DISTRIBUTED SECURITY FRAMEWORK FOR MULTIMEDIA TRANSMISSION

BY

RAGHAVENDRA VINAYAK PRABHU

B.E., University of Mysore, 2000

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, 2002

Urbana, Illinois
ABSTRACT

In the last few years, there has been considerable progress in the area of multimedia streaming. We have seen, and continue to see a variety of applications being developed, which stream digital multimedia content like video and audio to a set of clients, who can be in the same building, on a local network, or even half way across the world, over the Internet.

As new and exciting applications continue to develop, an important issue is how to provide security to these applications. Security could involve a number of issues, like authentication of clients, data transmission security and copyright protection. For each of these security needs, a number of security protocols have been developed and a great deal of research continues in this area. The problem then is how to flexibly integrate security protocols into multimedia streaming applications.

This thesis presents the Distributed Security Framework for Multimedia Transmission (DSFMT). DSFMT provides a flexible and efficient way to add data transmission security to multimedia transmission applications. It defines the various components involved in a secure streaming application and describes their abstract interfaces. It is designed to work either as a black box Application Programming Interface (API) or a white box objected oriented framework.

Although currently, DSFMT is designed to provide only data transmission security, we believe its design is general enough to evolve into a complete security solution for a large class of multimedia content delivery applications.
To That
ACKNOWLEDGEMENTS

I would like to express gratitude to Prof. Klara Nahrstedt, my thesis advisor, for nurturing my interest in multimedia systems and giving me an opportunity to work in the MONET Research Group. Her technical guidance, constant support and patience played an important role in my research. This work would not have been possible without her inputs.

I am grateful to all the members of the MONET Research Group for stimulating discussions, inputs, help with anything I needed and general advice. My special thanks to Diwakar Gupta for providing me his video streaming application and helping me to use it in my experiments.

I thank DARPA for the financial support to conduct this research under grant number F30602-97-2-0121 and NSF for the infrastructure grant number EIA 99-72884 EQ.

Finally, I am forever indebted to my parents for their love and support.
# TABLE OF CONTENTS

## CHAPTER 1 Introduction

1.1 Background ..................................................................................... 1  
1.2 Secure Multicast of Multimedia Data ............................................... 2  
1.3 The Problem – Integrating Security into Applications ...................... 2  
1.4 Solution – DSFMT ........................................................................... 3  
1.5 Thesis Outline.................................................................................. 4

## CHAPTER 2 Related Work

2.1 Secure Multicast Protocols – A Quick Survey ................................. 5  
  2.1.1 Central Issue – Key Distribution ............................................. 5  
  2.1.2 Naïve Key Distribution ............................................................ 6  
  2.1.3 Core Based Tree Key Distribution .......................................... 8  
  2.1.4 Iolus Key Distribution.............................................................. 9  
  2.1.5 Single Transmission Key Distribution ..................................... 9  
  2.1.6 Hierarchical Tree Key Distribution ........................................ 10  
  2.1.7 Some Remarks..................................................................... 12  
2.2 Frameworks................................................................................... 13  
2.3 Research similar to DSFMT .......................................................... 16

## CHAPTER 3 DSFMT – Design and Architecture

3.1 DSFMT – Design ........................................................................... 18  
  3.1.1 What is DSFMT meant for? .................................................. 18  
  3.1.2 Design Principles.................................................................. 19  
  3.1.3 Design................................................................................... 22  
3.2 DSFMT – Architecture ................................................................... 26  
  3.2.1 Architectural Issues .............................................................. 26  
  3.2.2 DSFMT – Architectural Possibilities ..................................... 27

## CHAPTER 4 DSFMT – Interfaces and Implementation

4.1 Components – Interfaces and Implementation.............................. 31  
  4.1.1 SecureSender/SecureReceiver ............................................ 31  
  4.1.2 SecureProtocol ..................................................................... 36  
  4.1.3 Network (Server/Client) ........................................................ 38  
  4.1.4 Crypter .................................................................................. 41  
4.2 Application Model .......................................................................... 44  
  4.2.1 What does DSFMT provide? ................................................ 44  
  4.2.2 What does DSFMT NOT provide? ........................................ 45  
  4.2.3 What all does an Application Developer need to know? ..... 45  
  4.2.4 What is the Step-by-step Procedure of using DSFMT? ...... 46
CHAPTER 5 Experiments and Results .......................................................... 48
  5.1 Experimental Setup ........................................................................ 48
    5.1.1 Test Applications ................................................................... 48
    5.1.2 Test Bed .............................................................................. 49
  5.2 Results .......................................................................................... 50
    5.2.1 DSFMT-Application Integration Effort ................................... 50
    5.2.2 DSFMT Overhead .................................................................. 55

CHAPTER 6 Conclusions and Future Work ............................................. 61
  6.1 Conclusions .................................................................................. 61
  6.2 Future Work ................................................................................ 62

REFERENCES ....................................................................................... 64
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Naïve Key Distribution</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Hierarchical Key Distribution</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Model View Controller Class Diagram</td>
<td>14</td>
</tr>
<tr>
<td>3.1 DSFMT Sender – Class Interaction Diagram</td>
<td>23</td>
</tr>
<tr>
<td>3.2 DSFMT Receiver - Class Interaction Diagram</td>
<td>23</td>
</tr>
<tr>
<td>3.3 DSFMT Architecture – Subgroups</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Group Manager-Type Architecture using DSFMT</td>
<td>29</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 DSFMT Test Bed – PC Configurations</td>
<td>49</td>
</tr>
<tr>
<td>5.2 Encryption/Decryption overhead for 56-bit DES</td>
<td>56</td>
</tr>
<tr>
<td>5.3 Encryption/Decryption overhead for 112-bit Triple DES</td>
<td>56</td>
</tr>
<tr>
<td>5.4 Encryption/Decryption overhead for 168-bit Triple DES</td>
<td>56</td>
</tr>
<tr>
<td>5.5 Average Key Switch Delay</td>
<td>59</td>
</tr>
<tr>
<td>5.6 Average Key Distribution Times</td>
<td>60</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1 Background

The growth of the Internet accompanied by cheaper access to increasingly high bandwidth has led to the development of a large number of applications which deliver digital multimedia content to users. These applications range from streaming audio to video-on-demand and video conferencing.

Such applications often involve the transmission of large volumes of data from a few sender nodes to a large number of receiver nodes. An efficient way of achieving this kind of high-volume single-stream transmission is by using a form of group communication called *multicast*. Multicast is typically implemented efficiently in the lower network layers, to ensure that the bandwidth required for such communication is substantially lower than what would be required for sending the data separately to each receiver node (referred to as *multipoint unicast*).

The availability of multimedia content in digital form, as well as the growth in popularity of communication channels such as the World Wide Web, has brought a number of security issues to the forefront. These include copyrighting, ownership assertion, authentication, access control, and secure communication. The importance of these issues has led to extensive research interest in fields like security protocols, cryptography, digital signature, digital watermarking, and forgery discussion.
1.2 Secure Multicast of Multimedia Data

As explained above, multicast is a useful mechanism in transmitting high volume multimedia data to a group of receiver nodes. An issue that arises in this context is the security of this communication. There may be a variety of reasons for security. For example, the military may use a video conferencing application for tactical planning in times of war. Such an application should be designed to extremely secure against attempts to listen in by hostile people. A video-on-demand application too would need to use secure communication to prevent unauthorized users from gaining access to the transmitted video, although the security requirement here can be less stringent.

A common way of providing secure communication is by encrypting the data. Once a group key has been distributed securely among the communicating parties, any of the popular symmetric key encryption algorithms (like DES) can be used to ensure privacy of data. The problem, therefore, is to efficiently distribute the group key, both before starting the transmission of data and when it is required to change the key. Such communication of group keys to the receiver nodes often involves use of asymmetric key encryption algorithms (like RSA) that are typically inefficient and should be used infrequently. A key distribution protocol must be designed to ensure minimum overhead on the communication, as multimedia data usually has real time constraints. A survey of various proposed protocols for secure multicast is presented in Chapter 2.

1.3 The Problem – Integrating Security into Applications

As will be described in Chapter 2, several protocols exist for efficient key distribution during secure multicast. The main aims of these protocols include network architecture independence, robustness and scalability.
Let us consider the viewpoint of a multimedia application developer. Some of the system design issues that arise could be:

- Out of the various security protocols that exist, which one to use?
- How to integrate the protocol into the multimedia application?
- Some of the protocols have tunable parameters, which can be dynamically changed to suit the needs of the application. How to do this with minimum effort?
- At a future date, a better protocol may be developed, or circumstances may develop in which another existing protocol may be more useful than the one currently in use. How to change the protocol with minimum effort?

Although it is sometimes suggested that security should be tightly integrated into multimedia transmission applications at the time of development, we find some objections to this argument. First, this approach is not flexible enough to permit future changes to the security protocols or cryptographic algorithms with minimum or no change to the application itself. Second, application developers usually have no knowledge of security. The protocols are usually implemented independently by programmers with cryptographic backgrounds. Finally, security protocols, once implemented, can often be reused with little change in a number of similar applications. Tight integration with one application will make this quite difficult.

1.4 Solution – DSFMT

A framework is an object oriented programming concept defined as the design of an application or a sub-system expressed as a set of abstract classes and the way objects in those classes collaborate [CS497LN01]. Frameworks are used to simplify the design of complex software by effective reuse of code and design. In this thesis, we propose a framework for the design of secure multimedia streaming applications, called the Distributed Security Framework for Multimedia
Transmission (DSFMT). DSFMT uses object oriented programming principles to enable flexible and simple integration of security into multimedia transmission applications. DSFMT is designed to address and effectively solve all the issues outlined in the previous section. Some of the features of DSFMT are outlined below:

- DSFMT, by defining simple interfaces, delinks the application from the security protocols and encryption algorithms, without compromising security or performance.
- DSFMT is designed to be independent of the network and the protocol suit used.
- Since DSFMT allows security protocols to be “plugged into” applications, it becomes easy to replace the protocol used in an application with another, at a future date, if required.
- DSFMT can be used in a variety of configurations, for e.g., large scale with moderate security and single sender (video-on-demand) or small scale with high security and multiple senders (military video conferencing).
- DSFMT also allows the system designer to distribute the security responsibility among the nodes involved in the transmission, as appropriate.

