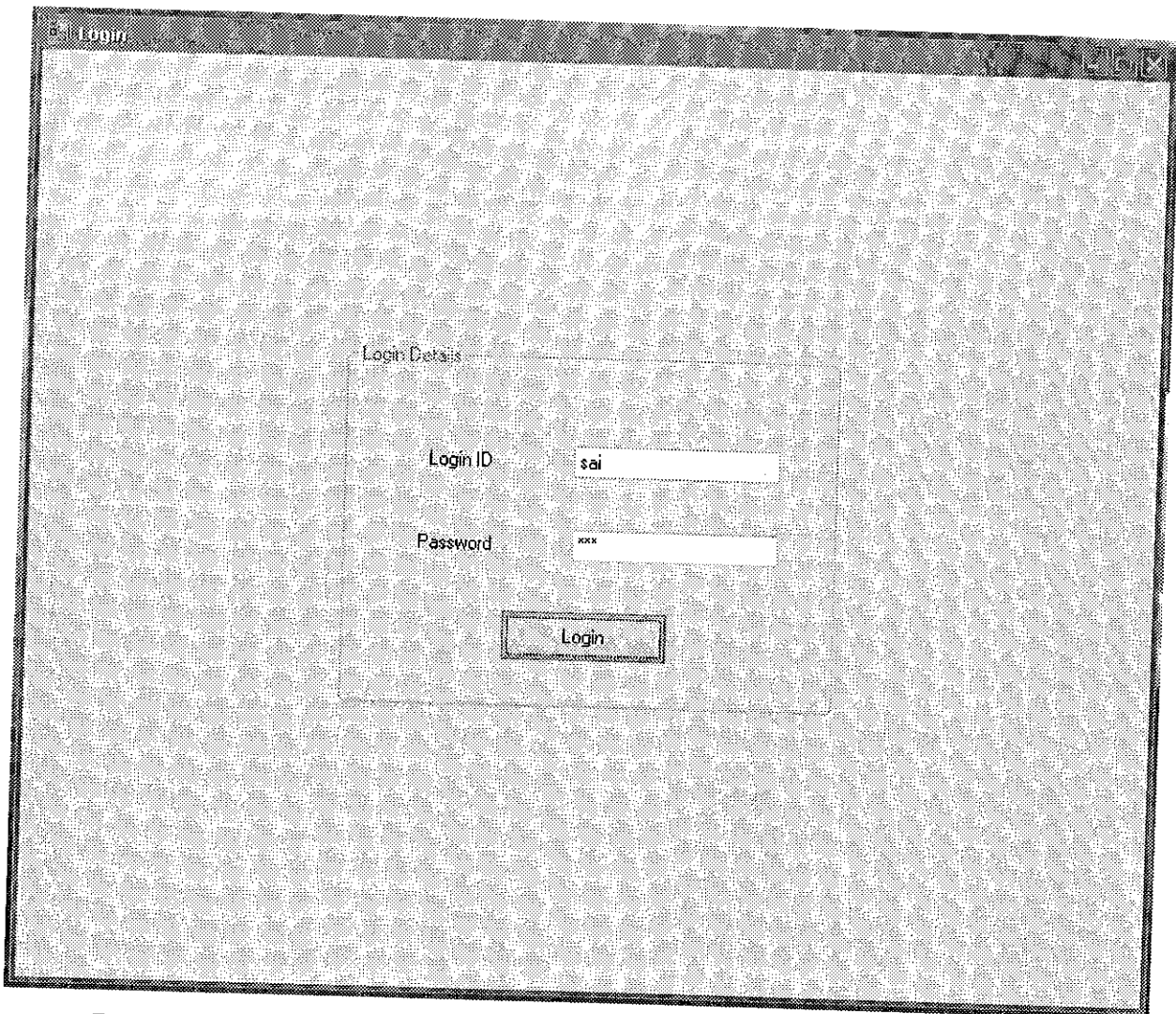
- i. Enter Key (only numeric should be entered).
- ii. Enter the Plain Text
- iii. Press "Encrypt" button to encrypt the plain text.
- iv. Cipher Text as Text String will be displayed.
- v. Press "Start Crypta-Analysis" button to begin the crypta-analysis.

KDC Server Settings



- i. Enter the server (instance name)
- ii. Press "Save and Use" button.

