ENGINEERING AND EVALUATION OF A SECURE MULTICAST SYSTEM FOR MULTIMEDIA DATA

BY

RITESH JAIN

B.Tech., Kurukshetra University, Kurukshetra, India. 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

ABSTRACT

Recently security has become one of the most significant problems for spreading new information technology. Digital data can easily be copied and multiplied without information loss. This requires security solutions for such fields as multicasting pay-per-view video (video-on-demand) and live video-streaming (video-conferencing), since the producers seek to provide access control mechanisms to prevent misuse and theft of material. There have not been any substantial efforts to engineer and analyze such a secure multicast system, though many theoretical solutions have been proposed. We have implemented one such system and our analysis of the obtained results show that it is in fact feasible to have such s system perform well in real-time.

To Him

ACKNOWLEDGMENTS

I would like to gratefully acknowledge the enthusiastic supervision of Prof. Klara Nahrstedt during this work. I thank her for the technical discussions on the encryption schemes and for the help with the protocol in general. Graduates of the MONET Research Group are thanked for numerous stimulating discussions, help with experimental setup and general advice; in particular I would like to acknowledge the help of Niti Yadav for her support.

I am grateful to all my friends here at the University of Illinois for their understanding and providing me encouragement.

I thank the National Science Foundation for the financial support under contract number NSF CCR 96-23867, DARPA Air Force grant under contract number F30602-97-2-0121, and NASA under grant number NASA NAG 2-1250.

Finally, I am forever indebted to my parents for their love and support when it was most required.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

Introduction

1.1 Background

Multicasting provides an efficient way of disseminating data from a sender to a group of receivers. Instead of sending a separate copy of the data to each individual receiver, the sender just sends a single copy to a gateway who is responsible for multicasting it to the group of receivers.

The availability of multimedia contents in digital form, as well as the growth in popularity of communications channels such as the worldwide web, have brought a number of security issues to the forefront. This includes copyrighting, ownership assertion, authentication, access control, and secure communications. The importance of these issues has promoted research and innovative applications of secure technologies such as security protocols, cryptography, digital signature, digital watermarking, time stamping, smart card technology, data hiding, forgery detection, and others.

1.2 Problem Description

There have been many efforts to look into and solve these issues. One of the more interesting problems which is a blend of the above mentioned issues is to have a secure multicast protocol for video data involving copyright protection. People have done research on this topic, they have proposed solutions, written papers, but there has not been a single concrete effort to design and implement such a system.

There have been many proposed and successfully implemented solutions to the general problem of multicasting. But, when security is also introduced in the multicast, the problem becomes more difficult to solve. And when one is talking about multicasting video data and that too in real-time, the problem becomes a lot more complicated. This is because, firstly, the video data is a large amount of data, and secondly, since it is being recorded and displayed in real-time, the operations have to be very fast. The standard known encryption schemes like DES or RSA are very effective in terms of security, but they are slow to implement and are not very feasible in a real-time application. To achieve a net frame rate of about 30 frames per second, the encryption (along with other house-keeping operations) has to be performed 30 times per second, and so we need a very fast scheme, even if we need to compromise a bit on security.

1.3 Solution

Our aim in writing this thesis is to design and implement a system which supports a secure multicast protocol to transfer video data over the network, to check its feasibility in real-time and to evaluate it against various performance metrics. The applications that we are focussing at are video-on-demand (e.g., pay-per-view) in which the video is pre-recorded, and video-conferencing (e.g., live video transmission) where the video is recorded in real-time.

To achieve the desired frame rate, we have devised and implemented our own real-time encryption scheme called the permutation encryption which is very much suitable for binary data like jpeg frames. We are compromising a bit on the security front by using this scheme instead of using the standard DES; but by changing the keys after about every 30 seconds of video, we take care of this problem too.

1.4 Thesis Outline

The outline of the thesis is as follows: Chapter 2 describes related work in the area. Chapter 3 describes in detail the architecture and design of the Secure Multicast protocol. Chapter 4 deals with the implementation details and discusses all the data structures, frame formats and other issues that came up while implementing the system. Chapter 5 lays out the experimental testbed and scenarios. It also presents and analyzes the experimental results for video-on-demand and video-conferencing. Chapter 6 concludes with lessons learned and directions for future work.

CHAPTER 2

Related Work

In this chapter, we present a short account of the different multicast protocols and architectures in use. We also discuss various encryption schemes and their use while guarding attacks against security in a communication system.

2.1 Reliable Multicast Protocols

Reliable Multicast holds great promise for bulk data transfer applications, such as video data transfer in pay-per-view. Multicasting over a network allows a sender to distribute data to all interested parties while minimizing the use of network resources. Another factor that is of concern to us, besides reliability, is scalability of the Multicast Protocol.

2.1.1 TRAM: A Tree-based Reliable Multicast Protocol

Designed at Sun Microsystems Laboratories TRAM [Chi98] offers both reliability and scalability by using selective acknowledgement mechanism and adopting a hierarchial tree-based repair mechanism. The receivers and senders of a multicast session in TRAM interact with each other to dynamically form the repair groups. These repair groups are linked together in a hierarchial manner to form a tree with the sender at the root of the tree. Every repair group has a receiver that functions as a group leader; the rest function as the group members. These members are said to be affiliated with their leader. Except for the sender, every group leader is a part of some other repair group. All members receive data multicasted by the sender. The group members report lost and successfully messages to the group leader using selective acknowledgement mechanism. The repair leaders cache every message sent by the sender to the group and provide repair service for messages that are reported as lost by the members.

This protocol is suitable for applications that need to scale to large receiver populations, require real-time data delivery and high degree of reliability.

2.1.2 Scalable Reliable Multicast (SRM)

SRM [Flo96] uses global Negative Acknowledgements to report missing packets, and transmits repairs globally. Timer based random backoff mechanisms are used to suppress duplicate NACKs and repairs. Data is discarded from caches on a timer basis. SRM supports application-level framing which allows an additional reliability mechanism to be built into the applications. SRM has been deployed in shared dare applications with multiple senders, such as whiteboard application. Recent work has focused on increasing scalability by localizing session messages and error recovery.

2.1.3 Reliable Multicast Transport Protocol (RMTP)

RMTP [Pau96] [Pau98] uses a hierarchial tree-based approach to overcome ACK/NACK implosion at the sender. RMTP supports local repair via localized multicast or unicast, reducing the reach of unsolicited repairs. RMTP's tree formation relies on the static configuration. Able repair nodes are designated in advance, then selected by the rest of the members. RMTP is targeted at bulk and live data applications.

2.2 Multicast Routing Architectures and Security Issues

2.2.1 Core Based Tree (CBT) Architecture

CBT [Bal97] builds a single delivery tree per group which is shared by all of the group senders and receivers. The primary advantage of the shared tree approach is that it offers more favorable scaling characteristics than other multicast algorithms. The CBT protocol is a network layer multicast routing protocol that builds and maintains a shared delivery tree for a multicast group. The sending and receiving of multicast data by hosts on a subnetwork conforms to a traditional IP multicast service model. CBT achieves most of its design objectives including scalability, robustness, simplicity and interoperability with other multicast protocols.

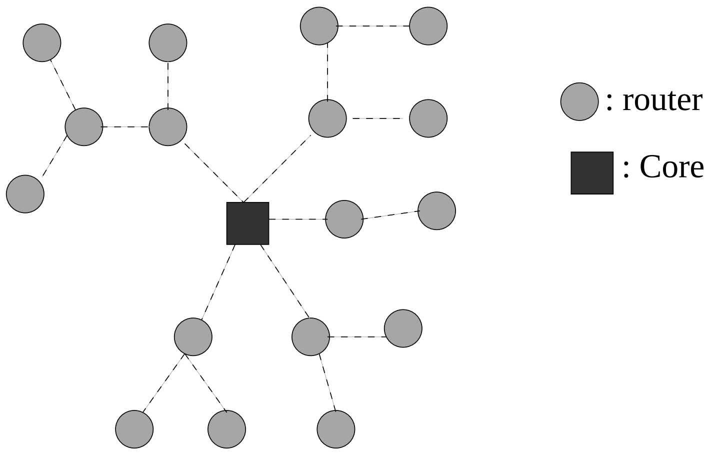

Figure 2.1 A CBT Tree

2.3 Encryption Schemes

Encryption is the transformation of data into a form that is almost impossible to read without the appropriate knowledge (a key in our case). Its purpose is to ensure privacy by keeping information hidden from anyone for whom it is not intended, even those who have access to the encrypted data. Decryption is the reverse of encryption; it is the transformation of encrypted data back into an intelligible form.

An encryption system is designed so that the process of converting the encrypted information is allowed only under two conditions. The first of these conditions is usually that the person attempting to decrypt the information must have the encryption system, which in modern times is likely to be a specially designed computer program. The other condition is that the person must also have a piece of information, called the encryption key. This is a piece of information that will be presented to the encryption system when the information is being encrypted and decrypted. If a piece of information has been encrypted with one key and a person attempts to decrypt the information with another key, the encryption system will not be able to make sense of the information. (Although this is not strictly true as we shall soon see.) Therefore, the output of the encryption system will be meaningless.

There are several different kinds of encryption systems available today [enc]. Most of these different systems generally fall into one of the categories listed below:

- Block Ciphers
- Stream Ciphers
- Product Ciphers
- Public key encryption systems

There are two ways that encryption systems currently use encryption keys: symmetric key encryption systems and asymmetric key encryption systems. The above categories fall into those two groups.

1. Symmetric key encryption systems:
    - Block Ciphers
    - Stream Ciphers
    - Product Ciphers
    - DES - It is the most common symmetric encryption scheme.
2. Asymmetric key encryption systems:
    - Public-Key encryption systems, e.g. RSA.

2.3.1 Block Ciphers

A block cipher accepts a plain-text input block of a fixed length. This input block is then converted into a cipher-text output block also of a fixed length. The enciphering and deciphering processes are constructed so that every binary bit in the output block depends on every binary bit in the input block and every binary bit in the key block. The size of the input and output blocks depends on the strength of the encryption required. A small block size will allow a feasible brute force attack, whereas a large block size would make a brute force attack unfeasible.