Later chapters in this thesis will describe in detail the design of DSFMT as well as its implementation and performance.

1.5 Thesis Outline

The outline of this thesis is as follows: Chapter 2 examines related work by summarizing various secure multicast protocols and also describing how frameworks have proved useful in the past for various applications. Chapter 3 describes in detail the design and architecture of DSFMT. Chapter 4 deals with DSFMT interfaces and implementation details, while Chapter 5 lays out some experiments to show how DSFMT can be used and its effect on performance and flexibility. Chapter 6 has the conclusions and future work.
CHAPTER 2

Related Work

This chapter is organized as follows. Section 2.1 presents a brief survey of some of the secure multicast protocols. This field is continuously under research, and more and more efficient protocols are being developed. Section 2.2 discusses frameworks, why they are useful and how they are used. A couple of useful frameworks are described as examples. Section 2.3 briefly looks at research with similar aims to that presented in this thesis.

2.1 Secure Multicast Protocols – A Quick Survey

The Distributed Security Framework for Multimedia Transmission (DSFMT) is especially targeted at a set of multimedia applications that multicast content to a group of clients. We, therefore, begin by describing some of secure multicast protocols that have been developed.

We first present a discussion of some of the key issues involved in secure multicast. Some of this discussion is drawn from [RFC2627].

2.1.1 Central Issue – Key Distribution

The process of secure multicast involves two basic steps: key distribution and transmission of encrypted data. Once a group key has been securely established among the members of the multicast group, it can be used with any fast symmetric encryption algorithm to encrypt the data to be transmitted. Therefore,
the challenge in developing secure multicast protocols is primarily in designing efficient schemes for key distribution.

Let us identify the conditions that require establishment of a new group key during a secure multicast session:

- At the beginning of the session, when the group is formed – This is the first time that the group key is established.
- When a member of the group leaves or is expelled – Rekeying is required to prevent the member from using its key to decrypt future communication. This is called **Forward Rekeying**.
- When a new member joins the group – Rekeying is required if the new member is to be prevented from decrypting earlier communication (which it could have stored). This is called **Backward Rekeying**.
- When a timeout occurs – Keys are usually associated with a timeout after which they become potentially insecure. The length of this timeout depends on many factors like key length, encryption algorithm used, wired or wireless network etc. If such a timeout exists and occurs, rekeying is required. This is referred to as **Periodic Rekeying**.

We thus see that key distribution can take place quite often during a multicast session. It is very important to optimize this process. Most of the secure multicast protocols, therefore, differ from each other only in the key distribution scheme. The following sub-sections briefly describe some of the key distribution protocols that have been developed, starting from a naïve method.

### 2.1.2 Naive Key Distribution

Consider a group of N members. The server shares a key with each member, i.e., N keys in all (these keys are established when each member joins the group, typically using some public key algorithm). Whenever a new group key needs to
be communicated to the members, the server encrypts and communicates the new key to each member separately.

**Figure 2.1** Naive Key Distribution

For example, consider Figure 2.1, in which there is a server and 3 clients, A, B and C. If the key shared between member A and the server is called KA and the group key is K, the server sends $E(K)_{KA}$ over the multicast channel, where $E(X)_Y$ implies X is encrypted by Y, to communicate the new key to member A. Thus, in this method, a total of N encryptions and N transmissions would be required to communicate the new key. Obviously for large N, the delay this would cause is very expensive and would be unacceptable, given the high bandwidth requirements of multimedia. As for storage requirements, the group leader has to store N keys, while each member has to store just one key besides the group key.
2.1.3 Core Based Tree Key Distribution

This method was proposed by Ballardie in [RFC1949]. The following description is taken from [Chu99].

The Core Based Tree Key Distribution is based on the hard state multicast protocol like the Core Based Tree, where the multicast routers permanently maintain the state of the multicast tree, e.g., their adjacent routers in the tree. The key distribution algorithm can take advantage of the hard state approach by appending various security information into the hard state of the tree, e.g., the access control list (ACL), the group key, and the key-encrypting key (which is used for re-keying the group key). The algorithm contains the following steps: (1) the initiating host first communicates, via asymmetric encryption, the ACL to a core router, (2) the core router generates the group key and the key encrypting key, (3) when a new non-core router joins to become a part of the multicast tree, the core router authenticates the new non-core router and passes the security information using asymmetric encryption to the non-core router, and (4) as the multicast tree expands, the authenticated non-core router further authenticates other new incoming non-core routers and distributes the security information.

This distributed key distribution approach has an efficiency improvement over a centralized key distribution approach where the group key is distributed to the group members by only one or a few centralized servers. However, the security level of the CBT key distribution scheme is based on a strong assumption that the involved multicast routers can be trusted not to leak the security information. In addition, the key distribution algorithm does not address dynamic membership operations.

Note that the CBT protocol does not satisfy our requirement of network protocol independence.
2.1.4 Iolus Key Distribution

This method was proposed by Mittra in [Mitra97]. This description too is taken from [Chu99].

This method divides the group into regional subgroups, and each subgroup is managed by a trusted Group Security Intermediary (GSI). Each subgroup is treated almost like a separate multicast group with its own subgroup key and its own multicast channel. The GSI in each subgroup manages its subgroup key distribution and authenticates new members joining/leaving its subgroup. The advantage is that the subgroup runs independently of each other, and the GSI can perform dynamic member operations efficiently and independently without involving members of other subgroups. To bridge data across the subgroups, the GSIs use another separate multicast channel managed by the Group Security Controller (GSC). As a result, each data transmission requires three different multicasts. The sender first multicasts data in its subgroup channel. When the sender's GSI receives the data, it multicasts the data to the other GSIs. Then the other GSIs multicast the data to their subgroup members through their subgroups' multicast channels.

Iolus, similar to the CBT approach, depends on the security level of the various GSIs residing at various locations in the network. Its overhead contains the three multicast transmissions per data transmission, the management of multiple subgroups, and multicast channels.

2.1.5 Single Transmission Key Distribution

Having looked at a couple of network protocol dependent schemes, let us consider a network independent approach. This method, proposed as part of the protocol in [Chu99], saves on the number of transmissions by clubbing the N rekeying transmissions of the naive protocol into one message. This large message will consist of the group key encrypted by each of the N member keys
in a sequence. Each member knows where in the list is the key it can decrypt. It retrieves, decrypts and stores it appropriately.

In this method, however, there is still the requirement of N encryptions. Moreover, transmission time of a message increases considerably with its size. The efficiency improvement over the naive protocol, therefore, is not much.

### 2.1.6 Hierarchical Tree Key Distribution

This scheme was proposed in [RFC2627]. The description that follows is taken from [Chu99].

Hierarchical tree key distribution is an efficient and scalable approach that supports dynamic group membership. The algorithm contains the following steps:

- Each multicast group contains a key server that maintains a rooted tree structure, and each leaf node corresponds to a group member.
  - Each node in the tree contains a key - each leaf node holds a pairwise key established between the server and the member, each intermediate node holds a key generated by the server, and the root node holds the group key that is used to encrypt the data.
  - The server sends to each group member a sequence of keys on the path from his/her leaf node to the root.
  - Each key is encrypted with the previous key in the sequence to ensure security.

Now, if a member is expelled from the group, the server generates a new group key and changes intermediate and group keys on the path from the expelled member’s leaf node to root. Each new key is encrypted using its two immediate child keys. This new sequence is assembled into one rekey message that is multicast to all members using a multicast channel. Upon receiving the rekey message, the members decrypt only those keys that they need and no more.
To illustrate, let us consider Figure 2.2. The Server maintains a tree (in this case, binary) of 7 keys in all. Each client is communicated the keys along the path from its key to the root. For example, client A is first sent $E(K1)_{KA}$ and then $E(K)_{K1}$. To expel client A, the server sends the following messages over the multicast channel – $E(NK1)_{KB}$, $E(NK)_{NK1}$ and $E(NK)_{K2}$, where NK is the new group key and NK1 is a new key which replaces K1. Since member A cannot decrypt the first message, it does not know NK1. But it does not know K2 either, so it cannot figure out NK, and thereby is unable to decrypt future communicated data.

For a group of size N, each membership update rekey operation requires one rekey message and $O(\log N)$ encryptions. The server needs to store the complete tree of $2N – 1$ keys, while each node stores around $\log N$ keys. This method, therefore, is very efficient and scales well.
2.1.7 Some Remarks

We have described above some of the better-known secure multicast protocols. The hierarchical key distribution protocol is the most efficient of these and has been extensively researched to further improve its properties. For example, a recent paper [Banerjee01] describes the use of a particular method of clustering to improve performance by several orders of magnitude, for bulk changes in group composition. This and other such protocols, which are based on the hierarchical tree protocol, are not described here for the sake of brevity.

Here are some observations we can make from this protocol survey:

- There are essentially two categories of protocols – centralized and distributed. The centralized protocols, like the Hierarchical Key Distribution Protocol, have the inherent disadvantage of having to depend on one node for successful operation. The central node can become a performance bottleneck. Distributed protocols do not have this problem. However, the two distributed protocols we saw were network dependent. This is not a property we would like to have. Later in this thesis, we will show how DSFMT can mitigate this problem by making it easy to hierarchically distribute the responsibility of central nodes among other selected nodes in the system.

- As we can see from the survey, there are already several protocols available. Each protocol has some advantages and disadvantages in terms of complexity, efficiency, storage requirements etc. Which protocol to use depends on the requirements of the application in question. These requirements may change dynamically after deployment, such that another protocol becomes more suitable. Also, a new protocol may be developed which outperforms the implemented one, in terms of application requirements. Therefore, applications should be flexible enough to allow changes in security protocols and parameters without too much trouble. This is one of the main aims of DSFMT.
2.2 Frameworks

The following discussion is based on [CS497LN01].

