GROUP-ORIENTED SECRET SHARING USING SHAMIR'S ALGORITHM

by

KALYAN KOUSHIK ALAPATI

A thesis submitted to the

Graduate School-Camden

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of Master of Science

Graduate Program in Scientific Computing

Written under the direction of

Dr. Jean Camille Birget

And approved by

_____

Dr. Jean Camille Birget

_____

Dr. Sunil Shende

_____

Dr. Suneetha Ramaswami

Camden, New Jersey

January 2018

THESIS ABSTRACT

Group-Oriented Secret Sharing

using Shamir's Algorithm

by KALYAN KOUSHIK ALAPATI

Thesis Director**:**
Dr. Jean Camille Birget

In the current state of highly distributed and hybrid-cloud systems environment, managing and securing enterprise or government systems/data requires effective access control techniques and protocols. Currently, individual and independent logins using single or multi-factor passwords are widely used across the industry, but they are highly vulnerable to hacking, phishing and various password stealth techniques.

For securing highly sensitive IT assets, comprehensive data management and governance programs include group-oriented login or authorization procedures, wherein a group of individuals or processes (as opposed to a single individual) provide their credentials/passwords or keys to gain access to the sensitive resource.

To implement the group-oriented login, a widely acclaimed cryptographic technique, *Secret Sharing*, offers an elegant and secure solution. In this technique, the secret (password) is divided into multiple shares in such a way that a threshold number of shares are essential to reconstruct the secret (password). *Shamir's Secret Sharing* uses this cryptographic technique, and the Secret Share splitting and reconstruction are based on a polynomial over a finite field. The goal of this thesis is to study and evaluate this technique with reference to threshold based group-login by various examples.

Acknowledgments

I would like to express my sincere appreciation to Dr. Jean Camille Birget for his guidance in my thesis work without whom this thesis would not have been possible.

TABLE OF CONTENTS

Chapter 1 : Introduction

In the current digital era, collection of information for efficiency and productivity gains has become an essential and integral part of every government, business, education and research organization. Information security and assurance have become even more critical in maintaining integrity of the information, safe guarding and providing access to the rightful owners. Organizations practice comprehensive safeguarding policies ranging from auditing, job rotation and separation of responsibilities to protect the data center and networks from undesirable acts either by a malicious or inexperienced employee acting alone.

For example, multi-party authorization process is used extensively across the industry to safeguard and protect telecommunications networks, data centers and industrial control systems. In this process, a second (or additional) authorized user approval is needed before the action actually takes place. While the implementations vary radically from system to system, at its core, the authorization process involves a group of individual users authorizing access to a resource. While the authorization is handled by this process, the resource authentication (login) is still a single unit and known to one party, which makes it highly vulnerable to various hacking techniques.

A simpler and more elegant solution can be implemented using Adi Shamir's Secret Sharing[1] cryptographic technique, using which the resource authentication (login) is split into multiple shares across a given group in such a way that the secret (login) can be reconstructed only when a predefined (threshold) number of shares are available from the group. When this technique is property implemented, either an individual share or shares

less than the threshold number of shares are of no use on their own and will not be able to reconstruct the secret (password).

Chapter 2 of this thesis explains the theory of Shamir's Secret Sharing scheme, its vulnerabilities, fixes proposed by various researchers and alternative schemes. Chapter 3 describes typical examples of group authorization, Chapter 4 provides the details of the flow charts and Java application developed to illustrate and study the limitations of Shamir's Scheme. Chapters 5 and 6 provide the compilation of the test cases, results and conclusions. Details of all the application Java modules, classes and the code used for conducting the studies in this thesis are provided in Appendix 1, and details of the CPU and JAVA SDK versions are listed in Appendix 2.

Chapter 2 : Secret Sharing

Suppose two friends rent a locker to securely store their valuables and there is only one key to access the locker. Now arises the question of trust: what if the friends do not really trust each other and are afraid that other might access the locker and take everything? There needs to be a solution to ensure that their valuables are secure.

An easy way to solve this issue is to design the locker in such a way that at least two keys are needed to access its contents. Now each person is given a key. If any one of the friends wants to access the locker, the other person must also provide his key and this solves the problem of trust. This explains the basic concept of secret sharing.

In cryptography, secret sharing is a method of distributing a secret among a group of participants each of which is allocated a share. The secret can be reconstructed only when a certain specified number of shares are combined.