The strength of a block cipher can be improved by using a process, called chaining. Chaining is a process used during the encryption of plain-text. This process makes the cipher-text not only dependent on the current input block and key block but the previous input block or output block. This improves the strength of the block cipher encryption system, because if you have two identical input blocks, then the cipher-text will not repeat (as it would if you were not using chaining).[enc]

2.3.2 Stream Ciphers

A stream cipher uses a device, called a bit-stream generator, to produce a stream of binary digits. This stream of bits is called a cryptographic bit-stream. The cryptographic bit-stream is combined with the plain-text to produce the cipher text. The algorithm can be used in reverse, combining the cryptographic bit-stream to the cipher-text to produce the plain-text. The way in which the cryptographic bit-stream and either the plain-text or cipher-text is combined is through the use of either the exclusive-OR (This is an elementary binary logical operator.) (XOR) function or using modulo-2 addition.

If the bit-stream generator was completely random, the stream cipher would be an unbreakable encryption system. This would entail securely transporting the generated cryptographic bit-stream to the recipient. This is similar to most encryption systems and can compromise the security of the cipher. Therefore, most stream ciphers use an algorithmic bit-stream generator. Another problem of having a totally random bit stream and having to transport it around is the size of the bit stream. The bit-stream has to be the same size as the encrypted data. Therefore, if a large amount of data is being transported around, the resources needed to provide a secure transport for it can prove impractical to implement.

Each bit in the output cipher-text is dependent on the corresponding bit in the input plain-text, not on any other bits in the cipher. This is different to the block cipher which has a complex dependency upon different bits in the plain-text and the cipher-text.

As with block ciphers you can also use chaining on stream ciphers. It does not improve the stream cipher to a great degree as it does in the block cipher. This is because the

XOR function already stops patterns repeating in the cipher-text. Chaining, however, does add a little more noise to the cipher-text, making it slightly harder to break if it is attacked.[enc]

2.3.3 Product Ciphers

A product cipher is a block cipher that iterates several weak operations such as substitution, transposition, addition, subtraction and linear transformation. Examples of modern product ciphers include: LUCIFER, DES, SP-Networks, LOKI, FEAL, PES and Khufu and Khafre. Feistel ciphers are a class of product cipher which operates on one half of the cipher-text each iteration, after which the cipher-text halves are swapped around. Examples of Feistel ciphers are DES, LOKI, LUCIFER and FEAL. The DES encryption system is discussed in detail later.[enc]

2.3.4 DES

DES is the United States Government's Data Encryption Standard. DES is a product cipher that operates on blocks of data which are 64-bits in size and uses a key which is 56 bits long (eight 7-bit characters). The DES encryption process works by transforming a 64 bit plain-text message into a 64-bit cipher-text message using a key that is 56 bits long. This is done by using an encryption process that has 16 different stages (rounds). Each stage is a separate product cipher in itself. A different subset of the key which is 48-bits long is used during every stage of the encryption. This would mean that stage one will use a subset of the main key that is 48-bits (key one) in length during its encryption of the plain-text block. Then the second stage will use another subset of the main key for its encryption stage - this will be key two. This will continue until all 16 stages have been completed. Therefore, all of the 64-bit cipher-text blocks will have the sixteen different subset keys controlling each stage of their encryption.

To decrypt the cipher-text back into the plain-text, the stages are run in reverse, with the keys being used in their correct stages. Therefore, key two will be used in stage two, key one will be used in stage one. Therefore, to decrypt a DES cipher-text block, you use the stages of encryption in reverse using each stage's corresponding key. $K_1...K_{16}$ are the subset keys. $S_1...S_{16}$ are the encryption stages. Therefore, to decrypt a block, the block (B) will be run through $S_{16}(K_{16})$, stage sixteen with key sixteen. Then, through $S_{15}(K_{15})$ and so on until all the stages have been completed. When the block has been decrypted, it will be stored and then the next block will be processed.

To use DES on a computer system is quite simple. This is because the DES encryption system can be easily encoded onto only one chip. Devices exist that can be attached to computer systems and networks that automatically encrypt and decrypt the information as it is sent and received through a network or through a modem along a normal telephone link. Each user's key is randomly selected and revealed only to those that are privileged to the information. The chances of discovering the key is very remote because, with DES having a key of 56-bits in length, there are over 70 quadrillion possible key combinations.

However, some cryptography experts have challenged the strength and security of the DES algorithm. They say that it is too vulnerable if it is attacked by very high-powered decoding methods and parallel systems.[enc]

2.3.5 Triple DES

For some time it has been common practice to protect information with triple-DES [tri] instead of DES. This means that the input data is, in effect, encrypted three times. One way of doing this is to define triple-DES encryption with keys $k1$, $k2$, $k3$ as

C $= E_{k3}(D_{k2}(E_{k1}(M)))$,

where E_k and D_k denote DES encryption and DES decryption, respectively, with the key k. This mode of encryption is sometimes referred to as DES-EDE. Another variant is DES-EEE, which consists of three consecutive encryptions.

2.3.6 RSA

RSA [RSAb] is a public-key cryptosystem. This means that one publishes part of the encryption key to allow people to encrypt a message. The rest of the key is kept secret and it is used to decrypt messages. Given the public encrypting key, it is not possible to decrypt the messages. This solves what was known as the Key Distribution Problem. Before public key cryptogrphy, people who wanted to communicate using encryption, needed to share a single secret key, which both encrypted and decrypted the message. The problem was that these keys needed to be frequently changed in case one had become compromised. The cost of distributing keys was very expensive. With public key cryptography, one can distribute the key to as many people as possible without compromising security.

Another use for public keys is in the authentication of messages. It takes advantage of the fact that if the private decryption key is used to encrypt the message, the encryption key can be used to decrypt it. This enables the sender to prove that they did in fact write the message (as no one else should have access to the private key).

To generate a key pair, a user must first choose two large prime numbers. Let's call these primes p and q. Then the user must calculate $n = p * q$. This is the public modulus and is part of the published public key. [RSAa]

The next step is to chose an encryption exponent, e, such that the greatest common divisor of e and $(p-1)*(q-1)$ is 1; i.e. that $gcd(e,(p-1)*(q-1)) = 1$. We now have a public key and can publish the public exponent and modulus (e, n) so that people can send encrypted messages.

Finding the decryption exponent, d, is trivial ONLY if one knows the values of p and q. We calculate it so that $d.e = 1(mod(p-1)*(q-1))$. This, along with the modulus n must be kept secret.

Encryption: Assume that the message, M, can be written as a number or list of numbers less than the modulus, n. To compute C, the ciphertext, for M, we raise M to the power of e modulus n.

Decryption: The reverse of this process is similar. We raise the ciphertext, C, to the power of d modulus n, to get the original message M.

2.4 Communications Security using Encryption

In this section, we will discuss methods in which a conventional encryption system such as DES can be incorporated into a data processing network. From a security point of view, end-to-end encryption, which is where the data is encrypted at its point of origin and only decrypted at its final destination, is considered the most secure type of encryption across a communications system. We will give a quick overview of the types of networks and the various ways in which encryption can be implemented.

The main types of networks are distributed data processing networks; they are the most common in use. This is where a main computer (the host) can send tasks away to other computers on the network so that they can perform the task while the host processor performs other tasks. The devices on a data processing network can be computers, printers, satellites or other electronic devices (called nodes). All these devices are linked to each other by telephone lines and microwave links. A network is usually described by the pattern its links and devices make. The different types of link are, Star, Loop, Tree, Bus or Mesh systems. There are also many other types. The main reason for a network is so that a user or a program can send data from one location to another, in the same building or across the world. A network architecture should be able to provide the ability to implement security measures if they are required.

2.4.1 Network Encryption Types

There are three ways to implement encryption systems into a network communications system. They are:

1. **Link Encryption:** It protects data between adjacent network nodes. The encryption system's devices are situated at each end of the communications line between two network nodes. The devices are equipped with identical keys. [enc]

2. **Node Encryption:** It is similar to Link encryption except that each pair of nodes share a key to protect the data. However, data passing through the intermediate node is not plain-text as it would be in Link encryption. At an intermediate node the cipher-text is transformed from being enciphered under one key to another key. This means that the information is decrypted when it is received (by a security module), but before it even passes into the node, it is re-encrypted with a different key, and only then it is sent into the node. [enc]

3. **End-To-End Encryption:** This form of encryption continually protects the data during transmission through the intermediate nodes. The plain-text would be converted to cipher-text at the origin node and then transmitted thorough the network to its destination. As it passes through each node it will not be altered until it reaches its final destination, where only there will it be converted back into plain-text. [enc]

2.4.2 Kerberos - A Network Authentication System

Kerberos [Tun] is a security product - specifically, an authentication system-that verifies who you are. Authentication is used for a wide range of security services: authorization, privacy, auditing, and so forth. Kerberos was developed as open software. That is, its development was open to review, and many individuals contributed to it. Open development has ensured that a wide variety of viewpoints and needs are addressed in Kerberos.

CHAPTER 3

Architecture and Design of the Secure Multicast Protocol

We shall now discuss the architecture of the protocol and the design of our implementation in details. The protocol tries to simulate a secure multicast of video data over TCP links between random nodes located anywhere.

3.1 Architecture

The two most important nodes in the basic architecture of a secure multicast protocol might be a gateway (used for multicast) and a group leader (for security and key management). Once the multicast group is setup and working, with these two nodes; any number of members can join and leave the group at any time. This is exactly what is being done in our protocol. So, from a broader viewpoint, the architecture of the group might look like figure 3.1. Let us discuss each of the three nodes shown in the figure in detail.