KDC Client- Login Screen



The screenshot shows a window titled "Login" with a standard Windows-style title bar. The window has a light gray background with a subtle grid pattern. In the center, there is a "Login Details" section enclosed in a thin border. This section contains two input fields: "Login ID" with the text "sai" entered, and "Password" with masked characters "XXXX". Below these fields is a "Login" button. The window also features standard Windows window controls (minimize, maximize, close) in the top right corner.

Login

Login Details

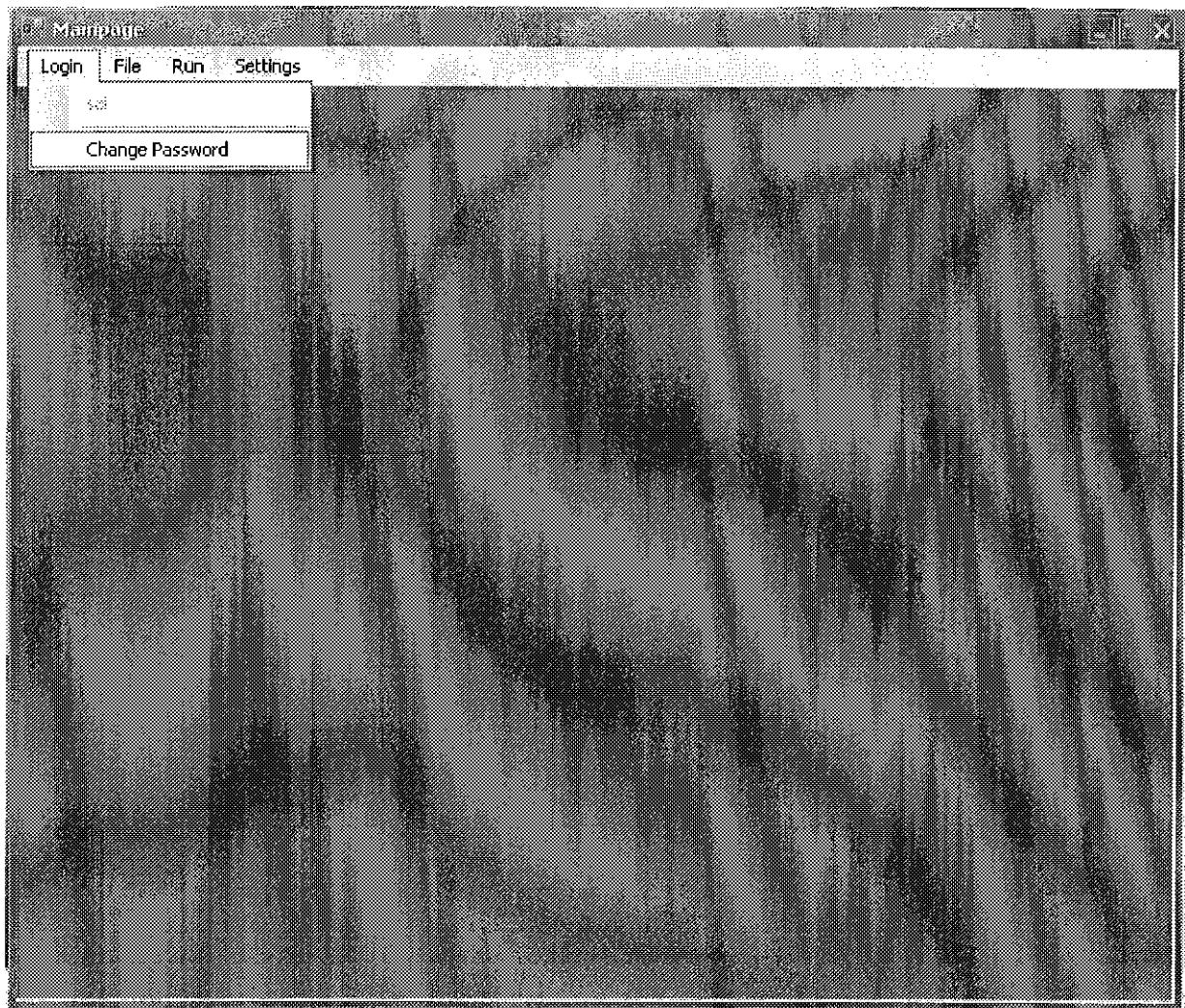
Login ID: sai

Password: XXXX

Login

- i. Enter the Login ID
- ii. Enter the password
- iii. Press "Login"