A framework is defined as the design of an application or system expressed in terms of abstract classes and how they collaborate. Some of the features of a framework are:

- **Object-oriented** – Frameworks are developed using object-oriented design principles.
- **Reusable** – A framework’s main purpose is to be reusable. It is particularly useful when there are sets of applications or systems that are “essentially” similar. A set of components can be developed for one application and then generalized to be reusable in other similar applications.
- **Extensible program** – A framework is practically just a program, but one that can be easily extended to suit an entire class of applications. Thus, a framework involves both design and code.

Once a framework has been developed, it becomes much easier to build new applications using it. Since the design becomes much more clear, applications built using frameworks tend to be flexible and easy to maintain. The customers of frameworks are application programmers, who can take a framework, reuse some components, develop some others and put together new applications.

Frameworks can be white box or black box. White box frameworks need the users to know the internal design and implementation of its components. They are typically customized by sub-classing the components. They are easier to design, but harder to use. Black box frameworks, on the other hand, are much more difficult to design, but relatively easy to use, since the user just has to know the interfaces between components. Customization is achieved by plugging in and configuring the right components. Application development using black-box
frameworks does not usually involve much programming. As we shall show, DSFMT is designed to be a black box framework.

We now present a couple of well-known frameworks as examples.

**Model-View-Controller (MVC)**

The Model-View-Controller (MVC) is a widely used framework first developed in 1980 as part of Cincom’s Smalltalk. MVC has undergone continuous fine-tuning since then, and now a much-refined version is part of Cincom’s VisualWorks. The class diagram of MVC is given in Figure 2.3.

![Model View Controller Class Diagram](image)

**Figure 2.3** Model View Controller Class Diagram

MVC is a general framework used to quickly build visual interfaces for applications. The basic MVC, as its name suggests, consists of three classes, namely, Model, View and Controller. Model represents the application model, and by itself, has no user interface. The user interacts with the application through one or more Views. A View is just an image of the Model, as it appears to the user. For example, a clock application may have an analog view and a digital view. The user can choose how to view the model, which is essentially only a counter. MVC is designed such that any change to the Model is automatically reflected in the View.

The Controller is used to convert user actions into messages that can be handled appropriately. These messages cause some change in the state of the model. For example, the user may wish to change the currently set time. To do so, he/she should interact with the View. Such interaction is converted into
appropriate messages by the Controller, which eventually causes a change in state of the application’s counter. This is reflected back in the View.

MVC is used extensively in VisualWorks to develop almost every application that requires a visual interface. Application development is much faster, since the interaction between the classes has already been worked out in the framework. Further, to build most new applications, already existing UI classes can be used with minimum changes. If necessary, the developer can extend the existing code by subclassing. Since a programmer who uses MVC needs to understand the functioning of each class, MVC is a white box framework.

**HotDraw**

HotDraw is a framework for 2-D graphics originally developed by Kent Beck and Ward Cunningham at Tektronix.

HotDraw consists principally of 5 components – DrawingEditor, Drawing, Figure, Tool and Handle.

DrawingEditor represents the whole system.

A Drawing is a collection of figures. It keeps track of the figures it contains and can perform operations like adding and removing figures, moving figures etc.

A Figure knows how to display itself. It knows which drawing it is part of and its location in that drawing. It can change its size or structure, and inform other drawings or figures when it changes.

A Tool consists of an icon in the palette and a cursor on the canvas. It is actually just a state of the controller. The controller knows the current Tool and delegates all operations to it.

A Handle is a special kind of figure that looks like a little box and is attached to another figure. It performs some operation on the figure when pressed. Each Figure has a set of handles associated with it.
The objects described above interact together to form a 2-D graphics editor. HotDraw is typically implemented on top of the MVC framework. Since the framework is bundled with a set of standard tools and handles, the programmer usually only needs to configure the objects together to build an application. There is no need to understand the implementation of each object. Therefore, this is an example of a black box framework. HotDraw-style graphics editors are widespread today.

From the above two examples, it should be clear what frameworks are meant for and how they are designed and used. There are several such frameworks in use today for developing a variety of classes of applications.

2.3 Research Similar to DSFMT

As explained earlier, the aim of this thesis is to present a framework for developing secure multicast applications. There are a few others in the secure multicast community who are independently pursuing research with similar aims. Most of this research originates from the IRTF Secure Multicast Research Group [SmuG], which is now broken up into two groups, namely IRTF Group Security Research Group [GSEC] and IETF Multicast Security Research Group [MSEC]. While GSEC aims at investigating new and upcoming technologies related to multicast communication, MSEC deals with standardizing well-developed technologies. We now point towards two very similar research attempts.

A Toolkit for Secure Internet Multicast [Saha98]

The architecture of the toolkit revolves around a group owner who interacts with one or more “controller/reflector” nodes. These nodes in turn interact with group members. The reflector deals with distribution of data messages, while the controller is responsible for control messages. The toolkit defines the procedures for joining and leaving the group and message flow. Wallner’s hierarchical tree [RFC2627] is used for key distribution. The application programmer is provided
with simple send/receive API and encryption can be turned on/off with a flag. The Controller can be configured using the provided GUI.

This differs from DSFMT in that it is a ‘toolkit’ and not a ‘framework’. It is much easier for an application programmer to use a toolkit, but in contrast to a framework, the security protocols and encryption algorithms cannot be changed or adapted at will. The programmer can only configure what is available through the GUI. A framework, on the other hand, provides a lot more flexibility, as will be shown later in this thesis.

**Reference Framework and Building Blocks [Hardjono00]**

This IRTF draft describes a reference framework for secure multicast that is under development. The framework aims at capturing the essential components of a secure multicast application. It consists of four main components – Sender, Receiver, Group Controller/Key Server and a Policy. It specifies the kind of interaction between these functional units. It identifies two kinds of designs – Centralized and Distributed. Centralized design has only one policy domain, while the policy decisions in a distributed design span across administrative domains. The draft then develops 4 building blocks, namely, Data Transforms, Membership Management, Key Distribution and Policy Definition.

The primary difference between this work and that presented in this thesis is that the framework presented above is very general and aimed at encompassing all the various secure multicast architectures. Unlike DSFMT, it is not a framework in the sense of object oriented programming and does not, at this point of time, involve actual code. It is geared towards standardizing some of the secure multicast technologies, and not meant to be directly useful to application programmers.

Having described work related to this thesis, we will now proceed, in the next chapter, to explain the design of DSFMT in detail.
CHAPTER 3

DSFMT – Design and Architecture

This chapter has two sections. Section 3.1 describes in detail the design of Distributed Security Framework for Multimedia Transmission (DSFMT) and its component classes. Section 3.2 deals with the architecture of an application developed using the DSFMT framework.

3.1 DSFMT – Design

This section is organized as follows. We begin by describing what DSFMT is meant to do, and the kind of applications that could use it. We then list out the design principles we identified and kept in mind when designing DSFMT. Next, we develop the design of the framework in detail, along with functional design of each component in it. Precise details like interfaces and implementation details are covered in Chapter 4.

3.1.1 What is DSFMT meant for?

Before we look into the design of DSFMT, let us look into the kind of applications that DSFMT is meant for, and what DSFMT provides to them.

DSFMT is a framework that is meant for providing data transmission security to multimedia streaming applications. Examples of multimedia streaming applications include video-on-demand, video-conferencing and live telecast of video or audio. Although DSFMT is especially targeted towards applications that use multicast, it can be used for unicast (possibly multipoint) streaming applications as well.
DSFMT provides these applications with secure streaming by encrypting the transmitted data. DSFMT also takes charge of key distribution and management in such a way that the application need not concern itself with them.

Although DSFMT can be used as a black box for secure transmission, it is also meant to provide the application developer with flexibility in terms of choosing a particular security protocol, encryption algorithm or network protocol suite.

We are now ready to look at the design principles of DSFMT.

3.1.2 Design Principles

Before working out the design of DSFMT, we identified a set of principles to follow during the design. These principles were required to ensure that the design was consistent with the goals of DSFMT, and that the framework actually achieved all that it was meant to.

In order to develop the design principles, let us examine the issues facing a multimedia streaming application developer.

**Which security protocol to choose?**

The first issue is to choose the security protocol most suitable to the application at hand. For secure multicast, for example, there are a wide variety of protocols, as we have seen, and there are tradeoffs between them in terms of efficiency (both time and bandwidth) of key distribution, storage requirements, scalability, complexity of implementation and so on. Some protocols scale well, but may not be as efficient. Some others are very efficient for small group sizes. Yet another protocol may be perfect in terms of efficiency required, but may be too difficult to implement and deploy in the time frame available.

It would help if there were pluggable implementations of protocols to choose from. Then the designer could try out some protocols to see which performs best. Better still, the process of choosing the right protocol, given some parameters
like number of members in the group, real time requirements and storage space available, could be automated.

**How to integrate the security protocol with the application?**

Upon choosing the right protocol, the next step is to integrate it with the application. Usually (although it is inadvisable to do so), the application itself is developed independently and security is added as an afterthought. Implementing the protocol and encryption algorithms in such a manner as to suit the application design is not usually easy.

If a comprehensive design of the whole system, including security aspects, were to be available, it would help ease this problem.

**How to change/reconfigure the implemented security protocol?**

Depending on the parameters of the application, the system designer initially chooses and configures a security protocol. However, some of the parameters of the system change at a future date. For example, the number of users of the system may grow much faster than originally envisaged. Also, even if the parameters do not change, given the pace of research in this area, a far more suitable security protocol may be developed. In such cases, how to change or reconfigure the protocol? This will not be easy if it has been tightly integrated with the application.

A good design that delinks the security protocol from the rest of the application would help here.

**How to modify the level of security?**

Each application has a certain security requirement associated with it. For example, a military video conferencing application would require a very high level of security as compared to a video-on-demand application. The latter would probably prefer sacrificing some security for higher throughout, if the video being transmitted were not so valuable. However, the security level may need to be
increased/decreased at a later date. This would mean changing the key size and/or encryption algorithm itself.

Here again, delinking the encryption/decryption routines from the rest of the application and the security protocols would help.

**How to ensure network and network protocol suite independence?**

If not carefully designed, a lot of code specific to a suite of network protocols gets mixed up in the application and security protocol implementation. If it were to be deployed over a different network, a lot of reworking would need to be done.