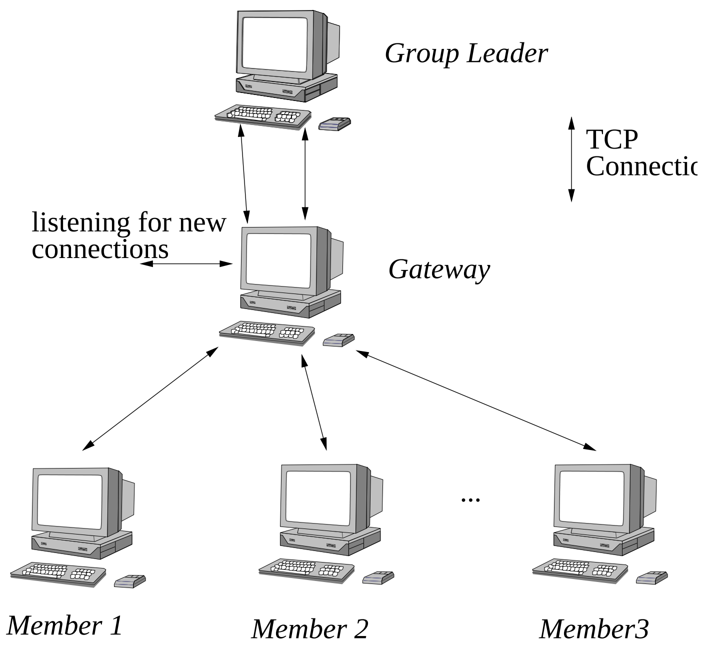

Figure 3.1 Architecture of the multicast protocol

3.1.1 Gateway

The gateway is the central node of the system. As can be seen, every node has a connection with the gateway only. So, all a node needs to know to join the group is the location of the gateway. The gateway is responsible for multicasting the data to the group. The sender sends the packets to the gateway and it forwards them to the rest of the members. This puts a lot of load on the gateway. It has to keep polling on all the sockets (with the members) and as soon as it finds data on one of the sockets, it multicasts it to the rest of the members. Another task of the gateway is to look for connections from new members and initiate the JOIN protocol with the group leader. So, whenever a new member connects to the gateway, the gateway initiates the JOIN protocol with the group leader, gets the response back, and depending on the response, lets the new member join the group. At the same time, it also keeps track of any member leaving the group, and inform the group accordingly.

3.1.2 Group Leader

The group leader is the most important entity from security's point of view, as it is responsible for authentication and key management. Whenever a new member wants to join the group, the group leader authenticates it by checking its ACL (Authentication Control List). Once the new member is verified, the group leader executes the JOIN protocol (initiated by the gateway) by sending the relevant information about the group to the new member (via the gateway). The group leader also keeps looking for KEY INFO packet from the sender. This packet contains the keys with which the data is encrypted along with the range of the frame numbers for which they are used. The keys are encrypted by the sender, using the public key of the group leader. Once received such a packet, the group leader decrypts the keys using its private key, and then generates the corresponding VERIFICATION PACKET, which contains these keys encrypted with the public keys of the members (so that they can go ahead and decrypt the keys using their own private keys).

3.1.3 Member

The member node is a normal member of the group and can JOIN/LEAVE at any time and SEND/RECEIVE data to/from the group. The member needs to know the location of the gateway in order to join the group. After it connects to the gateway, the gateway confirms from the group leader if the member can be a part of the group or not (by initiating the JOIN protocol). If verified, the member ends the JOIN protocol by sending its information to the group (via gateway). After joining successfully, the member enters the sending/receiving phase. It can either initiate a send session in which it can multicast either a pre-recorded video file (video-on-demand) or it can transmit live video in real-time, or it can become a receiver and receive video from another sender and display it on its screen in real-time.

3.1.4 Multicast Protocol

The data transmission in the multicast protocol can be divided into three phases: (1) the key selection and multicast phase, when the sender selects a key at random and the range of frames for which the key will be used and then multicasts it to the group, (2) the sending phase when the sender multicasts an encrypted data message, and (3) the receiving phase when the receivers receive both the keys and the data and decrypt the data. Refer to figure 3.2 to understand it better. These phases will be discussed in details in the following sections.

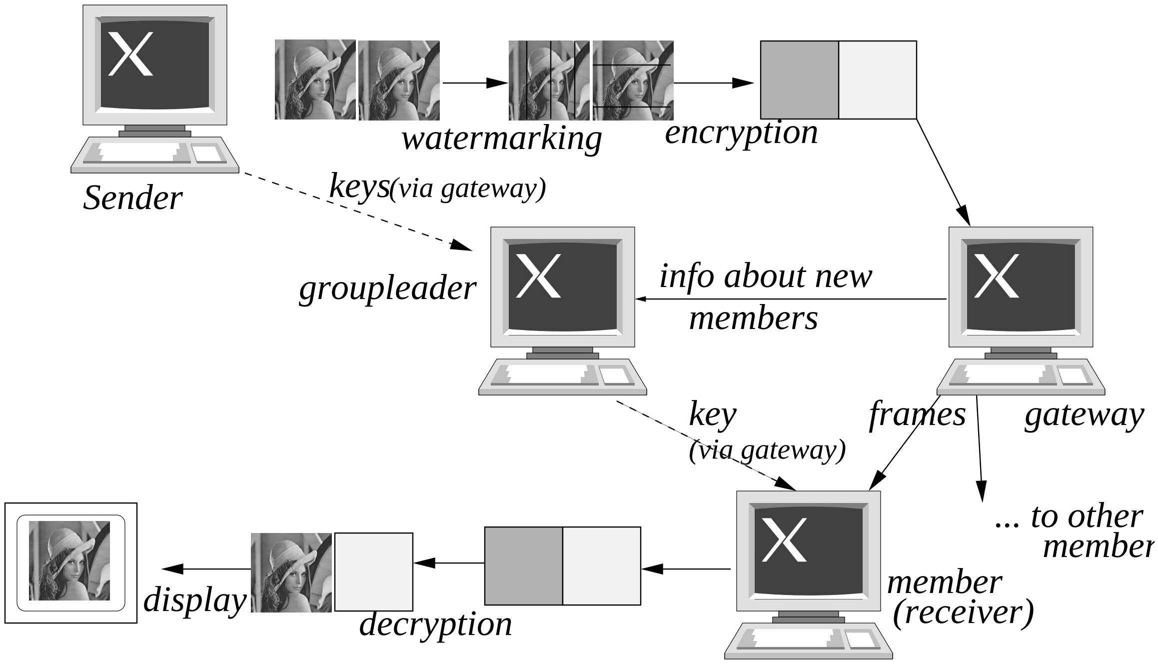

Figure 3.2 The Multicast Protocol

3.1.5 Encryption Schemes

There are two encryption schemes used in the implementation. To encrypt the multimedia data, a symmetric encryption algorithm needs to be used. We have used our own algorithm called the *permutationencryption* for the purpose. This algorithm will be discussed in detail in the later sections. To encrypt the keys used in permutation encryption, an asymmetric encryption algorithm needs to be used. We have used the standard RSA algorithm for the purpose.

3.2 Design

In this section, we will describe the actual JOIN, LEAVE, SEND, RECEIVE protocols and the design by which we went on to implement these protocols.

3.2.1 Initialization/Setup of the group

The first node to come up in the group is the central node, namely, gateway. The gateway opens up a socket on a well-known port number and listens for a connection from the group leader. It assumes that the first connection request it gets, will be from the group leader. The next node to come alive is the group leader. The group leader connects to the gateway on the well known port number (it should be knowing the IP address of the gateway) and the initialization sequence starts. The gateway accepts the connection from the group leader and stores the socket in a list. (The gateway maintains a list of all the sockets for all the members that have connected so far. We shall be discussing this in details later). Then the group leader again connects to the gateway, and the gateway accepts another connection on a new socket. We call this the verification socket. This socket is used to verify the identity of a new member and no real data transfer takes place on this link. So, there are two TCP links between the gateway and the group leader. One is for data and one is for verification of the new members. The group is now ready to accept connections from new members. The gateway spawns a new thread to keep listening on the well-known port number for any new connections, while the main thread starts polling the list of the sockets for previous members for any available data that needs to be multicasted to the group. The group leader, in the meantime, generates its public and private keys, and then spawns a new thread to take care of the connections from new members. The main thread listens for any data on the socket connected to the gateway and if the data is found, it acts accordingly.

For a flowchart representing The Initialization Protocol refer to figure 3.3.