KDC Client- Main Window



Contains Option

- i. Change Password
- ii. Session Key Exchange
- iii. Change Key Pairs
- iv. Settings
- v. Client Server

KDC Client –Change Password

The screenshot shows a graphical user interface for changing a password. The window is titled 'Mainpage - [Change Password]'. It features a menu bar with 'Login', 'File', 'Run', and 'Settings'. The main content area is titled 'User Details' and contains four text input fields arranged in a 2x2 grid. The fields are labeled 'Login ID', 'Old Password', 'New Password', and 'Confirm Password'. Below these fields is a single button labeled 'Change Password'.

- i. Enter Login ID
- ii. Enter Old Password
- iii. Enter New Password
- iv. Enter Confirm Password
- v. Press "Change Password"

KDC Client -Session Key Exchange

Mainpage [Session Key Exchange]

Login File Run Settings

Follow The Following Steps Given Below For Session Key Exchange

Step 1: Get Public Key Of Client

Connect To KDC To Get Key Of Client

Step 2: Connect To Client

IP Address Of Client: 172.18.5.232

Connect

Step 3: Generation Of Session Key

Symmetric Algorithm:

☒ DES ☐ Triple DES ☐ Rijndael ☐ RC2

Generate IV And Key

Step 4: Encrypt Session Key And IV

Encrypt Data Using Client Public Key

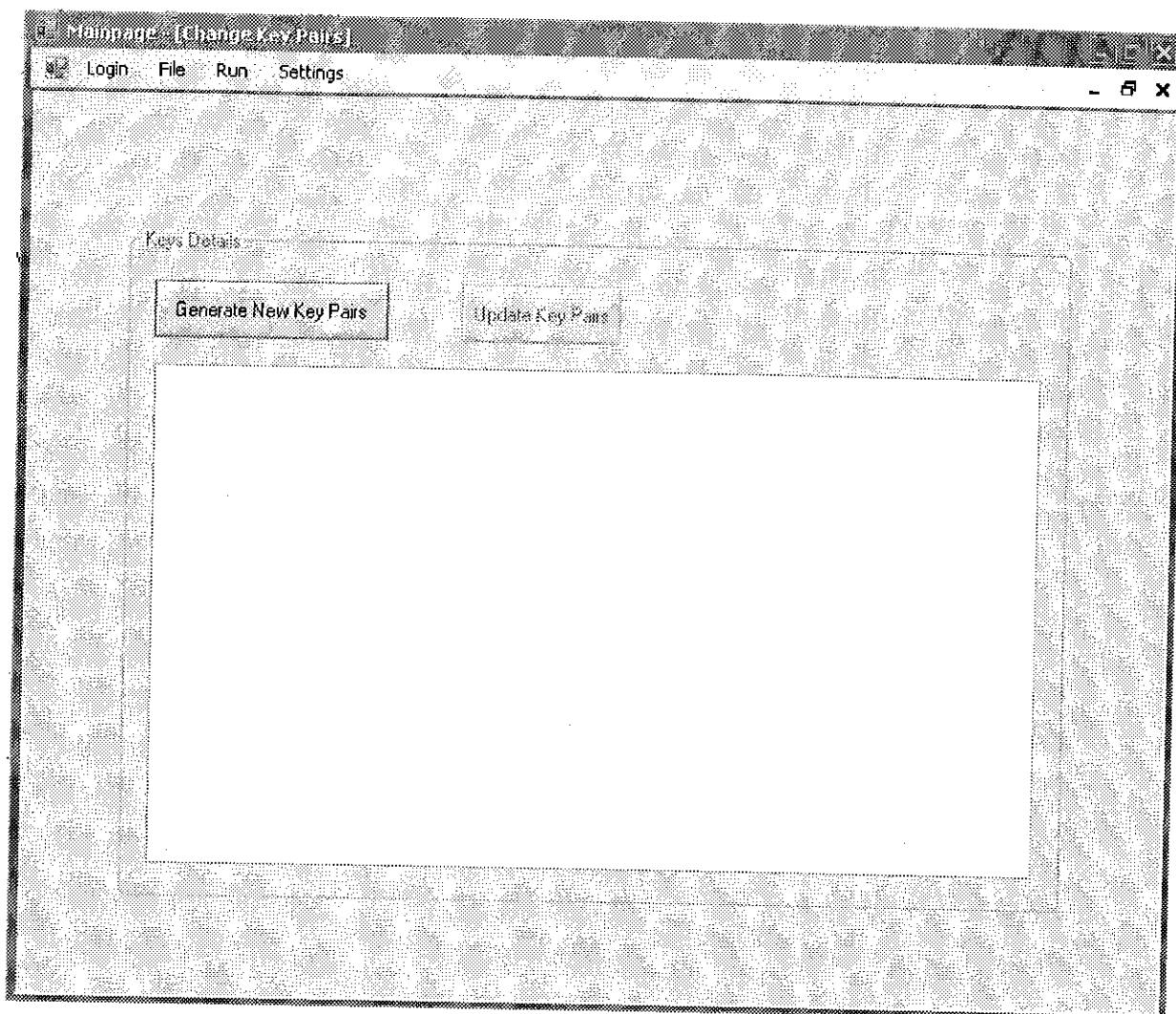
Encrypt Data Using Own Private Key

Step 5: Send Data To Client

Send Data To Client And Exchange Message

- i. Press "Connect To KDC To GET Key Of Client" to connect to KDC
- ii. Enter the "IP Address Of Client"
- iii. Press "Connect" button
- iv. Choose the type of algorithm
- v. Press "Generate IV and Key"
- vi. Press "Encrypt Data Using Client Public Key"
- vii. Press "Encrypt Data Using Client Own Private Key"
- viii. Press "Send Data To Client And Exchange Message"

KDC Client – Change Key Pairs



- i. Press "Generate New Key Pairs"
- ii. New key pairs are generated and displayed.
- iii. Press "Update Key Pairs" to send the updates key pairs to KDC.

KDC Client- Settings

The screenshot shows a terminal window titled "Mainpane - [options]" with a menu bar containing "Login", "File", "Run", and "Settings". The window displays two configuration sections:

KDC Client Configuration

Enter the KDC Server IP Address

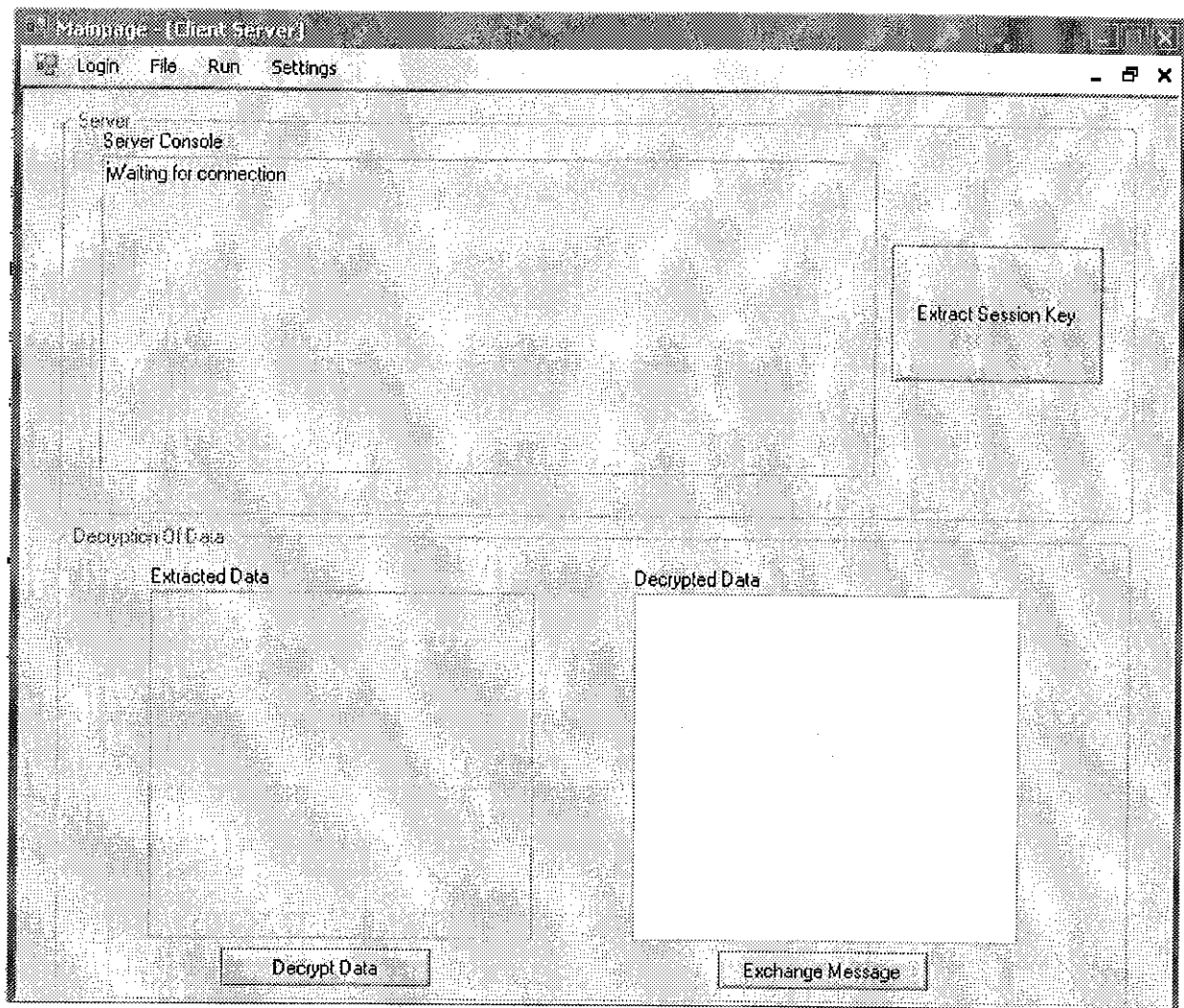
KDC Session Key Timeout

Enter The Timeout Period Of The Session Key

Mins	Secs
2	00

- i. Enter the Ip address of KDC server.
- ii. Press "Save And Use" button.
- iii. Enter the Timeout period of session key
- iv. Press "Save And Use" button.

KDC Client – Client Server



- i. When connection is received
- ii. Press "Extract Session Key" button to extract the session key.
- iii. Press "Decrypt Data" to decrypt the data.
- iv. Press "Exchange Message"

7.3 Test Data

Text Encryption

SYMETTRIC ALGORITHM

ALGO	MODE	KEY	PLAINTEXT	CIPHERTEXT	RECOVEREDTEXT
DES	ECB	default	bharat	hjlXGHvwVHw=	bharat
		default	sairaju	lMb8Zs3Nfu0=	sairaju
		12345678	sairaju	qsA2xpvbAUk=	sairaju
	CBC	default	sairaju	gmFaH7kulMc=	sairaju
		12345678	sairaju	hxpKI68Z9Dk=	sairaju
	CFB	default	sairaju	FZu+r8xrNRs=	sairaju
	ECB	default	SAMPLE INPUT.TXT	aS49IA7yzVaVP9w4 Mx8bqEqhygbJ1nOC	This is a test input.