Once again, delinking the network specific code from the other components would make things easier.

Besides the above, there are a couple of architectural issues too, which are discussed in section 3.2.

DSFMT had to be designed to address all these issues. The following design principles were, therefore, laid out:

1) The first important principle is to ensure ease of integration of a given application into DSFMT. Since most applications use socket libraries for communication code, the best way to make it easy to integrate DSFMT would be to have DSFMT provide a socket-like interface to the application.

2) Another critical principle is to enable different security protocols to be able to work with DSFMT. To achieve this, the most general aspects of a security protocol must be identified, and used to define the interface of the security protocol module.

3) The encryption/decryption algorithms should form an independent module.

4) The networking implementation should be separated from the other modules.
5) The interaction among the above modules should be generalized as much as possible, so as to accommodate the many different protocols and encryption mechanisms, together with a large class of applications.

6) The design should keep in mind the architectural challenges as well (these are discussed in section 3.2).

Having identified these principles, we can now explain the design in detail.

3.1.3 Design

Keeping in mind the design principles cited above, we designed DSFMT to have five main components (see figures 3.1 and 3.2 for interaction diagrams):

1) SecureSender/SecureReceiver – The SecureSender on the sender side and the SecureReceiver on the receiving side are the components that form the core of the DSFMT framework. These two components are responsible for mediating between all the other components. The other components interact among one another mainly through these two components.

2) Sending/Receiving Application – The Sending Application on the sender side contains the application code responsible for generating the multimedia data to be transmitted. The Receiving Application contains the application code that receives the transmitted data on the receiver side. These applications interact with DSFMT only through the socket-like interfaces of the SecureSender/SecureReceiver modules. That way, the applications are delinked from the implementation of other framework components, and any changes in those modules do not generally affect the applications.

The interface is specified in detail, in the next chapter, but the essential methods that the application uses are given below.

- When the Sending Application is ready to send data, it uses the secureSend method of the SecureSender, which has a similar interface to that socket library function send.
Figure 3.1  DSFMT Sender – Class Interaction Diagram

Figure 3.2  DSFMT Receiver – Class Interaction Diagram
• When the Receiving Application is ready to receive data, it uses the
  \textit{secureReceive} method of the \textit{SecureReceiver}, which has a similar
  interface to that socket library function \textit{receive}.

When the application starts up, before it can use DSFMT, it needs to allow it to
setup, and give it some session parameters, like the number of receivers, their IP
addresses and security level required. After the session is over, it should indicate
the same to DSFMT to allow it to shutdown smoothly.

3) \textbf{SecureProtocol} – This component is responsible mainly for key distribution
and implements a key distribution protocol. DSFMT is designed to provide
sender-side key distribution responsibility. This way, in a multiple-sender
scenario, each sender enforces its own security and distribution policy. There is
no separate group management entity in the framework. If a certain protocol
requires such a separate entity, it can use a dummy sender node (see section
3.2 for more on this). Alternately, the group manager(s) can be made to reside on
one or more of the sender nodes. The \textit{SecureProtocol} object is responsible for
sending key distribution and other control messages on the sender side, and
receiving them and taking appropriate action on the receiver side.

Although ideally the \textit{SecureProtocol} object should probably interact only with the
mediator objects, this is difficult to do in practice. This object needs to directly use
the interfaces to the \textit{Network} and \textit{Crypter} modules too (this is not shown in the
figures).

4) \textbf{Crypter} – This component encapsulates the encryption and decryption
algorithms. The two important methods it implements are given below.

• When the \textit{SecureSender}/\textit{SecureReceiver} wants to encrypt some data,
  it uses the \textit{encrypt} method of the \textit{Crypter}.

• When it wants to decrypt some data, it uses the \textit{decrypt} method.

The \textit{Crypter} is setup initially to use an appropriate set of algorithms, based on the
security level of the session, which is provided to DSFMT by the application.
5) **Server/Client** – These modules encapsulate the code for sending and receiving messages over the network. The interaction between them and the *SecureSender/SecureReceiver* objects are mainly through these methods:

- When the *SecureSender/SecureReceiver* needs to send a message over the network, it calls the `sendMessage` method, which has a signature similar to the standard socket library `send`.

- When the *SecureSender/SecureReceiver* is ready to receive a message, it calls the `receiveMessage` method, which has a signature similar to the standard socket library `receive`.

The interface is designed to make it possible to encapsulate a variety of network protocols (for instance, transport protocols like TCP and UDP) within these two modules, in such a way that the rest of the framework and the application are not tied to a specific implementation.

Note that the *SecureSender* can receive messages and the *SecureReceiver* can send messages over the network. This is required, since control communication can go either way. However, data messages are usually only sent from the Sending Application to the Receiving Application. In most protocols, there is minimal work on the receiver side, and the *SecureProtocol* object may be small or absent.

We have explained the design principles of DSFMT and described its various components. We have briefly outlined the interfaces between them and how they interact. The interfaces will be covered in more detail in the next chapter.

The next section covers the other major perspective – architecture of a DSFMT application.
3.2 DSFMT – Architecture

This section is organized as follows. We first lay out some of the architecture issues involved in building a secure streaming application. We then show how DSFMT is designed to address them.

3.2.1 Architectural Issues

In section 3.1.1, we brought out some of the principles involved in designing the secure streaming application from the software perspective. There is, however, another viewpoint – architecture. An application of this kind involves several nodes on the network. There can be one or more senders and several receivers. How are these nodes organized? Where is the group management responsibility placed? These questions come under the gambit of “application architecture”. We now list out some of the things to be considered.

1) **Centralized vs. Distributed Control** – Key management and other group management functions are centrally handled by a group leader node in some protocols, while some other protocols distribute the responsibility among other nodes in the network. Centralized protocols typically have the disadvantage of the central node being a bottleneck in communication, especially for large groups. Distributed protocols, while being more efficient, tend to usually be dependent on the underlying network protocols. For example, the Core Based Tree protocol [RFC1949] distributes the group management responsibility among certain intermediate routers, but the protocol requires Core Based Tree routing to be used. We are, however, only interested in protocols that are network protocol suite independent. Therefore, it would be useful if there were to be some way of using a centralized protocol, but hierarchically distributing the responsibility by having sub-groups. We will show how DSFMT makes this possible.
2) **Multiple Protocols?** – Consider an application with a very large group of members spanning across continents. Do we use the same security protocol and security level throughout the system? There may be various factors that may render uniformity unviable. For e.g., traffic patterns and laws vary from region to region. One combination of protocol/security level may be suitable in a given country while another combination might be the best for another country. In such situations, again, it would be useful to be able to easily create sub-groups of members, with possibly different security protocols and/or levels.

3) **Multiple Senders** – Another case is that of a many-to-many transmission application. Of the various senders, each sender may have a different security level requirement for its data. How to accommodate such differences within the same system? DSFMT provides a solution, as will be shown below.

The following sub-section shows how DSFMT provides architectural flexibility to system designers.

### 3.2.2 DSFMT – Architectural Possibilities

The DSFMT framework puts group management responsibility at the sender node. The rationale is that each node that wants to send some data takes the responsibility for protecting it. Obviously, the node can trust itself rather than some third party (although such a third party may be required for cases where authenticity of the sender is an issue). For completeness, DSFMT does provide the system designer the ability to place some security responsibility on the receiver nodes, but doing so is not advisable, since receivers cannot be trusted. We now explore some architectural possibilities.

**How to create subgroups?**

As shown in the previous sub-section, subgroups can provide a great deal of flexibility in terms of hierarchically distributing the group management
responsibility in a centralized protocol, using multiple protocols in the same system and so on. DSFMT makes it easy to create subgroups by placing both a DSFMT Receiver and a DSFMT Sender on the same node, as shown in Figure 3.3.

![DSFMT Architecture - Subgroups](image)

**Figure 3.3** DSFMT Architecture – Subgroups

The dummy DSFMT Receiver Application object actually only acts as the *Network* object of the DSFMT Sender. Since a DSFMT Sender can enforce its own protocol and security levels, it can effectively act as the group manager of a subgroup. Using this method, it is easy for the system designer to distribute functionality among members of the group, in case it gets large and unwieldy. All the designer has to do is to install DSFMT Sender code in the node chosen for subgroup management.
How to implement protocols that require third party group management?

As we saw earlier, some secure multicast protocols need a separate group leader or group manager to be in-charge of group management and key distribution. In DSFMT, there are two options for implementing such protocols:

1) Place the group leader on one of the sender nodes. This is the easy way.

2) Create a dummy DSFMT Receiver on a node. All the senders send data to this node. This node then sends the data to the receivers, through a DSFMT Sender. In essence, this node acts like a gateway. The DSFMT Sender on this node can act like a group leader. This situation is illustrated in Figure 3.4.

![Figure 3.4 Group Manager-Type Architecture using DSFMT](image)

Depending on the nature of the system, the designer may come up with alternative implementations as well.
What about multiple senders with different security requirements?

This issue is easily solved by DSFMT, since the security responsibility is placed on the sender node. Each sender can apply its own security protocol and policy.

It should now be clear how DSFMT provides a lot of flexibility both in implementing a secure multimedia transmission application and in designing its system architecture. If the proper protocols and algorithms are implemented, DSFMT could prove useful in providing data transmission security to a broad category of applications.

The next chapter covers in detail the component interfaces and implementation aspects of DSFMT.
CHAPTER 4

DSFMT – Interfaces and Implementation

This chapter is organized as follows. We first outline in some detail the interfaces of each component of DSFMT and their implementation. This is followed by a look at the application model of DSFMT.

4.1 Components – Interfaces and Implementation

In Chapter 3, we mentioned the various components in the DSFMT framework and touched briefly upon how they interact. In this section, we take each component, describe its abstract interface and provide some details of its implementation.

4.1.1 SecureSender/SecureReceiver

The SecureSender component on the sender side and the SecureReceiver on the receiver side are the central components in DSFMT. These components have two important functions:

- To delink other DSFMT components from each other as much as possible, so as to make it possible to modify one component without having to make (many) changes in another. In essence, they act as mediators between other DSFMT components.