2.1 Shamir's Secret Sharing

Shamir's secret sharing scheme is a threshold scheme which is based on polynomial interpolation. It has two parameters: *t*, the threshold and *n*, the number of participants/players. The main idea of the scheme is that *t* points are sufficient to define a polynomial of degree $t-1$, for example 2 points are sufficient to define a line, 3 points can define a parabola and so on. Similarly, using this scheme, a dealer *D* splits a secret s into shares and distributes them to n players such that at least t shares are needed to reconstruct the secret s and any fewer than t players cannot learn anything about the secret. Such a scheme is termed as (t, n) threshold scheme.

2.1.1 Splitting the Secret

According to the Shamir's secret sharing scheme, a dealer *D* distributes a secret *s* among *n* players $\{P_1, P_2, P_3, \ldots\ldots, P_n\}$ such that at least *t* players are required to reconstruct the secret and *t* should be less than or equal to n i.e. $1 \leq t \leq n$.

To split the secret into shares the dealer *D* creates a polynomial $f(x)$ of degree $t-1$ and a constant term a_0.

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots . + a_{t-1} x^{t-1} \ (mod\ p)$$

where p is a prime number selected based on the level of security needed for the secret, and the constant term a_0 is the secret s. Higher values of *p* result in greater security, and the secret s is always less than the prime number p, and typically more than *n*.

Randomly choose the $t-1$ integer values $a_1, a_2, \ldots\ldots, a_{t-1}$ such that $a_i \in [0, p)$ for all *i*. Then the dealer *D* chooses *n* random distinct evaluation points $x_i \neq 0$ and secretly distributes to each player P_i the share

$$Share_i(s) = \big(x_i, f(x_i)\big)$$

Figure 2.1 below provides the process details of group creation, prime number selection, secret generation, Shamir's Secret split, and the secret share distribution by dealer *D*.

Figure 2.1 Shamir's Secret Sharing – Secret Generation & Share Distribution

Create a Group to Share a Secret

- select group size ***n***
- select the number of group members required to reconstruct the secret, threshold ***t***

**details of the group management features are explained in the next chapter*

Select a Prime Number

- length of ***p*** is based on the **security level required**
- select minimum number for secret ***m***