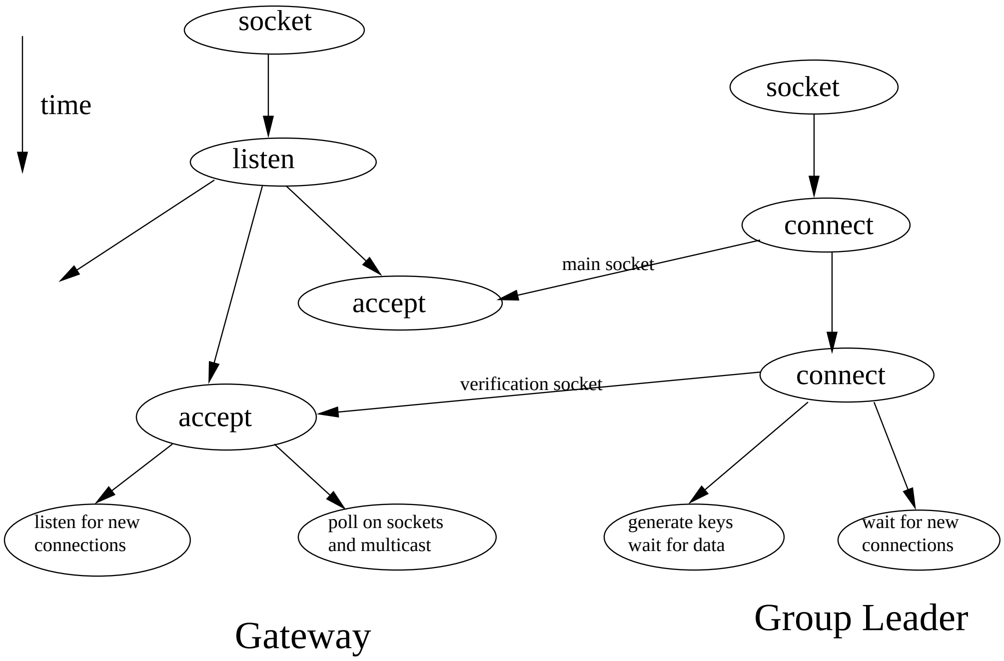

Figure 3.3 Initialization Protocol

3.2.2 JOIN protocol

Whenever a member wants to join the multicast group, it has to initiate the JOIN protocol doing a *connect* to the gateway which is continued further by the gateway and then by the group leader. The member first sends a TCP connect request to the gateway. The gateway then sends the IP address of the new member to the group leader for verification on the verification socket. The group leader checks its Access Control List (ACL) to verify if the new member can join the group or not (e.g. by checking if he/she has paid the member fee etc.) If the member is not verified, the group leader simply sends a one byte packet, indicating the negative response. The gateway, on receiving the negative response, notifies the member and closes the connection. But, if the member is verified, the gateway prepares a packet containing the information about the composition of the group so far. This information mainly contains the number of members in the group and the public keys (along with key-lengths) of all the members so far in the group. The first key in the list corresponds to the group leader and the following keys belong to the normal members. The gateway forwards this packet to the new member as a response to the connect request and adds the new member to its list of existing members by adding its socket to the list of sockets. The member ends the JOIN protocol by preparing a JOIN INFO packet which contains its own information (its public key) and sends it to the gateway. The gateway, in turn, multicasts it to the group and the group becomes aware of the joining of the new member. After the member sends the JOIN packet, it spawns a new thread to take care of any incoming packets. The main thread prompts the user to either start the SEND protocol or to LEAVE the group.

For a flowchart representing The Join Protocol refer to figure 3.4.

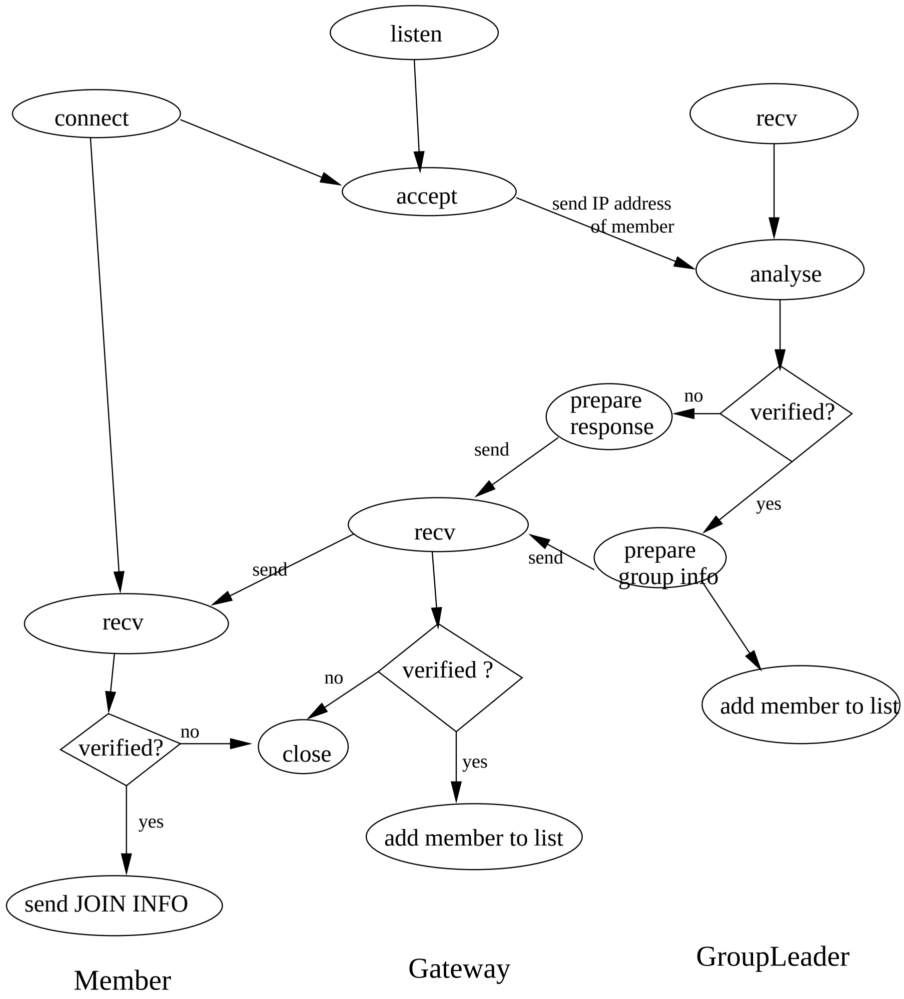

Figure 3.4 Join Protocol

3.2.3 LEAVE protocol

To leave the group, all a member needs to do is to close the connection with the gateway. The gateway, in response, prepares a LEAVE packet, which contains the IP address of the member, and multicasts it the rest of the group. The group members update their data structures accordingly.

For a flowchart representing The Leave Protocol refer to figure 3.5.

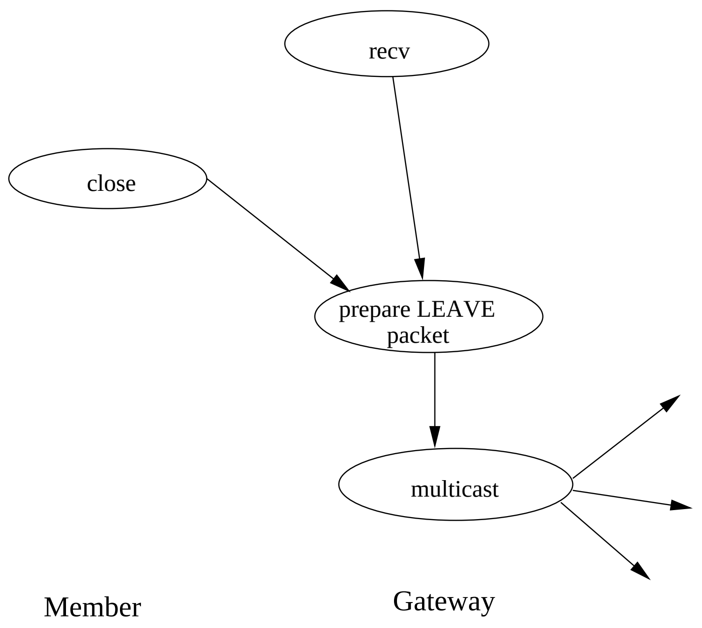

Figure 3.5 Leave Protocol

3.2.4 SEND protocol

This is the most complex of all the protocols. A member can either send a pre-recorded video file (video-on-demand), or it can transmit live-video (video-conferencing). In either case, the SEND protocol remains the same. The member starts by first sending a SEND INFO PACKET, which contains the number of frames to be sent, the number of frames after which the key will be changed, and the file name (in case of video-on-demand). The group members, after receiving this packet, prepare themselves for receiving video and then displaying it on their screen. The group leader prepares the bit string B_{uid}s array for each member in the group. These arrays decide which keys are sent to which member to ensure copyright protection. Then it prepares a KEY PACKET which contains the two keys with which the frames will be encrypted and the range of frames for which those keys will be used. The two keys are encrypted using the public key of the group leader, so that only the group leader can access them. It then multicasts this packet to the group via the gateway. The group leader, on receiving this packet, first decrypts the keys using its private key, and then prepares a VERIFICATION packet which contains these keys encrypted with the public keys of all the members. It then multicasts this packet to the group. Now, the member can start sending the data. In case of video-on-demand, the member opens the corresponding file to be sent in read mode and extracts the data frame by frame, and in case of video-conferencing, it initializes the camera and starts recording video frame by frame. In either case, in one iteration, it gets one jpeg frame to be sent. Once it gets the frame (either from the file or from the camera), it watermarks it using two different patterns (for copyright protection), and then encrypts the two watermarked frames using the keys that were sent in the KEY packet. It then prepares a DATA packet in which it stores these two frames back to back and multicasts it to the group. This recording, preparation, and sending of the frames continue till half the number of the frames after which the key was to be changed have been sent. Then, it selects two more random keys with which the data will be encrypted for the next set of range of frames, prepares a KEY packet and multicasts it to the group. This sending of the KEY packet is in parallel witht he sending of the data frames as a separate thread is created for the purpose. (For example, if the number of frames after which the key is to be changed is 1000, then after the first 500 frames are sent, the sender will send a KEY packet containing the keys for frames 1000-1999). This keeps on repeating till all the frames have been sent.

For a flowchart representing The Send Protocol refer to figure 3.6.

3.2.5 RECEIVE protocol

All the other members in the group except the sender act as receivers. They are on the look-out for KEY and DATA packets. Whenever a receiver receives a KEY packet, it scrolls through it and gets its key from its respective slot. It also stores the range of frames for which the key is valid. Then, when it receives the DATA packet, it checks the frame number, applies the appropriate key to decrypt the data and then displays the frame on the screen.

For a flowchart representing The RECEIVE Protocol refer to figure 3.7.