- To expose a simple interface to the applications. The sending and receiving applications deal only these two components, and do not need to care how other components are implemented (however, if they do care, then DSFMT
provides a mechanism for the applications to interact with specific components, as will be shown later).

Let us now look at the interfaces and implementation of these two components.

**SecureSender**

class ASecureSender
{

public:

virtual void setup(SessionParam) = 0;

virtual void secureSend(const char *, int) = 0;

virtual void eventOccured(char *, int) = 0;

virtual void notifyNet(char *, int) = 0;

virtual ~ASecureSender() {};

};

The *ASecureSender* interface is used by the application that sends data. The class *ASecureSender* is an abstract class that defines the interface of this component. Although the implementation of this component is not expected to vary much, we decided to make it abstract in order to allow it to be changed easily, if required.

The sending application uses this interface to setup a secure channel to its receivers, and to stream data to them. In Section 4.2, we outline the way an application can use this interface. Here, we explain what *SecureSender*, which is the default implementation class for the *ASecureSender* interface, does when each method is called.

The method *setup* is used by the sending application to inform the *SecureSender* that a streaming session is going to start. The *SecureSender*, in order to setup
the session, needs some parameters, which are given to it through the SessionParam structure. As DSFMT evolves, the SessionParam structure can be modified to include other parameters, but currently it is defined as follows:

typedef struct _SessionParam
{
    int sLevel;
    int noOfClients;
    char *ipList;
} SessionParam;

The sLevel parameter is an integer that specifies the security level of the session. The SecureSender uses this parameter to configure the Crypter component. The Crypter component in turn can map each level to an appropriate encryption/decryption algorithm. The other parameters are an integer specifying the number of receivers and a character array giving their IP addresses.

SecureSender, after instantiating other components, passes control first to the Network asking it to setup connections with the receivers, and then to SecureProtocol, allowing it to contact the receivers and distribute the group key.

The secureSend method is used by the application to actually send its data. The signature of this method is similar to the standard socket library function send. SecureSender uses the Crypter to encrypt the data, attaches an appropriate header indicating it is a data message, and then asks the Network to send it to the clients. Before encrypting the data, however, SecureSender checks with the SecureProtocol whether a group key change needs to occur. If it does, it asks SecureProtocol to carry out the update. The need for this will become clearer in Section 4.1.2, where we describe the SecureProtocol module.
The \textit{eventOccured} method is used by the application to inform DSFMT that a particular event of significance, like a member leaving or joining has occurred. \textit{SecureSender} just forwards this to the \textit{SecureProtocol} module to take appropriate action. The \textit{SecureProtocol} component, in turn, can ask the \textit{Crypter} and/or \textit{Network} components to take suitable action.

The \textit{notifyNet} method is provided in cases when the application needs to exchange information with the \textit{Network} module. This may be required if the \textit{Network} module has a special implementation which requires some coupling between it and the application. One example of its use is to allow the application to set socket options, if socket-based communication is implemented by \textit{Network}. \textit{SecureSender} just forwards this notification to the \textit{Network} module.

Finally, the \textit{ASecureSender} interface has a virtual destructor to enable its implementation class to clean up, when the application calls \textit{delete} on it. \textit{SecureSender} implements its destructor to call \textit{delete} on other DSFMT components.

\textbf{SecureReceiver}

\begin{verbatim}
class ASecureReceiver
{

public:

  virtual void setup() = 0;
  virtual void secureReceive(char *, int *) = 0;
  virtual void notifyNet(char *, int) = 0;
  virtual ~ASecureReceiver() {};

};
\end{verbatim}
The abstract class `ASecureReceiver` is the counterpart of `ASecureSender` on the receiver side. This is the interface that DSFMT exposes to the receiving application. We provide a default implementation for this interface called `SecureReceiver`, whose implementation we describe below. Again, how the application uses this interface will be outlined in section 4.2.

The method `setup` is similar to the one on the sender side, except that it accepts no session parameters. This is because the parameters on the receiver side (as of now, only security level) are configured based on the information received from the sender. When this method is called, `SecureReceiver` instantiates the other DSFMT components and instructs the `Network` component to wait for a connection from the sender. Once the connection is established, it hands over control to the `SecureProtocol` component to setup the secure session and receive the group key.

The receiving application calls `secureReceive` when it is ready to receive data from the sender. `SecureReceiver` asks `Network` to receive a message, and when it gets it, decrypts it and checks if it is a data or control message. If it is a control message (rare case), it is forwarded to the `SecureProtocol` component and the action is repeated. Otherwise the received data is returned to the application.

Just like the `ASecureSender` interface, `ASecureReceiver` provides a `notifyNet` method for direct communication between the application and `Network` component. As with `ASecureSender`, the `ASecureReceiver` interface provides a virtual destructor. `SecureReceiver` implements its destructor to call `delete` on the other components.

We can see from the above discussion how the `SecureSender/SecureReceiver` components form a kind of Façade between DSFMT and the application, and also as a kind of Mediator between DSFMT components.
4.1.2 SecureProtocol

This module is the most important one in the DSFMT framework, since it carries the key management responsibility. For uniformity, the SecureProtocol modules on the sender and receiver side both share a common interface, defined by the abstract class below. Obviously, the implementations on the sender and receiver side are quite different.

class ASecureProtocol
{
  public:

    virtual void setup() = 0;
    virtual void eventOccured(char *, int) = 0;
    virtual void gotControlMessage(char *, int) = 0;
    virtual bool isGroupKeyChanged() = 0;
    virtual void updateGroupKey() = 0;
    virtual ~ASecureProtocol() {};
};

The implementation of the ASecureProtocol interface is closely tied to the security protocols described in Chapter 2. Currently, we have implemented two protocols: Manual Key distribution and Heirarchical Key distribution. Rather than repeating details of these protocols again in this chapter, we will outline in the following discussion a general idea of what each method in the above interface represents. Our implementation just maps this general idea to the specific key distribution protocol, and as such, the details of our implementation of each protocol are not provided.
The method `setup` is called by the `SecureSender/SecureReceiver` components to allow the `SecureProtocol` modules on the sender and receiver sides to negotiate, exchange keys and finally communicate the group key from the sender to the receivers. Our implementations of `ASecureProtocol` on the sender side take the number of receivers and handles to the `Crypter` and `Network` components as constructor arguments. It is similar on the receiver side, except that the ‘number of receivers’ parameter is obviously absent. The implementation of `setup` depends on the key distribution protocol used. For example, in hierarchical key distribution, a tree of keys is formed on the sender side with each leaf representing a receiver. Each receiver is then sent all the keys in the path from the node representing it to the root in the key tree. The group key is at the head of the tree.

The method `eventOccured` is used to indicate to the `SecureProtocol` component that an event of significance has occurred. **It is important to note that DSFMT does not specify what these events can be. Therefore, implementers of security protocols should specify clearly in their documentation what events they support and how the application can trigger them.** The `SecureSender/SecureReceiver` modules simply forward the buffer containing event information to the `SecureProtocol` modules.

In our implementation, there are three events supported on the sender side – `Member Leave`, `Member Join` and `Periodic Rekey`. The receiver side supports no events. All the three events cause rekeying, and the member leave and join events deny the new group key to the specified member(s).

**In DSFMT, rekeying occurs asynchronously with the data transmission.** It is recommended that every implementation of `ASecureProtocol` support this feature. In our implementation of `ManualProtocol`, for example, the sender side spawns a new thread for rekeying and instructs the receivers to do the same. The new threads then communicate the new group key to the receivers, even as data continues to be transmitted using the old key for encryption. Once the new key has been distributed and acknowledgements received, a flag is set on the
sender side indicating that the group key needs to be changed. The SecureSender component detects this by calling the isGroupKeyChanged method before encrypting data. This method returns true only when the new group key has been successfully communicated to the receivers. If it returns true, SecureSender calls the updateGroupKey method. This method replaces the old group key with the new one and instructs the receivers to do the same (at this point, they have already received it). Once done, the data transmission continues normally with the new key. Note that redistribution of keys can be used to expel members or add members. As we will show in Chapter 5, this technique for rekeying is quite robust and does not adversely affect the application’s data rate by a significant amount.

The gotControlMessage method is called by the SecureSender/SecureReceiver modules when they receive a control message over the network meant for the SecureProtocol component. Currently, this method serves no purpose on the sender side. It is used for rekeying notifications on the receiver side.

As usual, the virtual destructor is provided for cleanup.

In the above discussion, we hope to have explained the general ideas for implementation of this crucial interface. It is possible that this interface may need to be redefined as DSFMT evolves to support a variety of security protocols.

4.1.3 Network (Server/Client)

The Network component encapsulates the networking part of DSFMT. The application designer can implement this component to suit the application needs.

The Network component is defined by two interfaces, AServer on the sender side and AClient on the receiver side. These interfaces are given below.
class AServer
{
public:

    virtual void getConnected() = 0;
    virtual void sendMessage(const char *, int) = 0;
    virtual void sendMessageTo(int, const char *, int) = 0;
    virtual void receiveMessageFrom(int, char *, int *) = 0;
    virtual void noteFromApp(char *, int) = 0;
    virtual void getNoOfClients(int *) = 0;
    virtual void getIpList(char *) = 0;
};

class AClient
{
public:

    virtual void getConnected() = 0;
    virtual void sendMessage(const char *, int) = 0;
    virtual void receiveMessage(char *, int *) = 0;
    virtual void noteFromApp(char *, int) = 0;
};
It should be clear from the interfaces above that *AServer* and *AClient* are designed to provide the same functionality as would be provided by a high-level network library. We now briefly describe the classes *Server* and *Client*, which respectively implement the above interfaces.

The method *getConnected* is used by the *SecureSender/SecureReceiver* modules to inform the *Network* components to setup connections. The class *Server*'s constructor takes the number of receivers and their IP addresses as components and when *getConnected* is called, it sets up socket connections with each receiver (this implementation is unicast UDP). The *Client* object responds to *getConnected* by waiting to accept a connection from the sender, and then establishing it.