Generate The Secret

- secret is a random number between ***m*** and ***p-1***

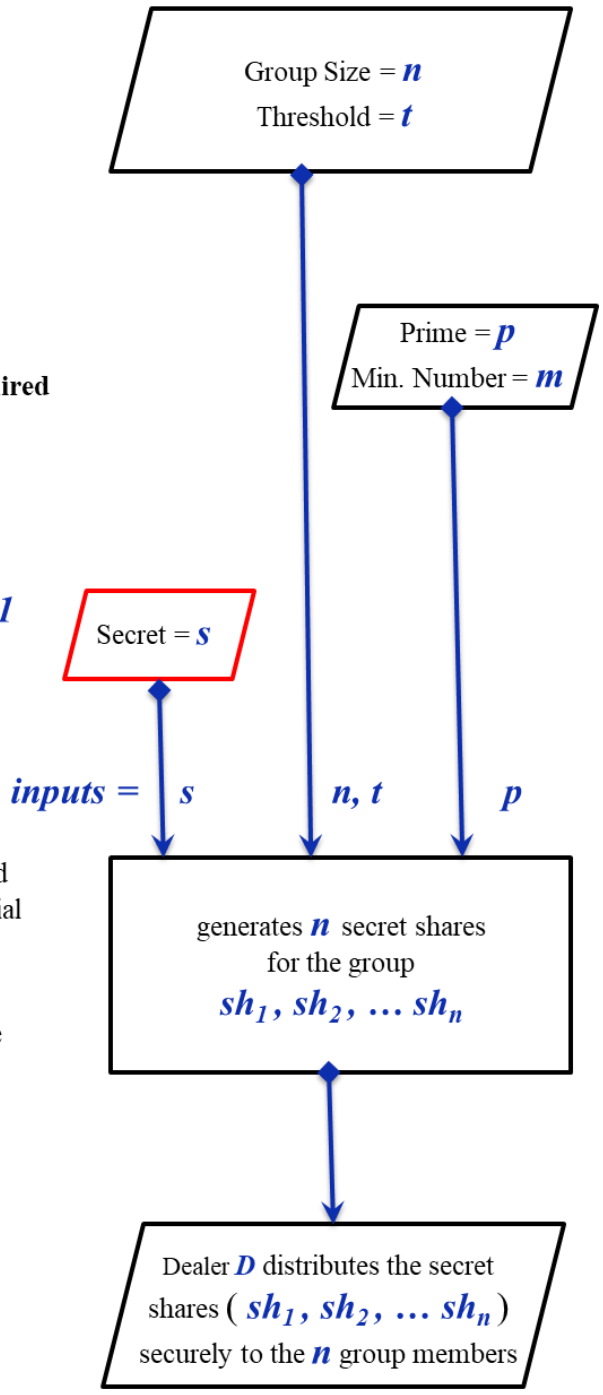
Shamir's Secret Split

- pass the input ***n, t, p, s*** to ***secretSplit*** method
- ***secretSplit*** method builds the unique polynomial ***f (x)*** with the inputs above
- and the method selects ***n*** random distinct evaluation points on the polynomial, which are secret shares $sh_1, sh_2, \ldots sh_n$ where $sh_i = Share_i\ (s)$

Dealer

- distributes the secret shares to the group
- method builds the polynomial and generates ***n*** shares , $sh_1, sh_2, \ldots sh_n$

**functions of the dealer are explained in the next chapter*



2.1.2 Reconstructing the Secret

Goal here is to reconstruct the secret by considering any subset of *t* shares out of *n* shares.

We will mark the subsets to be

$$\left(x_0, f\left(x_{i_0}\right)\right), \left(x_1, f\left(x_{i_1}\right)\right), \left(x_2, f\left(x_{i_2}\right)\right), \ldots\ldots, \left(x_{t-1}, f\left(x_{i_{t-1}}\right)\right).$$

Lagrange interpolation is used to compute the unique polynomial $f(x)$ of degree $\leq t-1$ from the t shares. In this process, the known data points $(x_0, f(x_0))$, $(x_1, f(x_1))$, …,$(x_{t-1}, f(x_{t-1}))$ with all x_i different, are used to re-construct the unique polynomial $f(x)$ that passes exactly through these data points.

Using Lagrange Interpolation formula, the polynomial $f(x)$ can be written in the form

$$f(x) = \sum_{i=0}^{t-1} f(x_i) * L_i(x),$$

where $L_i(x)$ is the Lagrange Polynomial.

$$L_i(x) = \prod_{j=0\ j\neq i}^{t-1} \frac{x - x_j}{x_i - x_j}.$$

$L_i(x)$ has value 1 at x_i, and 0 at every other x_j.

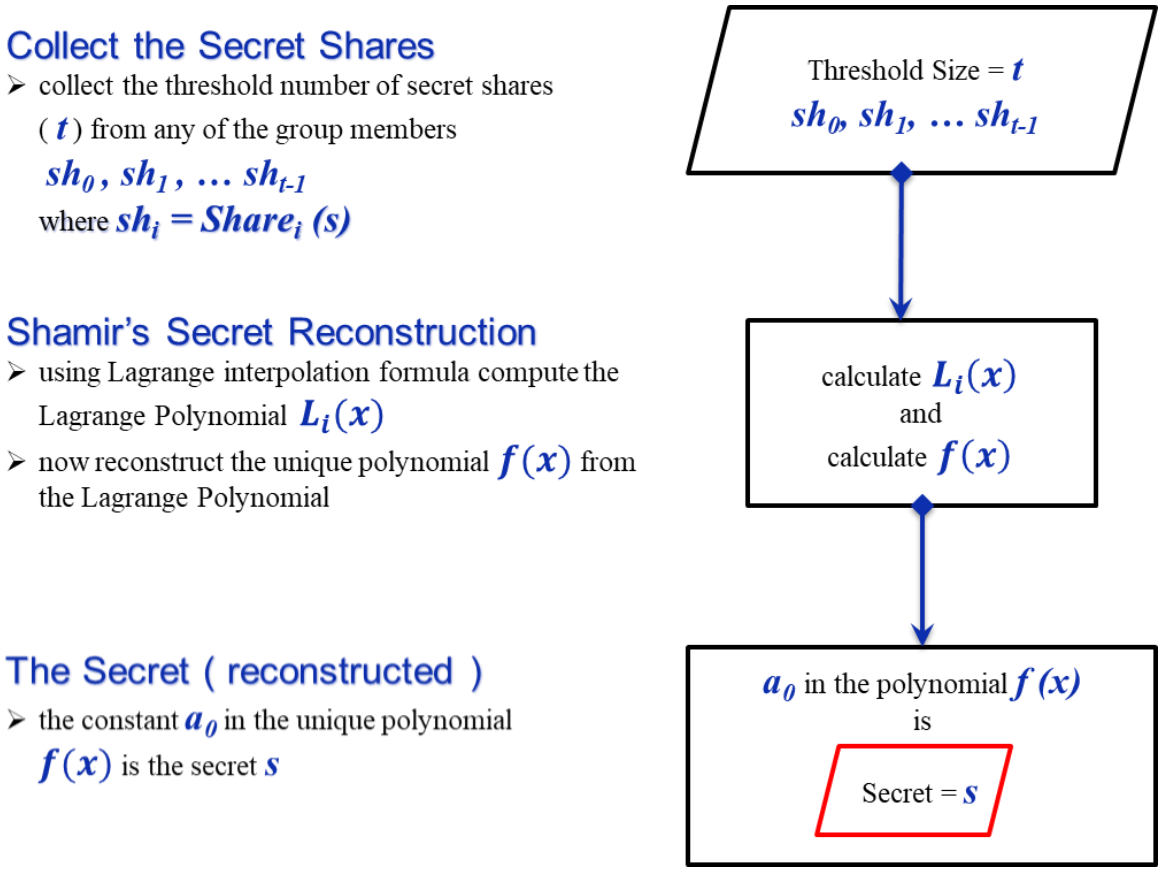
The reconstructed unique polynomial $f(x)$ is