ALGO	MODE	KEY	PLAINTEXT	CIPHERTEXT	RECOVEREDTEXT
TripleDES	ECB	default	sairaju	iuslr8vF9tl=	sairaju
		default	sairaju	0HTz7wFRCu4=	sairaju
		1223333444444 44444111111	sairaju	5dWWuI9tFcw=	sairaju
	CBC	default	sairaju	3+7EwaB00a4=	sairaju
		1223333444444 44444111111	sairaju	KMjhvnNQ/Fw=	sairaju
	CFB	default	sairaju	ISiYmDwZzKk=	sairaju
	ECB	default	SAMPLE INPUT.TXT	+OUNtO2W4/HxaLfec JBSK/1dHGvklX+m	This is a test input.

ALGO	MOD E	KEY	PLAINTEXT	CIPHERTEXT	RECOVEREDTEXT
RIJNDAEL	ECB	default	sairaju	+h8xjF+qQhQHjO/Vd8HQIQ==	sairaju
		default	sairaju	UxdolKqywMNDGJBD7qU8bA==	sairaju
		1234567890123456 1234567890123456	sairaju	fDMAok8p7lb6WCJRLz0pOA==	sairaju
	CBC	default	sairaju	lhSJYzhOlzsfzvTNJE+Baw==	sairaju
		1234567890123456 1234567890123456	sairaju	b81ibdmdeE/Ym8PManA+5Q==	sairaju
	CFB	default	sairaju	E9iscNWXB+DfaAc42FI/tg==	sairaju
		1234567890123456 1234567890123456	sairaju	YapkZy973J+m4BuJscudZQ==	sairaju
	ECB	default	SAMPLE INPUT.TXT	u+KrQ+0kFW0J0QOxr8pLDWHfwDHFO ZY	This is a test input.