A call to *sendMessage* on the *Server* side results in sending the packet to all the receivers, while in *Client*, the packet is sent to the sender node. The *receiveMessage* method on the *Client* side is used by the *SecureReceiver* to receive messages from the sender side. The Server provides *sendMessageTo* and *receiveMessageFrom* methods for client specific communication, which is required particularly by the *SecureProtocol* component.

The *noteFromApp* is a hook method used when the application wants to bypass the *SecureSender/SecureReceiver* to communicate directly with the *Network*, as explained earlier. It is called by the *SecureSender/SecureReceiver* in response to the *notifyNet* call that the application makes. The buffer passed as argument can be used to pass any kind of message, as agreed upon by the application and the *Network* implementation. Our *Server* class currently does not use this call, but the *Client* uses it to allow the application to set its data socket to be blocking or non-blocking.

Finally, the *getNumberOfClients* and *getIPList* methods are used by the *SecureProtocol* to extract the respective information from the *Network* module.
4.1.4 Crypter

The Crypter module encapsulates the encryption/decryption functionality. It also has the responsibility of generating keys on demand and keeping track on the group key.

Crypter's interface is defined by the abstract class ACrypt, given below.

class ACrypt
{
public:

    virtual void encryptData(char *, int *) = 0;
    virtual void encrypt(char *, int *, HCRYPTKEY) = 0;
    virtual int decryptData(char *, int *) = 0;
    virtual void decrypt(char *, int *, HCRYPTKEY) = 0;
    virtual void setGroupKey(HCRYPTKEY) = 0;
    virtual HCRYPTKEY getExchangeKey() = 0;
    virtual HCRYPTKEY generateNewSymmetricKey() = 0;
    virtual HCRYPTKEY generateNewAsymmetricKey() = 0;
    virtual void exportExchangeKeyPair(char **, int *) = 0;
    virtual void exportPublicKey(char **, int *) = 0;
    virtual void exportSymmetricKey(HCRYPTKEY, char **, int *, HCRYPTKEY) = 0;
    virtual HCRYPTKEY importExchangeKey(char *, int) = 0;
}
virtual HCRYPTKEY importSymmetricKey(char *, int, HCRYPTKEY) = 0;

virtual void getSecurityLevel(int *) = 0;

virtual void setSecurityLevel(int) = 0;

};

We first mention two points worthy of note about the above interface:

- The interface is quite lengthy, compared to other modules
- It uses a Microsoft Windows CryptoAPI [MSDNcapi] data type called HCRYPTKEY, which is a handle to a key.

Since the current DSFMT implementation is for the Microsoft Windows operating system, the Crypter module is dependent on the Microsoft CryptoAPI functionality. Microsoft CryptoAPI is designed to provide a set of cryptographic services to applications. CryptoAPI exposes a kind of generic framework (not in the object oriented sense) for various cryptographic functions, like encryption, decryption and key generation. Specific implementations of algorithms are provided by Cryptographic Service Providers (CSPs).

Our implementation of the ACrypt component is called the Crypter, and uses the Microsoft Strong CSP [MSDNscsp] for Windows 2000. The ACrypt interface is generic enough to support any different CSP using the CryptoAPI. If a particular implementation does not wish to use CryptoAPI (say, if DSFMT is ported to a platform other than Microsoft Windows), the ACrypt interface should still be able to support it, provided the data type HCRYPTKEY is defined to be interpreted correctly as representing some sort of key handle. Therefore, we believe the ACrypt interface is generic enough to satisfy DSFMT design goals.

We now outline the purpose of the methods that comprise the ACrypt interface. In the following discussion, we have skipped some of the implementation details. The interested reader is referred to CryptoAPI documentation [MSDNcapi].
The first four methods in the interface deal with the most important functionality of the Crypter component – encrypting and decrypting data. These methods take in a buffer of bytes and its length as argument, and return the encrypted/decrypted data, with the new length. There are two versions – one that takes a key handle as an argument and another that does not. The latter version uses the group key, and is called by the SecureSender/SecureReceiver components for regular data encryption/decryption.

ACrypt provides a set of typical “set-get” methods for setting and retrieving the group key, security level, public key and so on. Our Crypter implementation supports 3 security levels, which map to 56-bit DES, 112-bit Triple DES and 168-bit Triple DES encryption algorithms respectively.

The two generate methods are used to generate new symmetric or asymmetric keys. CryptoAPI provides a special system for safe handling of keys and for their transfer over an insecure network. Keys can be encrypted and stored in a unintelligible sequence of bytes called Key Blobs. These Key Blobs can then be passed around, without fear of a malicious entity breaking into them. The export methods are used to create Key Blobs and the import methods are used to retrieve the keys from the blobs. This system is used by SecureProtocol implementations to securely exchange keys between the sender and receivers.

In this section, we have attempted to give an idea of what each DSFMT component provides and how it can be implemented to do its task. We have also described briefly our current implementations. However, we have so far not explained clearly how an application can use the DSFMT interface. This is the objective of the next section.
4.2 Application Model

In this section, we describe the DSFMT application model. We envisage two kinds of users of DSFMT – 1) developers who will use it as a framework and 2) developers who will use it as a kind of black box API. This section caters to the needs of the latter category of users, although the former too will find it useful.

DSFMT is aimed at the class of applications that need to securely stream real time content (or otherwise), possibly multicasting to a group of receivers. DSFMT takes over the burden of key management and encryption/decryption. However, the application developer is free to implement those components of DSFMT that he/she wishes to. The current implementations can also be extended through inheritance.

Those developers who need black-box security, on the other hand, can choose to use the implementations that we have provided. They would only need to understand the interfaces provided by the `SecureSender/SecureReceiver` components of DSFMT, since they will be interacting only with those interfaces.

The rest of this section is designed like an FAQ, and we attempt to answer the important questions regarding the DSFMT application model.

4.2.1 What does DSFMT provide?

DSFMT, as mentioned above, provides two main functions: 1) Key management and 2) Data security, when transmitted over the `Network`. The application developer can use the socket-like `SecureSender/SecureReceiver` interfaces to send and receive data securely over the `Network`. DSFMT currently makes no real time guarantees, but its overheads are quite minimal, as will be shown in Chapter 5. Further, DSFMT is a framework – its components can be extended, if required, through inheritance. For example, many application developers may choose to re-implement the `Network` components to suit their needs. Thus, DSFMT provides an ability to easily and flexibly incorporate security into streaming applications.
4.2.2 What does DSFMT NOT provide?

The first important thing that DSFMT does *not* provide is authentication. Applications on the sender and receiver sides are expected to perform authentication separately. The sending application then provides the `SecureSender` with a list of IP addresses of the receivers to whom the data should be securely streamed. However, we believe DSFMT can be extended in future to support authentication as well.

DSFMT does not make any claim towards reliability of data streaming, in terms of whether all packets are delivered to the receivers. This is because the `Networking` component can choose to be implemented using a variety of protocols, including unreliable UDP. However, the data that *is* delivered is guaranteed to be tamper-proof, as long as the `SecureProtocol` and `Crypter` modules are implemented correctly, or default implementations are used.

DSFMT currently does not make any real time guarantees. It is essentially a best effort system. Again, incorporating some kind of guarantees into DSFMT is a possible future work.

4.2.3 What all does an Application Developer need to know?

Although DSFMT can be used as some sort of black box API with the `SecureSender/SecureReceiver` interfaces, the application developer still needs to have some information about the implementation being used. For one, DSFMT does not standardize the list of rekeying events supported by the `SecureProtocol` module. The application developer needs to find what events are supported and how they can be triggered from the implementation documentation for that module. Second, the application developer should know what a particular security level maps to in the `Crypter` implementation. Finally, some specialized `Network` module implementations may need to communicate directly with the application (using the notification hook provided). Again, the module implementers are expected to provide documentation and the applications developers need to use it.
4.2.4 What is the Step-by-step Procedure of using DSFMT?

**Sender Side**

1) Create an instance of the class `SecureSender`.

2) Declare a variable of type `SessionParam` and fill its components with appropriate values (see the `Interface.h` header file for `SessionParam` definition).

3) Call the `setup` method with the `SessionParam` variable as argument.

4) `Interface.h` contains the declaration of two constants, called `MAX_PACKET_SIZE` and `MAX_MEMBERS`. Set these to appropriate values, as required for the session.

5) Use `secureSend` to send data to the specified receivers. The size of the buffer given as argument here should not exceed `MAX_PACKET_SIZE`.

6) To trigger a rekeying (or other significant) event, use the `eventOccurred` method. For details about arguments, refer to the documentation of `SecureProtocol` component being used.

7) To communicate directly with the `Network` module, use the `notifyNet` method.

8) When communication is over, call `delete` on `SecureSender` for proper cleanup.

**Receiver Side**

1) Create an instance of the class `SecureReceiver`, and call `setup` on it. At this point, DSFMT waits for and then sets up a secure connection with the sender.
2) Depending on whether you want the \textit{secureSend} call to be blocking or non-blocking, make a call to the method \textit{blocking} with a non-zero value or 0 as parameter, respectively. The default setting is blocking.

3) Set the DSFMT constants to appropriate values, as in step 4 above.

4) Use the \textit{secureReceive} method to receive data. Again, there is a restriction on the maximum buffer size, as mentioned in step 5 above.

5) When communication is over, call \textit{delete} on \textit{SecureReceiver} for proper cleanup.

This section has outlined the DSFMT application model. We again mention that the user has to read implementation specific README files and other documentation provided for details not covered here.

In this chapter, we covered the interfaces or DSFMT components and their current implementations in some detail. We also described the application model and how an application can be adapted to use DSFMT.

In the next chapter, we will describe a couple of sample applications and the results of some of the experiments we conducted to analyze overhead.
CHAPTER 5

Experiments and Results

This chapter is organized as follows. In section 5.1, we describe the experimental setup used to test the performance of the Distributed Security Framework for Multimedia Transmission (DSFMT). Section 5.2 presents some of the results.

5.1 Experimental Setup

In order to test DSFMT, we took two streaming applications and adapted them to use DSFMT for secure transmission. In Section 5.1.1, we describe these applications briefly. Section 5.1.2 deals with the hardware test bed used.