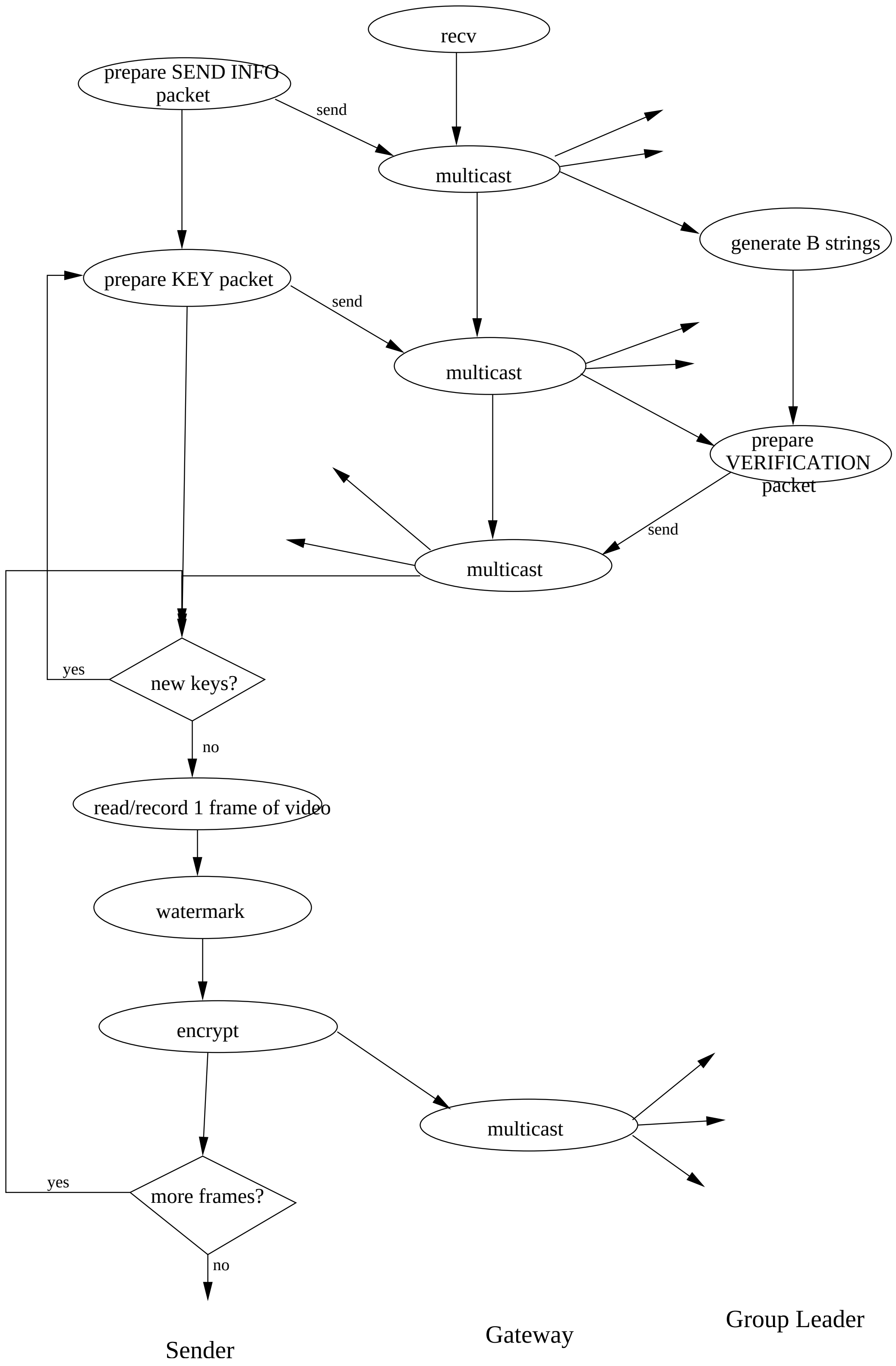

Figure 3.6 Send Protocol

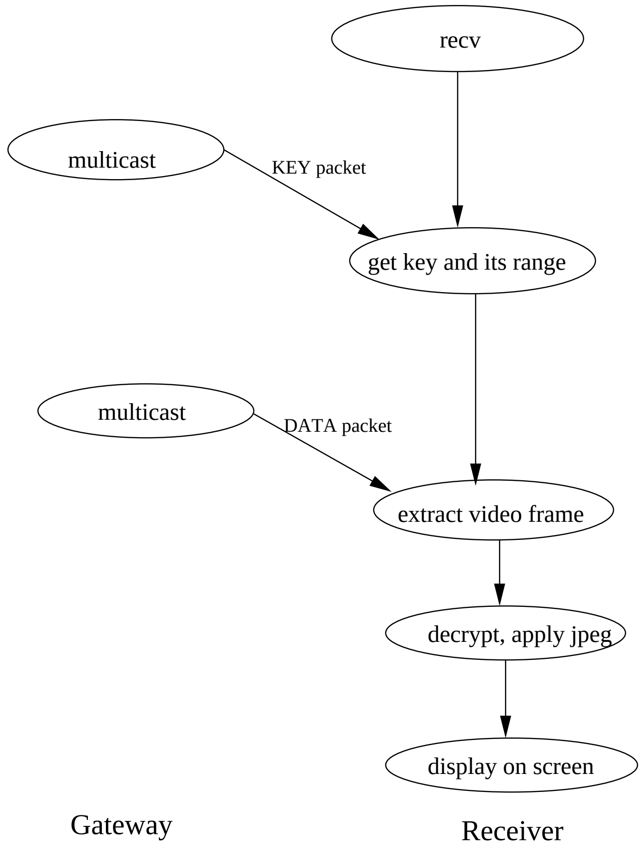

Figure 3.7 Receive Protocol

CHAPTER 4

Implementation

In this chapter, we discuss the complete implementation details like the data structures, the frame formats, the organization of the code and other problems that we faced during the implementation along with their solutions. The protocol was implemented in C on the Sun Solaris (OS version 5.7) platform.

4.1 Permutation Encryption

Before we describe the actual implementation of the protocol, we present here in detail, the symmetric encryption algorithm that we have used to encrypt the data frames. Instead of actually encrypting the data with, say, DES, i.e., actually substituting the data with cipher text, the algorithm simply transposes the data or permutes/reorganizes the data. The simple permutation of the data is much faster as compared to changing the data into cipher text since it involves much simple computation. DES is a very expensive operation and is not feasible for such an application which involves performing the operation some 30 times in one second. This permutation takes place according to a key, and exactly the same key is required to get back the original data.

4.1.1 Concept of Interleaved Memory

Interleaving is a design technique used to increase memory bandwidth. Memory interleaving increases bandwidth by allowing simultaneous access to more than one chunk of memory. This improves performance because the processor can transfer more information to/from memory in the same amount of time, and helps alleviate the processor-memory bottleneck that is a major limiting factor in overall performance.

Interleaving works by dividing the system memory into multiple blocks. The most common numbers are two or four, called two-way or four-way interleaving, respectively. In order to get the best performance from this type of memory system, consecutive memory addresses are spread over the different blocks of memory. In other words, if you have 4 blocks of interleaved memory, the system doesn't fill the first block, and then the second and so on. It uses all 4 blocks, spreading the memory around so that the interleaving can be exploited.

We shall use the same concept to disperse or permute the data around.

4.1.2 The Algorithm

The data is divided into blocks of different sizes. The number of blocks and the size of each block depend on the key. Let us explain this with a simple example.

Lets assume that the size of the data is 5000 bytes and the key with which it is to be encrypted is 5603417982.

We first find the sum of the digits of the key. In this case it is 45. The total number of digits (including 0) is 10.

So, we shall divide the data into 10 blocks. The sizes of the blocks will depend on the face value of each digit and it will be a percentage of the total file size. Hence the block sizes will be the following:

Block 1: (5/45) * 5000 = 555 B

Block 2: (6/45) * 5000 = 666 B

Block 3: (0/45) * 5000 = 0 B (it exists only logically)

...

Block 9: (8/45) * 5000 = 888 B

Block 10: (2/45) * 5000 = 222 B

This will total to 4995 B since the sizes were truncated to the nearest whole number. It leaves an eleventh block of size 5.

Once we have the block sizes, we start permutating the data into blocks (using the interleaved memory concept) according to the following rule:

```
The next byte of the original data goes into the first available
slot of the next block in the encrypted data.
```

Refer to figure 4.1 to make it more clear.

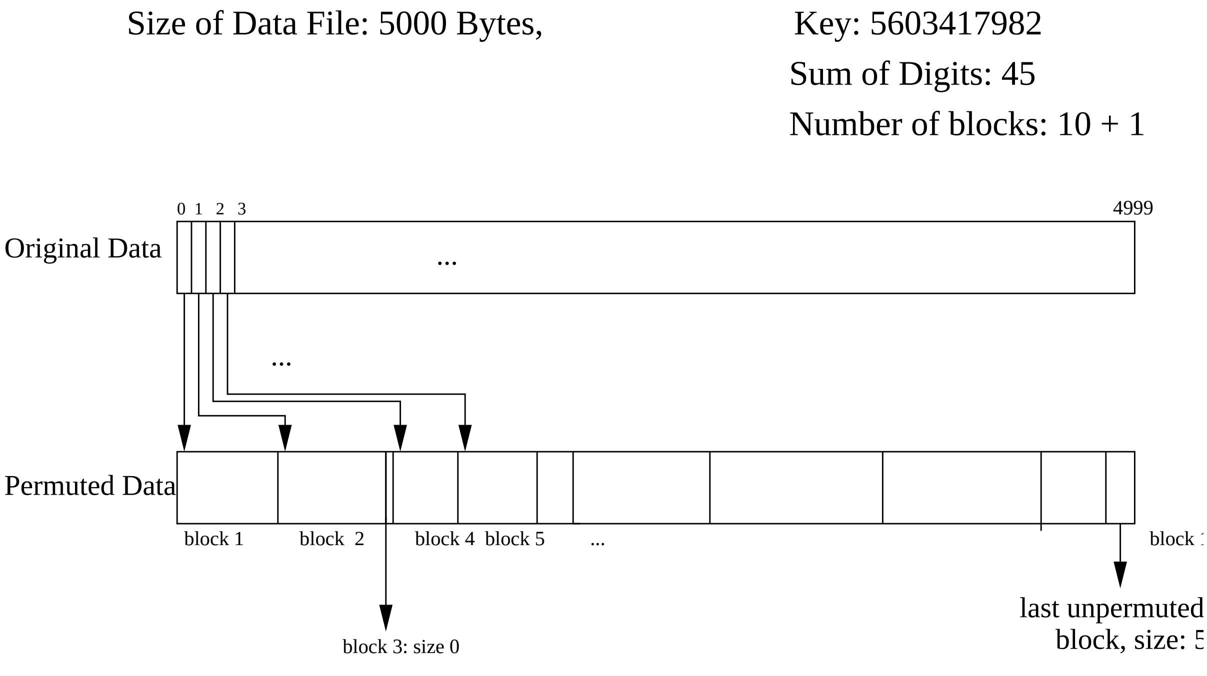

Figure 4.1 Permutation Encryption

We maintain a pointer for each block which keeps track of the number of slots already used and the number of slots available in each block. By sequentially traversing through the original data, we pick up each successive byte and place it in the encrypted data by following the above rule. The last block of a few bytes (it can be proved that the size of this block will be always less than the number of digits in the key, 10 in this case) is copied at the end of the encrypted data without permuting. After this is done, we get the encrypted data which has the same size as the original data, but is permuted according to the key and exactly the same key is required to put it back in original form. We shall discuss the advantages and limitations of this encryption scheme in the later chapters.