ALGO	MODE	KEY	PLAINTEXT	CIPHERTEXT	RECOVEREDTEXT
RC2	ECB	default	sairaju	2FOpXlhU1tQ=	sairaju
		default	sairaju	9epQdcsvJ1M=	sairaju
		12345678 12345678	sairaju	sZVq2zkS1IY=	sairaju
	CBC	default	sairaju	K+DnH7bgZv0=	sairaju
		12345678 12345678	sairaju	3c617hKRnx8=	sairaju
	CFB	default	sairaju	bmjKmoIwass=	sairaju
		12345678 12345678	sairaju	x4NAZ7RXB/w=	sairaju
	ECB	default	SAMPLE INPUT.TXT	Mzwx4YiUfdD4eJcB1pClnGFH54gTTPrG	This is a test input.

ASYMMETRIC ALGORITHM

PARAMETERS	PLAIN TEXT	CIPHERTEXT	RECOVERED TEXT
<p><P> <P>91XhNpKsM2Zxz0Md06UFsCziY / yIigTZsmBKXcxEhuNGyO7yPRsP5DQ L kQrGKER+QRIqW5gZRUI9vHTMBsO nZQ==</P> <Q>46AsZITfzlo5n053dw6xFuAhXSHXM VtxoZ BYjJwCcmuWQxKvTuXfdfCrE/MEDZvW2 DhxK IUyddLk3PV3or6GQ==</Q></p>	sairaju	Y0C1pxPKQJFXoEW2Te/s6M1KYB6Qh s ygNdy5SxnPSzubjrxSWAu+rmx fDESaqNAj0nNr1xCsorBSbljV4O6La uMuHa0kWHy0+FZFfzZjWFSWxD17 dp1laYh4aPD3RuI9G2Y+/LzuOQ9TkX CnwGExj45ZUeS6Pweu2YPLn5XwEw=	sairaju
<p><P>+inoP6EljrYAgIAATdLRpaUF8FV D/5ikdb7mJmOopTp8yTruIjF+aBOU MVyX23RTuKj6dwA3OtnD761ZVQJw Mw==</P> Q>6oD9ntoHZvTugXvEyyHZBh2+xH3nB GYIHZdPTIDh8NngxUI9QHwavDir5pWXNd qH6fmgzaGIROkEhw7q0xg4sEQ==</Q></p>	sairaju	bypXQZpf1gMOT8/3wkEFsndtrakSxMUz E+yJWECIzbq8peoNHOI49hvw3a3xvga fO+hyZemxwSnYHyxV5rmAjrO5rkBEh H6z8Nc5ARxA3czO3ygGSTu1xQ135H0 gQv6oLQSWZGdRGX7/x/bQY9genf8Gk WGzj/1nBwcwbFsTI=	sairaju
<p>P>+mzjZYEIFKFXV36nSCGWV+FhA FpDGWilpgElGgdARTid5OsCcNLdu3k NPpNFsS4nkwHetU69Gw4eIXUJFpYj GQ==</P> <Q>4j45cmJJ+1hGP4W3CcJmJygfW 54fOVSMcBkWX0PsjQWUm3z+ztUG JDyGrC1qKYVJtnnobCIIGNsyLuokos DBmQ==</Q></p>	sairaju	LvzcIV6qvp0abwGxCgWjZBxXKvPNByX 9hy1/Aov+3Q8LNdhoudRr9eEYCKMUol T+ffcfNbnvaV7AyzRkPyAOGtqioO08R9z q10CqjZWLKKA3tQ4s1M0UbpUwduQD Rq/mkYxyloT1MQ3KLxjWHWNu7jncuHJ 6qEXuU0LxqsbJnZE=	sairaju