5.1.1 Test Applications

A fundamental design goal of DSFMT (see section 3.1.2 for DSFMT design goals) was to be able to integrate security into multimedia streaming applications without much difficulty. To assess whether this goal was satisfied, we used two complete streaming applications and then modified them to work with DSFMT. These applications are briefly described below.

Text Streaming Application – This application simply sends a stream of text, at a constant rate, to a set of receivers. The data rate is low, at about 15-20 packets every second, and each packet being only around 50-100 bytes. We used this application mainly to test the correctness of the implemented DSFMT modules, while they were being developed, redesigned and improved.
Video Streaming Application – This application streams MPEG-1 video to a set of receivers at 30 frames per second. The frame size varies, depending on whether it is an I, P or B frame, but the average frame size of the video we used (4dice.mpg) was around 6000 bytes. Due to the high date rate and real time requirements of video, this application was ideal to test DSFMT performance in terms of security overhead. The application was originally developed as part of the work described in [Gupta02]. It uses the Dynamic Soft Real Time CPU Scheduler [DSRT] for scheduling the network and display threads.

The effort involved in integration of these applications into DSFMT and the experimental results will be presented in section 5.2.

5.1.2 Test Bed

DSFMT was implemented and tested on Intel compatible PCs running the Microsoft Windows 2000 operating system. Table 5.1 gives details of the four PCs we used for testing.

<table>
<thead>
<tr>
<th>Name</th>
<th>Processor</th>
<th>Clock Speed</th>
<th>RAM size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>monet-bangalore</td>
<td>Mobile AMD Athlon 4</td>
<td>1 GHz</td>
<td>256</td>
</tr>
<tr>
<td>monet-chicago</td>
<td>Intel Pentium 3</td>
<td>700 MHz</td>
<td>128</td>
</tr>
<tr>
<td>monet-munich</td>
<td>Intel Pentium 3</td>
<td>600 MHz</td>
<td>128</td>
</tr>
<tr>
<td>monet-sydney</td>
<td>Intel Pentium 2</td>
<td>400 MHz</td>
<td>128</td>
</tr>
</tbody>
</table>

Table 5.1 DSFMT Test Bed – PC Configurations

Each PC was connected to a 100 Mbps Ethernet, which was the network used for streaming.

In all our experiments, the server ran on monet-bangalore, the fastest of the four PCs used. Clients ran on all four PCs.
5.2 Results

The experiments had two major goals:

- To access the effort involved in taking an existing data streaming application and adapting it to use DSFMT for secure streaming.
- To compute the encryption/decryption as well as key-switching overhead of DSFMT.

In Section 5.2.1, we describe how we modified the two streaming applications to work with DSFMT. Section 5.2.2 presents DSFMT overheads.

5.2.1 DSFMT-Application Integration Effort

In this section, we informally describe how we integrated the two test applications into the DSFMT framework. We do not think it appropriate to formally access the effort involved in terms of man-hours, as this would depend on factors like the programmer’s familiarity with the application, and complexity and design of the application itself. Rather, we wish to show how easy it is to follow the steps given in section 4.2.4 of Chapter 4 to modify an application to use DSFMT.

Text Streaming Application

As mentioned earlier, this application streams a sequence of ASCII characters over the network to set of receivers, at a relatively low data rate. The application was developed specifically to test DSFMT, so the integration process was quite simple. We first developed a stand-alone application, using TCP over IP to stream the text. Here is the procedure we then had to follow to integrate it with DSFMT.
1) Our first step was to identify the part of the application code that dealt with the networking. Since we already had an implementation of TCP for DSFMT's Network module, we removed most of the connection setup code (like creating sockets and binding them to ports) from the application, since DSFMT would take care of it automatically.

2) We then added code to instantiate the SecureSender and SecureReceiver objects on the sender and receiver side respectively. This code (in C++) is reproduced below.

**Sender Side**

```c++
#include "Interface.h"

#include "SecureSender.h"

...

SecureSender *secSender;

SessionParam sp;

char ipList[4*IP_SIZE];

...

strcpy(ipList, "128.174.247.233");

strcat(ipList, "128.174.247.223");

strcat(ipList, "128.174.247.241");

strcat(ipList, "128.174.247.193");

sp.sLevel = 1;

sp.noOfClients = 4;

sp.ipList = ipList;
```
secSender = new SecureSender();
secSender->setup(sp);

**Receiver Side**

#include “Interface.h”
#include “SecureReceiver.h”

... SecureReceiver *secReceiver;

... secReceiver = new secReceiver();
secReceiver->setup();

The code is quite self-explanatory. The header file `Interface.h` contains the abstract DSFMT interface descriptions. The files `SecureSender.h` and `SecureReceiver.h` contain the class descriptions for the `SecureSender` and `SecureReceiver` classes, which respectively implement the abstract interfaces `ASecureSender` and `ASecureReceiver`. On the sender side, we fill the `SessionParam` structure with the information needed to setup a secure connection – security level of the session, number of participating clients and their IP addresses. Security level 1 maps to 56-bit DES encryption in the current implementation of the `Crypter` module. On the receiver side, the code is much simpler, as we just need to call setup, and DSFMT automatically communicates session parameters (in particular, the security level) from the sender to the receiver.
3) The next thing was to set appropriate values to the two constants (defined in Interface.h), namely, MAX_PACKET_SIZE and MAX_MEMBERS. We set these to 50000 and 50 respectively.

4) The next task was to identify the code where data was actually sent and received. Since the application used Winsock (Windows Socket Library) sockets for communication, we simply needed to replace the send and receive function calls with the secSender->secureSend() and secReceiver->secureReceive() calls respectively. The arguments of the two functions are identical to their Winsock counterparts, so there was no need to modify the arguments.

5) Finally, we needed to identify where the application terminates the session. We added code to call the SecureSender and SecureReceiver destructors for cleanup.

   delete secSender; // sender side
   delete secReceiver; // receiver side

A couple of points worth mentioning here:

- The main reason why this integration was smooth and simple was that the DSFMT SecureSender and SecureReceiver functions for data transmission have a signature identical to the standard socket library functions.

- In this case, the original application used TCP for communication, and since DSFMT already had a TCP implementation for its Network module, things were simple. If, however, an application needs to use a different network protocol suite for which no implementation is currently provided in DSFMT, then the Network module would need to be reimplemented. Moreover, since integration into DSFMT requires delinking of networking code from the main application, some effort may be needed to achieve this, if there is tight coupling in the current application code.
• This application (and also the one we describe next) used the current DSFMT implementations of the SecureProtocol and Crypter modules. If an application wishes to reimplement or extend these implementations, it would take some additional effort.

• Finally, if an application wishes to specify rekeying events, it needs to include calls to SecureSender’s eventOccured function. The format of these calls depends on the specific implementation of SecureProtocol used, and can be found from the documentation that, DSFMT recommends, should accompany each implementation.

Video Streaming Application

This application retrieves MPEG-1 video from a stored file on the server side and streams frames at 30 frames per second to a set of receivers, which then display the video. The process of integration was similar to that described above for the Text Streaming Application. A few points, however, are noteworthy.

• This application was a real test of DSFMT’s design, since it was not developed to test DSFMT, unlike the Text Streaming Application mentioned earlier. However, notwithstanding this, we found the integration process to be largely similar.

• On the sender side, the application had one thread for retrieving data from the file and control, and one network thread per client for streaming. Since DSFMT manages group communication independent of the application, we modified this thread model to have only one network thread, which streams data through DSFMT. To the application, this appeared as if there were only one client, but since the list of clients was specified to DSFMT, the data was actually distributed to all the four clients. Note that if the application needs to be aware of individual clients and send control information separately to them, it can do so by setting up a separate control channel independent of DSFMT. DSFMT can then be used for securely sending actual frames. This, separate channel, however, was not necessary in our application.
In the next section, we present experimental results measuring DSFMT overhead.

### 5.2.2 DSFMT Overhead

An important consideration in providing security to multimedia applications is to minimize the security overhead. This is especially true when the applications have real time requirements, like video and audio.

In this section, we analyze two important overheads that DSFMT adds to the application:

- **Encryption/Decryption** – Every data packet that is sent through DSFMT is encrypted on the sender side and decrypted on the receiver side. Obviously, these operations take some time to perform, depending on the size of the packets.

- **Key Switching** – During the streaming session, if a rekeying event is triggering by the application, DSFMT *asynchronously* distributes the new group key to the receivers. Note that this in itself does not add any overhead to the streaming, since the streaming continues normally while the key distribution takes place in separate threads. However, once the key has been distributed, the sender triggers a *key update*, and all the receivers and the sender replace their current group key with the new key just acquired. During this short period while the key switch takes place, streaming has to be temporarily suspended, which is why it is an overhead.

Another important measurement is the **Key Distribution Time** for our implementation of the *SecureProtocol* component. Although this cannot be considered an overhead (since key distribution occurs asynchronously), it is still an important statistic, as the application should determine how well in advance a key distribution event would need to be triggered.
Encryption-Decryption Overhead

Tables 5.2, 5.3 and 5.4 present statistics taken on the slowest of the PCs we used, i.e.,  *monet-sydney*. The overhead was naturally lower on the other PCs, since the encryption/decryption operations took lesser CPU time. The encryption and decryption times are averaged over 50 trials. These statistics pertain to the current implementation of the Crypter module, which uses the Microsoft CryptoAPI [MSDNcapi] and the Microsoft Strong Cryptographic Service Provider [MSDNscsp].