4.2 Data Structures

Before actually looking at the implementation details, let us see and understand the data structures that are used at the three nodes. We will also discuss the options that we had and why we chose these out of them.

4.2.1 Gateway

Since the gateway is the central node and every other node in the system has connections to it, it needs to maintain the information of all the other nodes in the system. The information mainly comprises of the IP addresses of each node and the sockets with which they are connected with the gateway.

We maintain a linked list for the purpose which has the following structure:

```
typedef struct memberQueue {
    int sockFd;     // the socket with which the member is connected
                    // with the gateway
```

```
    struct sockaddr_in addr;  // IP address of the member
    struct memberQueue *next;  // pointer to the next node in the list
}memberQueueRec, *memberQueuePtr;
```

There are two main reasons for choosing a linked list:

1. We shall always be accessing the elements of the list in a sequential order. We will access them once to poll at the sockets for any incoming data, and then to multicast the data to the remaining sockets.

2. New members can join and old members can leave at any time. The insertion and deletion of a node are relatively easy with linked lists.

We also have the following two global variables:

```
memberQueuePtr memberQueueHead; // it always points to the first
                                // node in the member queue


unsigned int memberCount;     // keeps a count of the number of
                              // members in the system so far
```

Then, since a new thread is spawned to take care of the connections from the new members, a structure needs to be initialized to be passed on to the child thread. All that structure needs to know is the sockets with which the gateway is connected to the new member and to the group leader. So, it looks like this:

```
typedef struct threadInfo {
        int sockFd;
        int verificationSockFd;
}threadInfoRec, *threadInfoPtr;
```

We have tried to keep the implementation as simple as possible because the system is to be used in real-time. So, it should be as efficient as possible and therefore, all the data structures are small and to-the-point.

4.2.2 Group Leader

Since the group leader needs to do the key management, it has to maintain a lot more information about the members in the group. This information includes the member Id of each member with which it is uniquely identified, the public keys of all the members, and some other miscellaneous information.

The group leader also maintains a linked list of members whose structure is like this:

```
typedef struct memberQueue{
    unsigned char memberId[5];
    char fileName[30];
    unsigned char *B;
    unsigned int framesKeyChange;
    struct memberQueue *next;
}memberQueueRec, *memberQueuePtr;
```

The $memberId$ is an array which contains the unique member id of the member concerned. $fileName$ stores the name of the file in which the public key of the member is stored. B is a special array which is very important for copyright protection. The group leader generates a bit stream of 0's and 1's for each member in the group. These bits correspond to each frame sent by the sender and therefore, the length of this bit stream is equal to the number of frames being sent by the sender in the current session. The B array contains this bit stream packed into a byte stream. $framesKeyChange$

is an integer denoting the number of frames after which the sender will change the keys during the current session.

There were many options to select a unique member id. One way was to select the IP address as the member id. But in the dot notation, it takes 15 characters to represent an IP address. Another way was to enumerate the members starting from 1. But then it will break the order whenever a member leaves the group. Also, each member will have to keep this extra information about every other member. We adopted a different system to assign unique member ids. This takes the IP address of each member to calculate the member id but at the same time uses only 4 bytes to represent it. We extract the 4 network bytes from the IP address of a member and store them in a 5 byte unsigned char array. The last byte of this array is a null character so that it can also be used as a string and a null indicates the end of string. So, by using only 5 characters, we can uniquely identify a member.

The same concept is used to assign a unique filename for the public key of the members. The member id is first converted into a long integer by using the following formula:

```
int = num[0]*256*256*256 + num[1]*256*256 + num[2]*256 + num[3]
```

and then it is converted to an ascii string which becomes the public key filename for that member. This filename will be unique for every member.

The child thread at the group leader takes care of the joining/verification of new members. So all that needs to be passed to the child is the verification socket with which it is connected to the gateway.

```
typedef struct _thread_Info {
        int verificationSockFd;
} threadInfoRec, *threadInfoPtr;
```

It may be noted here that the linked list of members is global to the child thread by the declaration of the global variable:

```
memberQueuePtr memberQueueHead;
```

This variable always points to the first node in the linked list.

4.2.3 Member

The member also maintains a linked list of nodes, the nodes representing the other members in the group. The first node always represents the group leader. The structure of such a node is as follows:

```
typedef struct _member_Queue {
    unsigned char memberId[5];
    char fileName[30]; // public key
    FILE *fp; // file to store the data
    VideoParams2Ptr vp;
    unsigned int numFrames;
    struct _member_Queue *next;
}memberQueueRec, *memberQueuePtr;
```

The *memberId* is the same 5 byte character array to store the unique member id formed by taking the IP address of the member. The *fileName* stores the name of the file in which the public key of the member is stored. Whenever that member becomes the sender of video data, the original member needs to store some video parameters which are used to display the video on the screen. The pointer *vp* takes care of storing that information. The integer *numFrames* stores how many frames the video will comprise of. Sometimes, the member will like to store the video instead of simply displaying it on the screen. The file pointer fp is used for that purpose and it points to a data file on the disk in which the video data will be written.

Again the global variable

```
memberQueuePtr memberQueueHead;
```

makes this linked list global to all the threads.

The child thread which is created initially to take care of the incoming packets from the gateway needs to know the socket with which it is connected to the gateway. So, the structure, which is passed to the thread, looks like this:

```
typedef struct _thread_Info {
        int sockFd;
}threadInfoRec, *threadInfoPtr;
```

There are two more threads, which get created and destroyed in the scope of a member's lifetime. One is the $sendKeyPacketThread$. This thread takes care of preparing the key packet to be sent in the middle of the transmission of the data. It might be recalled from the design above that new keys are generated and sent to the group leader in the middle of the range of the old keys. Since it will slow down the process of transmission of video, so a new thread is spawned for the purpose. The structure that is passed to this thread has the following data:

```
typedef struct _key_thread_Info {
        unsigned int startRange;
        unsigned int framesKeyChange;
        unsigned char *nextKey0Str;
        unsigned char *nextKey1Str;
        int sockFd;
```

```
}keyThreadInfoRec, *keyThreadInfoPtr;
```

The range, for which the new keys are valid, can be easily calculated using *startRange* and *framesKeyChange*. *startRange* denotes the start range of the new keys i.e., the starting frame number after which the keys are valid and *framesKeyChange* is the number of frames starting from *startRange* after which the keys become invalid. The keys are stored in two character arrays *nextKey0Str* and *nextKey1Str* in ascii format. It also needs to know the socket with which it is connected to the gateway, i.e., the socket on which the key packet will be sent.

Another thread that gets created at the member node is the *verificationPacket* thread. This thread is created whenever a verification packet is received. This packet contains the keys for the next range of frames. The processing of this thread takes time since the member has to scroll through the slots to find the key meant for it, and so we create a separate thread to take care of this. The structure that gets passed tot his thread contains the following data:

```
typedef struct _verification_thread_Info {
        unsigned char memberId[5];
        unsigned char *buf;
        unsigned int bufSize;
}verificationThreadInfoRec, *verificationThreadInfoPtr;
```

memberId contains the 5 byte member id of the sender. *buf* is a character array which contains the bytes that form the verification packet, and *bufSize* is the number of bytes in the verification packet.

These were the data structures at the three nodes. Then there are some more data structures which are required for recording and playback of video. They are the following:

For recording, we need the following structure:

```
typedef struct _video_rec   {
    XilCis cis;
    XilImage o1k_image;
    XilImage scaled_image;
    XilSystemState state;
} VideoParams1Rec, *VideoParams1Ptr;
```

And for playback, the following structure is required:

```
typedef struct _video_play  {
    XilSystemState State;
    XilCis cis;
    XilLookup colorcube;
    XilDitherMask dmask;
    XilImage imageYCC;
    float scale[3], offset[3];
    XilImage displayimage;
    Atom closedownAtom;
    Display *display;
} VideoParams2Rec, *VideoParams2Ptr;
```

These structures are in accordance with the *xil* libraries from SUN Microsystems to interact with a camera attached to the solaris machines. For more information regarding this, please refer to these libraries.

4.3 Structure

Now, we shall discuss the actual structure of the implementation, as in how the code is organized. As was seen in the architecture above, there are three main nodes, namely the gateway, the group leader, and the member. On the same lines, we have three separate programs in our implementation each of them simulating one of the three nodes. Other than that, we have a video library supporting the *videoRecord* and *videoPlay* functions and a utility library which has a lot of functions that are common to all the three nodes. Let us see some of them in details:

1. The TCP uses a byte stream to transfer data from one end to the other. So, whatever form the data may be in, it has to be converted to character bytes to send it across the network. The most common data type in our implementation is int. It has to be sent across the network very often. We have written a few helper functions which take care of converting int and short into byte arrays and back:

   ```
   unsigned int charStarToUInt(unsigned char *number);

   unsigned int charStarToUShort(unsigned char *number);

   unsigned char * uIntToCharStar(unsigned int number);

   unsigned char * uShortToCharStar(unsigned short number);
   ```

2. Also, since TCP is a byte stream with no actual frame boundaries, the *send* and *recv* functions do not ensure that the transfer has actually taken place. They return the number of bytes that were actually written or read from the socket respectively. To overcome this problem and to ensure reliability, we have written our

wrapperSend and *wrapperRecv* functions which ensure that the specified number of bytes are written/read to/from the socket respectively:

```
int wrapperRecv(int sockfd, void *buf, unsigned int
bytesToBeRead);
```

```
int wrapperSend(int sockfd, const void *buf, unsigned int
bytesToBeWritten);
```

These functions return to the main program only when the specified number of bytes are transferred.