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots . + a_{t-1} x^{t-1}$$

The constant term a_0 in the reconstructed unique polynomial $f(x)$ is the original secret s.

Figure 2.2 below provides the process details of collecting the secret shares, calculation of Lagrange Polynomial, the unique polynomial corresponding to the original secret, and finding the original secret.

Figure 2.2 Shamir's Secret Sharing – Secret Reconstruction



Algorithm 1 below lists the pseudo-code for key functions of the Shamir's Scheme.

- SECRET-SPLIT Method: (refer Figure 2.1) Builds the unique polynomial and generates the secret shares.
- SECRET-RECONSTRUCT Method: (refer Figure 2.2) calculates the Lagrange Polynomial and reconstructs the unique polynomial and derives the secret.
- LAGRANGE-INTERPOLATION Method: (refer Figure 2.2) uses Lagrange interpolation formula to build Lagrange Polynomial.

Algorithm 1: Shamir's Secret Sharing Scheme

START

1. *n* ← positive integer (group size), *t* ← positive integer (threshold value)
2. *p* ← bigint() /*prime number*/
3. m ← positive integer /*minimum number for the secret (typically > *n*)*/
4. *s* ← bigint /*random number between *m* and *p (m > s < p)*/
5. *shares* [n] [2] ← bigint /*a 2d array to store the calculated shares*/
6. *tshares* [*t*] [2] ← bigint /*a 2d array to store *t* shares for reconstruction*/

/* Functions */

7. SECRET-SPLIT (*n, t, s, p*)
8. SECRET-RECONSTRUCT (*tshares [], p, t*)

SECRET-SPLIT (*n, t, s, p*)

1. **for** i ← 0 to *t-1* **do**
2. coeff [i] ← random number from 1 to *p*
3. **end for**
4. **for** *x* ← 1 to *n* **do**
5. *share* ← *s*
6. **for** *expo* ← 0 to *t-1* **do**
7. *share* ← (*share*+ (coeff [***expo***]*****(x^{expo} %*p*)) %*p) %p*
8. **end for**
9. *shares [x-1][0] = x, shares[x-1][1] = share*
10. **end for**
11. **return** *shares*

SECRET-RECONSTRUCT (*tshares [], p, t)*

1. **if** length(*tshares []*) < *t* **then**
2. **print** reconstruction not possible
3. **else**
4. **for** i ← 0 to length (*tshares []*) **do**
5. *xarray [i]* ← *tshares[i][0]*
6. *yarray [i]* ← *tshares [i][1]*
7. **end for**
8. LAGRANGE-INTERPOLATION (*xarray [], yarray [], p*)
9. **print** *secret*
10. **end if**

LAGRANGE-INTERPOLATION (*xarray [], yarray [], p*)

1. *secret* ← 0, sum ← 0
2. **for** i ← 0 to *length (yarray [])* **do**

```
3.      product ← yarray [i]
4.      for j ← 0 to length (yarray []) do
5.              if  i != j then
6.                      product ← product * (x – xarray [i]) / (xarray [i] - xarray [j])
7.              end if
8.      end for
9.      sum ← sum + product
10. end for
11. secret ← sum
12. return secret
END
```

The group management, share distribution and collection features necessary for implementing Shamir's Scheme are explained in Chapter 3.