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Avg. Encryption Time (ms)</th>
<th>Avg. Decryption Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td>0.36821</td>
<td>0.37086</td>
</tr>
<tr>
<td>5000</td>
<td>0.69896</td>
<td>0.70662</td>
</tr>
<tr>
<td>7500</td>
<td>1.03116</td>
<td>1.03929</td>
</tr>
<tr>
<td>10000</td>
<td>1.36306</td>
<td>1.37914</td>
</tr>
</tbody>
</table>

**Table 5.2** Encryption/Decryption overhead for 56-bit DES

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Avg. Encryption Time (ms)</th>
<th>Avg. Decryption Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td>0.93979</td>
<td>0.94106</td>
</tr>
<tr>
<td>5000</td>
<td>1.82889</td>
<td>1.83652</td>
</tr>
<tr>
<td>7500</td>
<td>2.72652</td>
<td>2.71687</td>
</tr>
<tr>
<td>10000</td>
<td>3.64783</td>
<td>3.80544</td>
</tr>
</tbody>
</table>

**Table 5.3** Encryption/Decryption overhead for 112-bit Triple DES

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Avg. Encryption Time (ms)</th>
<th>Avg. Decryption Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td>0.99559</td>
<td>0.99264</td>
</tr>
<tr>
<td>5000</td>
<td>1.85147</td>
<td>1.84766</td>
</tr>
<tr>
<td>7500</td>
<td>2.74788</td>
<td>2.74268</td>
</tr>
<tr>
<td>10000</td>
<td>3.96411</td>
<td>3.97366</td>
</tr>
</tbody>
</table>

**Table 5.4** Encryption/Decryption overhead for 168-bit Triple DES

The tables give the average encryption and decryption times for four different packet sizes and three different encryption algorithms. The average size of an
MPEG-1 video frame varies between around 2500 bytes to 10000 bytes in our experience, and in particular, for the experimental file 4dice.mpg, the average was close to 6000 bytes. This is why we chose packet sizes in the range 2500 to 10000 bytes for our experiments.

From the tables, we can see that even for a 10000-byte packet encrypted with the strong (and practically unbreakable) 168-bit Triple DES algorithm, the average encryption and decryption times on a 400 MHz Pentium 2 PC were no greater than 4 milliseconds each. Now, a video streaming application like the one we used for testing usually has a maximum frame rate of 30 frames per second, or a period of about 33 milliseconds. For this period, the 4-millisecond overhead does not have any appreciable affect on the scheduling. In our tests, we easily achieved the 30 frames per second data rate, even using the 168-bit Triple DES encryption.

It is important to note the following two points:

- The processing power of today’s PCs is sufficient to provide a high level of security to multimedia applications which stream normal quality video at the reasonably high frame rate of 30 frames per second.

- MPEG-1 encoded High definition TV (HDTV) video frames, however, have a typical average size of about 80000 bytes. The average encryption/decryption times for a buffer of this size vary from about 11 milliseconds for 56-bit DES to about 30 milliseconds for 168-bit Triple DES, on the same PC, metropolitan-sydney. Therefore, the overhead on current PCs would be quite substantial for encrypting/decrypting HDTV quality video, and it may be not feasible to provide strong encryption without sacrificing frame rate. However, we believe with faster processors and/or implementation of encryption in specialized hardware, it may be possible to provide reasonably high security to high quality video.
Key Switch Overhead

The next aspect of DSFMT overhead that we present is that due to rekeying. Rekeying may be triggered by the application and carried out by the SecureProtocol module for a number of reasons, depending on the events supported by the particular implementation of SecureProtocol. For example, our implementations support events like Member Leave, Member Join and Periodic ReKey.

When such an event occurs, the SecureProtocol module on the sender side sets up a separate thread and distributes the new key to the receivers, where again there is a separate thread setup for key exchange. Since redistribution of keys occurs asynchronously, the time taken to distribute the new key cannot be considered an overhead. What is an overhead, however, is the time taken to perform the switch to the new key, once all the receivers have received it. In DSFMT, once the SecureSender finds that a new key has been distributed, it sends a message to the SecureReceivers telling them that the frames from that point of time onwards will be encrypted with the new key. It then performs the key switch and continues transmitting frames, encrypted with the new key. Before the receivers can decrypt these frames, they will need to have completed the key update.

Table 5.5 gives the average key switching time on the receivers, as measured on the four PCs used in our experiments. The delay was averaged over 15 rekeying events. Note that monet-bangalore also ran the server.

<table>
<thead>
<tr>
<th>PC</th>
<th>Delay (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>monet-bangalore</td>
<td>30.5803</td>
</tr>
<tr>
<td>monet-chicago</td>
<td>30.0441</td>
</tr>
<tr>
<td>monet-munich</td>
<td>30.9017</td>
</tr>
<tr>
<td>money-sydney</td>
<td>37.9857</td>
</tr>
</tbody>
</table>

Table 5.5 Average Key Switch Delay
We notice from the values that the key switch delay was in the range of 30-40 milliseconds. For an application with a period of 33 milliseconds, this delay may temporarily affect the period by a small amount. The frame rate may drop from 30 frames per second to 29 or 28 frames per second for a couple of seconds. In our tests, this did not lead to any perceptible difference in video quality on the receiver side for the video streaming application.

We believe that these results indicate that the design of DSFMT, involving asynchronously rekeying and synchronous key update, successfully makes it possible to perform rekeying without an appreciable change in the application's data rate. No matter what security protocol or kind of network is used, the key switching delay will remain the same.

**Key Distribution Time**

As mentioned above, the time taken to distribute keys is not considered an overhead, since data transmission continues in parallel while the key distribution takes place. However, this is still an important consideration, because the application should be able to trigger a key distribution event sufficiently in advance to ensure that the clients have received the new key by the time the key update is to take place. The application, therefore, should know how long it takes to distribute keys. Unfortunately, it is difficult to provide a bound on key distribution time, since it depends on a number of factors like security protocol used, number of receivers, distribution of these receivers on the network, what kind of network (for example, 100 Mbps LAN or Internet), bandwidth, congestion, transport protocol used etc. It also depends on whether unicast or multicast communication is used. Providing a bound on key distribution time is a challenging possibility for future work.

We now present some key distribution time measurements for our video streaming application. Since our Network implementation uses unicast UDP communication, these experiments use the Manual Key Distribution protocol for initial distribution of keys and rekeying events.
<table>
<thead>
<tr>
<th>Number of clients</th>
<th>Key Distribution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.301</td>
</tr>
<tr>
<td>2</td>
<td>25.3274</td>
</tr>
<tr>
<td>3</td>
<td>30.7099</td>
</tr>
<tr>
<td>4</td>
<td>34.1955</td>
</tr>
</tbody>
</table>

**Table 5.6**  
Average Key Distribution Times

Table 5.6 gives the key distribution times averaged over 5 rekeying events, for a set of 1-4 clients connected using a 100 Mbps LAN. We notice that the key distribution time increases almost linearly with number of clients. This is because, with unicast-multipoint communication, key distribution is not as efficient as with multicast, where a number of efficient protocols can be deployed.

In this chapter, we have presented the results of experiments we conducted on DSFMT, in terms of measuring both application integration effort and overhead of providing security. The next chapter concludes this thesis and identifies scope for future work.
CHAPTER 6

Conclusions and Future Work

6.1 Conclusions

In Chapter 1, we had laid out the goals of the Distributed Security Framework for Multimedia Transmission (DSFMT). We now look back at the main goals and see if we achieved them.

- **Ease of integration of security into streaming applications** – A fundamental goal of DSFMT was to be able to take a multimedia streaming application and easily modify it to add security, in terms of secure data transmission over the network. In section 5.2.1 of Chapter 5, we have shown the steps we followed in order to integrate two streaming applications into DSFMT. We have described how simple this process was, especially because of the standard socket-like interface DSFMT exposes to the application by means of the SecureSender module. The application does not need to concern itself with the details of other DSFMT modules. This design ensures smooth integration.

- **Flexible framework for implementing security protocols** – DSFMT is meant to be a framework in the object-oriented sense. Each module of DSFMT performs a specific task, and is decoupled from other modules to the extent possible. Although DSFMT may mature with use, we believe it is flexible enough to allow applications to work with a variety of security protocols, encryption algorithms and network protocol suites – all this without any substantial change in the application code itself.
• **Low security overhead** – In section 5.2.2 of Chapter 5, we present the two overheads that DSFMT adds to the data streaming part of the application – encryption/decryption overhead and key switch overhead. From the results, we can conclude that the overheads of our current implementations are small enough to cause only an intangible affect on the real time requirements of multimedia streaming applications.

We therefore believe DSFMT provides an effective solution to the various problems relating to integrating data-transmission security into multimedia streaming applications.

### 6.2 Future Work

An interesting characteristic of useful frameworks, as explained in section 2.2 of Chapter 2, is that they evolve continuously. No framework has been as useful in its first couple of iterations as it has been after maturing over a period of time. We therefore expect DSFMT too to evolve as more and more applications begin to use it. We present below some specific areas where there is potential for future work.

• **Multicast implementation of Network** – The Network module is DSFMT currently has only unicast implementations. This does not allow us to test and compare the performance of various secure multicast protocols, especially the two that we have currently implemented. The first thing that DSFMT probably needs is a multicast implementation of the Network module, say, using IP multicast.

• **More implementations of SecureProtocol and Crypter** – We have currently implemented two secure multicast protocols for the SecureProtocol module and have one implementation for Crypter using Microsoft CryptoAPI and the Microsoft Strong Cryptographic Service Provider. It would be useful to
have more implementations of these modules (possibly a hierarchy of implementation classes) to test the flexibility of DSFMT more thoroughly.

- **Extending DSFMT for other multimedia security considerations** – DSFMT currently is designed to provide data transmission security to multimedia streaming applications. However, the applications still need to take care of other security aspects like authentication of clients and watermarking of video. It would be interesting to explore if DSFMT can be extended to provide a flexible way to include these and other security responsibilities.

- **Exploring architectural possibilities** – As explained in Chapter 3, DSFMT was designed to work with various application architectures, in terms of distributing security responsibility among the participating nodes on the network. It would be of interest to actually try out a variety of architectures and see if DSFMT requires any adaptation to support some schemes.

We believe that with time and use, more possibilities for future work will arise.

In this chapter, we have examined to what extent the DSFMT design met its original goals, and the scope for potential future work relating to this thesis. In conclusion, we hope DSFMT will be useful to multimedia streaming applications of the future as a complete, flexible and effective security solution.
REFERENCES


[SmuG] IRTF Secure Multicast Research Group
http://www.ipmulticast.com/community/smug/

[GSEC] IRTF Group Security Research Group
http://www.securemulticast.org/gsec-index.htm

[MSEC] IETF Secure Multicast Research Group
http://www.securemulticast.org/msec-index.htm