3. To overcome the framing problem, we are using length based framing. We have written our *Send* and *Recv* functions which send/receive the length of the packet before actually sending/receiving the data. Throughout the implementation, we are consistently using these functions to send/receive the data.

```
int Send(int sockfd, char *buf, unsigned int length);
```

```
unsigned char * Recv(int sockfd, unsigned int *length);
```

The *Send* function first sends the length of the data buffer as a 4 byte character array and then the actual data using *wrapperSend*. In the same way, *Recv* function first receives the length as a 4 byte character array, converts it into integer and then receive that number of bytes using *wrapperRecv*. This ensures that the exact number of bytes are transferred in one *Send* or *Recv* call.

4.3.1 Frame Formats

Various different types of frames are used to send different types of data across the network. Let us discuss each one of them:

1. The first frame to be used is the GROUP INFO frame, which the group leader sends to a new member of the group, telling it about the configuration of the group so far.

   The frame looks like this:

4	5	4	key 0 length	5	4	key 1 length	
member count	member Id	key 0 length	key 0	member Id	key 1 length	key 1	...

Figure 4.2 GROUP INFO Frame Format

   Since this frame is not multicasted to the group, but is only sent to the new member by the gateway, we do not need a frame type flag for this.

   As can be seen, the public keys of all the members are sent to the new member one after the other. To distinguish between them, the key lengths are also sent along with the id of each member.

2. The JOIN INFO packet, which is sent by the new member after it becomes a part of the group, looks like this:

1	5	4	key length
type	member Id	key length	key

Figure 4.3 JOIN INFO Frame Format

The new member just needs to send its public key to the group, and so it prepares a JOIN INFO packet in the above format and sends it to the gateway, which in turn, multicasts it to the group.

3. Whenever a member leaves the group by closing its connection with the gateway, the gateway prepares a LEAVE packet on its behalf and multicasts it to the group.

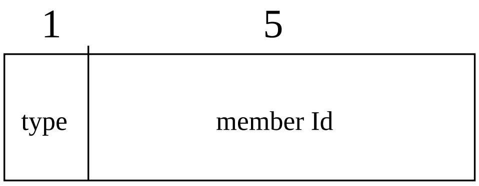

Figure 4.4 LEAVE Frame Format

All the LEAVE packet needs is the member id of the member that just left the group.

4. Next we discuss the DATA packet type. It contains the actual encrypted video data. Its format looks like this:

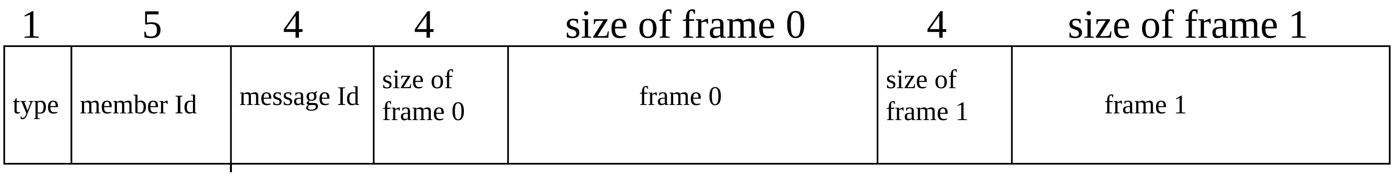

Figure 4.5 DATA Frame Format

As was previously described, two copies of the same frame are sent in one data packet. Each copy is watermarked in a different way. This is done to ensure copyright protection. These two copies are sent one after the other after being watermarked and encrypted. To distinguish between the boundaries, the frame sizes are also sent along with the frames. Message id is nothing but the frame number. It is used to check which key is to be used to decrypt the data at the receiver end.

5. The sender sends the keys with which the data is encrypted to the group leader before actually start sending the data, and then before the next range of frames is sent. It uses the following format for the KEY INFO packet:

1	5	4	4	key 0 size	4	key 1 size
type	member Id	start range	key 0 size	key 0	key 1 size	key 1

Figure 4.6 KEY INFO Frame Format

Along with the keys, the sender also needs to specify the range for which the keys are valid. For the purpose, the sends the start range of the keys. The number of frames after which the keys will be changed is already known to all the receivers. So, they know till when the keys are valid. The sender sends the two keys (for two different watermarked copies of the same frame) one after the other along with the key lengths. These keys are encrypted by the public key of the group leader, so that only the group leader can access them.

6. The group leader, on receiving a KEY INFO packet, prepares a VERIFICATION packet in response. This packet contains the same keys but encrypted using the public keys of each member. It is one of the most complex packet format. Refer to figure 4.7 for its format.

The group leader copies the member id and the start range from the KEY INFO packet. Then it stores the member count which makes it easier for the receivers to find their respective slot. Then it stores the keys for every member in the group along with their member id, and the size of the key. The receivers, on receiving

1	5	4	4	5	4	size of key
type	member Id	start range	member count	member Id	size of key	key
				member Id	size of key	key
				member Id	size of key	key
				...	...	...

Figure 4.7 VERIFICATION Frame Format

such a packet, scroll through it, till they find their own member id. They copy the encrypted key and then decrypt it using their private key. This key also contains the byte which is either 0 or 1, and indicates which frame they can decrypt using this key. This is done to ensure copyright protection.

7. Before the sender actually starts sending out data, it informs the group that it is starting a new send session and that they should get ready for it. For the purpose, it sends out a FILE INFO packet or LIVE VIDEO INFO packet depending upon whether it is a video-on-demand application or a video-conferencing application. The frame formats for the two figures are given in figures 4.8 and 4.9 respectively.

1	5	4	4	4	filename length
type	member Id	number of frames	number of frames, when key is changed	filename length	filename

Figure 4.8 FILE INFO Frame Format

The sender tells all the receivers and the group leader as to how many frames it is going to send, and also after how many frames, it will change the keys. In case of video-on-demand, it also tells the receivers which file it is going to send.

1	5	4	4
type	member Id	number of frames	number of frames, when the key is changed

Figure 4.9 LIVE VIDEO INFO Frame Format

4.4 Implementation Issues

We have so far seen all the data structures and the various frame formats that are used. Now we will see some issues which come up while using them and their solutions. This includes the synchronization issues involving the interspersing of data over the TCP byte stream, the choice of the encryption schemes, and the choice of TCP over UDP.

4.4.1 Synchronization

A very interesting issue that came up was that of synchronization. As it is known that the TCP uses a byte stream to transfer the data from one end to the other. If two processes or threads try to write to a TCP socket simultaneously, the data might get interspersed and the frame boundaries might get lost. The data will be corrupted and the two ends might abort the operation complaining of invalid frame formats.

It happened in the member node because we were creating a separate thread to send the KEY INFO packet. In the meantime, the DATA packets were also being sent on the same socket. Whenever a KEY INFO packet was sent, the frame formats got screwed up and the gateway and member aborted, aborting the whole system as a result. Then we realized that it was a synchronization problem as two threads were writing simultaneously on the same socket connected to the gateway. We then used a semaphore to implement mutual exclusion for the particular socket and the problem got solved.

4.4.2 Symmetric Encryption Scheme

To encrypt the video data, we initially started using the standard DES encryption scheme. Although this scheme is very secure, but the problem with it is that it is very slow. We were using the standard crypt command of the unix library to implement DES and it required the data to be read and then written to a disk file. This operation was very slow. Our implementation is designed for real-time applications and these expensive operations are not affordable.

We needed a scheme which was as secure as DES when the data is binary jpeg files, and at the same time very fast. At 30 frames per second, this operation needs to be performed 30 times a second. So, we came up with our own permutation encryption scheme. It is much faster than the DES algorithm as it only permutes the data instead of actually encrypting it. Let us see if we are in any way compromising on security.

1. We are using a 32 bit key and our implementation supports the scalability of the key to any length. Exactly the same key is required to decrypt the data. One encryption operation takes about 12 ms for a 7000 byte file, and to apply jpeg decoding, it takes about 13 ms. If somebody tries to generate all the permutations of keys (there are 2^{32} possibilities) and decrypts the data by exhaustively applying all the keys (or in other words, by using brute force) , then on average, it will take about 621 days to break the key, and about 6.21 days if, say, we use 100 computers simultaneously by dividing the key space among them.

2. We change the keys after every 1000 frames or so (approximately 30 seconds of video). So, the hacker code should be fast enough to break the code before that. If the code needs to be broken in say t seconds, then, one will require $\frac{2^{31}*25ms}{ts}$, or $\frac{53687091}{t}$ computers!!!

3. When the data is in the form of text, it is very easy to find patterns and to guess at the key. But since we are dealing with binary data here (compressed image in the form of jpeg), no definite patterns can be found and therefore it is very difficult (almost impossible) to take a meaningful guess at the key.

4. It works even for a bitmap image since if you permute all the pixels in the image, you cannot make heads or tails of the new image. Hence, we can easily afford to permute the data instead of encrypting it, when we can assume that the data will always be video data.

4.4.3 TCP vs UDP

We have used TCP for all the data transfers across the network in the protocol. Although it might be thought that UDP will be a better approach when dealing with large chunks of video data in a multimedia applications, it might be interesting to see why we chose TCP for our implementation:

1. TCP is a reliable protocol which means that there will be no bit-errors, no reordering or loss of packets etc. If we use UDP, our application will have to take care of all these reliability issues and it should be really efficient and fast.

2. Our main aim in this implementation is to analyze the feasibility of a secure multicast protocol in real-time applications. So, the main emphasis is on security issues rather than the networking implementations and reliability. TCP alleviates us of taking care of the reliability and other networking issues and we can therefore concentrate on the more important issue of security.

CHAPTER 5

Experiments and Results

5.1 Prototype Testbed

The testbed where our implementation and experiments were conducted consists of Sun Ultra-60 machines running SunOS Release 5.7, as shown in Figure 5.1. The systems were equipped with 10/100 Mbps Ethernet cards. There was one gateway, one group leader, and three members in the system. The experiments were designed to analyze the feasibility of real-time secure multicast and to measure its performance against security by recording the net frame rate achieved in a real-time scenario.