DIGITAL SIGNATURE

ALGORITHM USED	MESSAGE	HASHED MESSAGE	DIGITAL SIGNATURE
	manishkumar	J0b3k%!0000_00 0	A001_070v00q0#00h00V000_000000 0\$0>00E000_F0DF P0g0?0OR_0(2gr_j0%0i00X00=MP0LIOF0GDeVm00L00g0y
	KUMAR SAURAV	J0b3k%!0000_00 000VI00q0J22!h0T 00	A001_070v00q0#00h00V000_000000 0\$0>00E000_F0DF P0g0?0OR_0(2gr_j0%0i00X00=MP0LIOF0GDeVm00L00g0yg0)___0.20*0"I_000 0)_!0#0(8H00_'0000 4O_i_R00000yn080 0 <qu0#(0*_K0`d°_0Y_00q0P.E0I.
	KUMAR SAURAV	00VI00q0J22!h0T 00	_j0000(0}2_090@0/000_0QRY0"0i_#L,0_hn_0
	manishkumar	00VI00q0J22!h0T 00J0b3k%!0000_00 0	_j0000(0}2_090@0/000_0QRY0"0i_#L,0_hn_0x_m0+N(0'0} _0Am0s_:0c00'000H0_0Q00 00_

DES CRYPTA-ANALYSIS

KEY	PLAINTEXT	CIPHERTEXT AS TEXT STRING	SUPPLIED CIPHERTEXT	RECOVERED PLAINTEXT	RECOVERED KEY
asdfghjk	sairaju	n0Q6votd1Ag=	n0Q6votd1Ag=	sairaju	asdfghjk
12345678	MANISH	0uMm3RzgoJo=	0uMm3RzgoJo=	MANISH	12345678
22442244	KUMAR	V9CXXHdKjPE=	V9CXXHdKjPE=	KUMAR	22442244
33663366	BHARAT	FWogLKIB/4Q=	FWogLKIB/4Q=	BHARAT	33663366
10000000	THIS IS TEXT	+/Rx7Ar6Kj1nVXv9LM1D1Q==	+/Rx7Ar6Kj1nVXv9LM1D1Q==	THIS IS TEXT	10000000

FILE ENCRYPTION

ALGORITHM	KEY	ORIGINAL FILE	ENCRYPTED FILE	DECRYPTED FILE
DES	12341234	Sai.txt	Sai.txt.enc	Sai.txt
	22442244	Bharat.txt	Bharat.txt.enc	Bharat.txt
TRIPLE DES	123456789012 123456789012	Manish.txt	Manish.txt.enc	Manish.txt
	222222333333 222222333333	Saurav.txt	Saurav.txt.enc	Saurav.txt
RIJNDABL	1234567812345678 1234567812345678	Sai.txt	Sai.txt.enc	Sai.txt
	1234567812345678 1234567812345678	Bharat.txt	Bharat.txt.enc	Bharat.txt
RC2	1234567812345678 1234567812345678	Manish.txt	Manish.txt.enc	Manish.txt
	1234567812345678 1234567812345678	Saurav.txt	Saurav.txt.enc	Saurav.txt

Chapter 8

Conclusion And Future Work

This project aims at creating a tool that encrypt/decrypt text using various algorithms. Along with encryption/decryption we also implemented digital signature. Encryption and Decryption is done using symmetric, asymmetric and hashing algorithms which are the modern day standards. We successfully implemented various algorithms like DES, 3DES, Rijndael, RC2, RSA, DSA, SHA-1. We also performed crypta-analysis of DES with certain conditions like key restrictions. We were also able to successfully implement public key distribution of secret key using Key Distribution Center with the help of P-2-P communication. The GUI provide user friendly environment.

The whole coding is done in .NET Framework. .NET Framework is cohesive and logically consistent language which enables the development of secure, high performance and highly robust applications. It also supports multithreading at the language level. Many inbuilt functions are available which makes the coding easy.

Since, the KDC implementation was not ideal, more work can be done to enhance it.

Chapter 9

Bibliography

Books:

William Stallings . *Cryptography and Network Security Principles and Practices, Fourth Edition*, Prentice Hall India 2005.

Bishop. M. *Introduction to Computer Security*. Addison-Wesley, 2005.

Johannes A. Buchmann. *Introduction to Cryptography*

Douglas Stinson. *Cryptography: Theory and Practice, Second Edition*

Web Pages :

<http://www.rsasecurity.com/rsalabs>

www.faqs.org/faqs/cryptography-faq/

www.counterpane.com

www.rsasecurity.com

www.cert.org

www.cacr.math.uwaterloo.ca/hac/

www.bxa.doc.gov/Encryption/Default.htm

www.faqs.org/faqs/cryptography-faq/