5.2 Performance Metrics

In a normal real-time video application, the user expects a frame rate of about 25 to 30 frames per second. This gives us a processing time of about 33 ms to 40 ms for one frame. Our main aim in this implementation was to check if it was possible to actually record, watermark, encrypt, and send a video frame within this time period to achieve the desired frame rate. So, our main performance metric was the net frame rate achieved. This directly depends upon the time it takes to record and encrypt a single frame, which

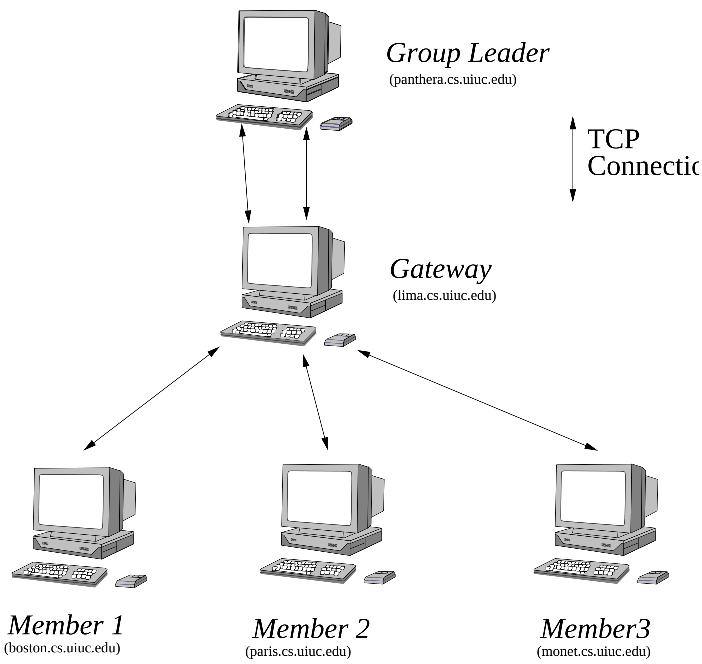

Figure 5.1 Experimental Setup

in turn depends on the size of the frame. So, we also measure and discuss these times in the coming sections.

5.3 Experimental Scenarios

We performed our experiments under two different scenarios - video-on-demand, and, video-conferencing. For both types of experiments, a gateway and group leader were started to set up the group. Then three members were made to join the group. The number of frames after which the key is changed was a variable for the different experiments. Also, the dimensions of the jpeg frames were 256x256 pixels giving them a disk size of approximately 6 to 7 kilobytes per frame. We shall discuss the two scenarios in greater details here:

5.3.1 Video-Conferencing

In this experiment, live video is recorded from a camera attached to the sender. The sender records the video in real-time frame-by-frame and sends it to the group after encrypting it. After the setting up of the group and the successful joining of the members, the machine *boston* acted as the sender and the machines *paris* and *monet* acted as the receivers as shown in figure 5.1. The experiment involved multicasting 5000 frames of video with an average frame size of about 6000 bytes.

5.3.1.1 Results

The number of frames after which the keys were changed were set at 300, 600, 900 and 1200 frames or approximately 10s, 20s, 30s, and 40s of video respectively. Here are the results of the four experiments along with the net frame rates that were achieved:

number of frames after which the key is changed	time to record (ms)	time to encrypt (ms)	time to send (ms)	total time for processing (ms)	net frame rate achieved (fps)
300	21.351	9.931	4.846	39.421	25.367
600	21.361	10.804	2.493	36.096	27.704
900	21.043	11.282	1.993	35.480	28.185
1200	22.909	9.615	1.441	34.715	28.806

Table 5.1 Summary of performance results for the secure video-conferencing application

5.3.1.2 Analysis

As can be seen, as we increase the number of frames after which the keys are changed, we get better performance, but at the same time, we are compromising on security. But, in a real world video broadcasting application, we shall normally be sending some hundreds of thousands of frames or some 3 - 4 hours of video. We can easily afford to change keys after say 40 s of video for such applications. A frame rate of about 29 is still achieved which is in the desired range. Even if someone gets the keys for a particular range of video, by the time they decrypt the video, the keys would have changed.

The average time to record a frame from the camera is 21.666 ms which gives us approximately 11 to 12 ms for the rest of the processing. Average time to encrypt the data is 10.408 ms. This data comprises of two copies of the original video frame or approximately 12,000 bytes. So, per byte it takes about 0.867 μs to encrypt. It takes on an average 2.693 ms to send a frame across the network. This time depends on the configuration of the network and also the load and congestion that the network is handling at that time. It is independent of the implementation of the protocol.

5.3.2 Video-On-Demand

In this experiment, the video is pre-recorded and stored in a data file on the disk. The sender reads this data file, extracts the video frame-by-frame and sends it to the group after encrypting it. Again, after the setting up of the group and the successful joining of the members, the machine *boston* acted as the sender and the machines *paris* and *monet* acted as the receivers as shown in figure 5.1. The experiment involved multicasting a pre-recorded video file of exactly 5000 frames with an average frame size of approximately 7000 bytes.

5.3.2.1 Results

The number of frames after which the keys were changed were set at 300, 600, 900 and 1200 frames or approximately 10s, 20s, 30s, and 40s of video respectively. Here are the results of the four experiments along with the net frame rates that were achieved:

number of frames after which the key is changed	time to record (ms)	time to encrypt (ms)	time to send (ms)	total time for processing (ms)	net frame rate achieved (fps)
300	9.174	11.635	3.226	24.136	41.432
600	8.947	11.443	1.914	22.395	44.650
900	8.887	10.975	1.625	21.580	46.340
1200	8.839	11.145	1.086	21.160	47.259

Table 5.2 Summary of performance results for the secure video-on-demand application

5.3.2.2 Analysis

Again, similar results can be observed. An increase in the number of frames after which the keys are changed, increases the performance. But, again in a real world video application, the huge amount of video frames will make it affordable.

A very interesting thing to note here, though, is the high frame rate achieved as compared to the video-conferencing application. This is because the video is pre-recorded and read from a file before processing. On an average, it takes about 8.962 ms to read a frame from a file as compared to 21.666 ms which are required to record a frame from the camera. The rest of the figures are comparable. For example, the encryption takes on an average 11.300 ms per data frame which comprises of two copies of video frames of size 7000 bytes each. So, it takes about 0.807 μs to encrypt one byte. Average time to send one frame is 1.963 ms which is again independent of the protocol.

CHAPTER 6

Conclusions and Future Work

This chapter presents some of the conclusions of the work carried out as part of this thesis. It also provides some guidelines for future work.

6.1 Conclusions

In this thesis we have analyzed the performance of a secure multicast protocol in a real-time environment. The results prove that it is feasible to have such a system in real-time. Where a video-on-demand application performs really well under such a scenario, a video-conferencing application is also not far behind achieving a net frame rate in the desired range.

6.2 Future Work

Although this implementation of the secure multicast protocol is very satisfying, there are many areas in which more work can be done to further improve the performance. We list some of them here:

6.2.1 Quality of Service

As can be seen, the net frame rate achieved in the video-on-demand application is about 45 frame per second which is much higher than the desired 30 frames per second. So, quality of service can be introduced at both the sending and receiving ends to make sure that the net frame rate achieved is within the required range. Also, in a video-conferencing application, quality of service can ensure a smooth video broadcasting by not encrypting a few frames to achieve a higher frame rate of about 30 frames per second.

6.2.2 Better Encryption Schemes

We have used the standard RSA scheme to encrypt the keys. The implementation of RSA which was used required the data to be in a data file which slowed down the things a bit. But since separate threads were created to take care of it and also it was done once every 1200 frames, it did not really matter. But still, a better implementation of RSA can be used to further enhance the performance. The permutation encryption algorithm works well in real-time due to its speed, but it can be improved further to have better security.

6.2.3 Multiple Sessions

This implementation can handle only one video session at one time. This is because of the limitation of the gateway. It can be improved to have multiple video sessions going on at the same time. The gateway can maintain a different list of sessions each comprising of the members involved in the session and the configuration of the sender.

6.2.4 Scalability

Due to the presence of only one gateway, scalability issues might arise. All the members have to connect to the same gateway which increases the load on the gateway. A tree structure for the gateway can be implemented comprising of parent and child gateways. This will take the load off the single gateway and greatly improve the performance of the system.

REFERENCES

[Bal97] Ballardie, A. RFC 2201: Core Based Trees (CBT) Multicast Routing Architectures, Sep 1997. `ftp://ftp.math.utah.edu/pub/rfc/rfc2201.txt`.

[Chi98] Chiu, D., Hurst, S., Kadansky, M., Wesley, J. TRAM : A Tree-based Reliable Multicast Protocol. Technical Report TR-98-66, Sun Microsystems Research, CA, july 1998.

[enc] Encryption Information. http://www.ladysharrow.ndirect.co.uk/Encryption/encryption_information.htm.

[Flo96] Floyd, V. Jacobson, Liu, C., McCanne, S., Zhang, L. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. In *Proceedings of IEEE/ACM Transactions on Networking*, November 1996.

[Pau96] Paul, S., Lin, J. A Reliable Multicast Transport Protocol. In *Proceedings of IEEE INFOCOM'96*, pages 1414 – 1424, 1996.

[Pau98] Paul, S., Whetten, M., Basavaiah, M., Montgomery, N., Rastogi, N., Conlan, J., Yeh, T. The RMTP-II Protocol. Technical Report draft-whetten-rmtp-ii-00.txt, Internet Draft, IETF, April 1998.

[RSAa] A Description of RSA. http://members.tripod.com/irish_ronan/rsa/rsa.html.

[RSAb] Welcome To RSA Security Inc. – The Most Trusted Name in e-Security. http://www.rsasecurity.com/.

[tri] RSA Laboratories — Cryptography FAQ — What is triple-DES? http://www.rsasecurity.com/rsalabs/faq/3-2-6.html.

[Tun] Tung, B. Kerberos - A Network Authentication System.