2.2 Secret Sharing – Vulnerabilities, Fixes and Alternatives

Over the years, many fixes have been proposed to address the exploitable vulnerabilities in Shamir's technique, and some proposed alternatives, enhancements to strengthen the secret sharing algorithm.

For example Tompa and Woll [3] demonstrated the vulnerability of Shamir's scheme when shares are distributed to one or more dishonest shareholders, and the shares are revealed asynchronously. Tompa and Woll suggested a small modification to address the issue, and the fix involves the distributor of the shares to sign each share with unforgeable signature[7] prior to the distribution of the shares. This process retains the security and efficiency of Shamir's scheme, but addresses the issue of cheating amongst the shareholders.

Tartary and Wang[6] addressed the problem of communications eavesdropping when the secret shares are transmitted over insecure networks either during distribution or reconstruction of the shares. The proposed fix involves a computationally secure approach

which would modify the threshold dynamically based on the capabilities of the eavesdropper (hacker).

Further, Rabin[8] introduced *information dispersal* scheme that would cut down the size of each secret share, making it space optimal. However, this scheme assumes that the shareholders are honest. To address this issue, Krawczyk[4] proposed a solution that combines the information dispersal with secure encryption of the secret shares.

Chapter 3 : Secret Sharing – Practical Applications

Shamir's Secret Sharing scheme provides elegant solutions to many situations, and few applications incorporating Shamir's scheme (group authentication and group authorization) are discussed below.

For the sake of clarity, here is a quick description of the terms Authentication and Authorization used in this thesis.

Authentication (also referred to as login) is the process of verifying the identity of an individual or a resource using a trusted authority. Or, simply stated, it is the process of verifying that "you are who you say you are". Authorization is the process of verifying that the identified individual (or resource) is allowed to do what that individual is trying to do.

Section 3.1 below provides the details of group authentication using Secret Sharing, wherein members of a group authenticate each other simultaneously.

And, Section 3.2 provides details of group authorization, wherein a subset of group members (threshold group) already authenticated by another authority, authorize an action of data retrieval, by pooling their secret shares together to reconstruct login credentials of the sensitive data cluster, which in turn are used by a proxy server to login to the cluster, and perform the required data retrieval action.

3.1 Group Authentication using Secret Sharing

In general, the user password based, public or private key based authentication schemes belong to one-to-one category of authentication protocol. In this protocol, to establish identity, the user (prover) interacts with the verification authority (verifier), and communicates the required credentials. Current day network applications are much more

complex, and the communications go beyond one-to-one (unicast) and one-to-many (multicast)[9].

Note: *In the next few paragraphs of this section* 3.1, *I will be citing an example and a novel idea proposed by L. Harn[9], which stands out as a relevant example for group authentication. However, this technique is not used either in the program development or other discussions in this thesis.*

L. Harn[9] proposed group authentication specifically designed for group-oriented applications that need simultaneous authentication of multiple users. The proposed group authentication schemes (*t,m,n*; where *t* is the threshold of the proposed scheme, *m* is the number of users participating in a group-oriented application, and *n* is the total number of group members; $t \leq m \leq n$), are based on Shamir's Secret Sharing scheme *(t, n)* using polynomial operations in finite space, and another scheme proposes the use of tokens obtained from the Group Manager (GM). In group authentication process, the GM initially registers all group members, and at that time uses Shamir's Secret Sharing Scheme to issue secret shares (tokens) to each group member. Subsequently, the group members interact together to authenticate each other without GM's assistance. The proposed Group Authentication Scheme (GAS) protocol is non-interactive, and the basic scheme works only for synchronous communications, and the advanced scheme caters to asynchronous communications.

In this scheme, the threshold value *t* is a very important parameter that directly relates to the security level required for the group authentication. The *(t, n)* based Secret Sharing scheme can provide safeguard against the collusion of up to *t - 1* inside attackers, who might collude to derive the unique polynomial *f (x)*, and forge any number of valid secret shares (tokens). As part of the group administration, GM will be issuing a new token only

when a new member joins the group. When a member leaves the group, the GM assumes that member's token is compromised, and indicates the member's departure from the group to the remaining group members. In this process, GM keeps a clear count of the members departing the group, and when that number reaches the threshold value for the group (*t*), GM will generate and issue new tokens to all the members remaining in the group.

With limited communication overhead, this scheme is very efficient and every participant needs to broadcast a value (secret share/key) to all other participants only once, and every participant needs to compute the polynomial operations once. In contrast, the conventional user authentication techniques authenticate one user at a time, whereas the GAS scheme authenticates all users at once.

3.2 Group Authorization using Secret Sharing

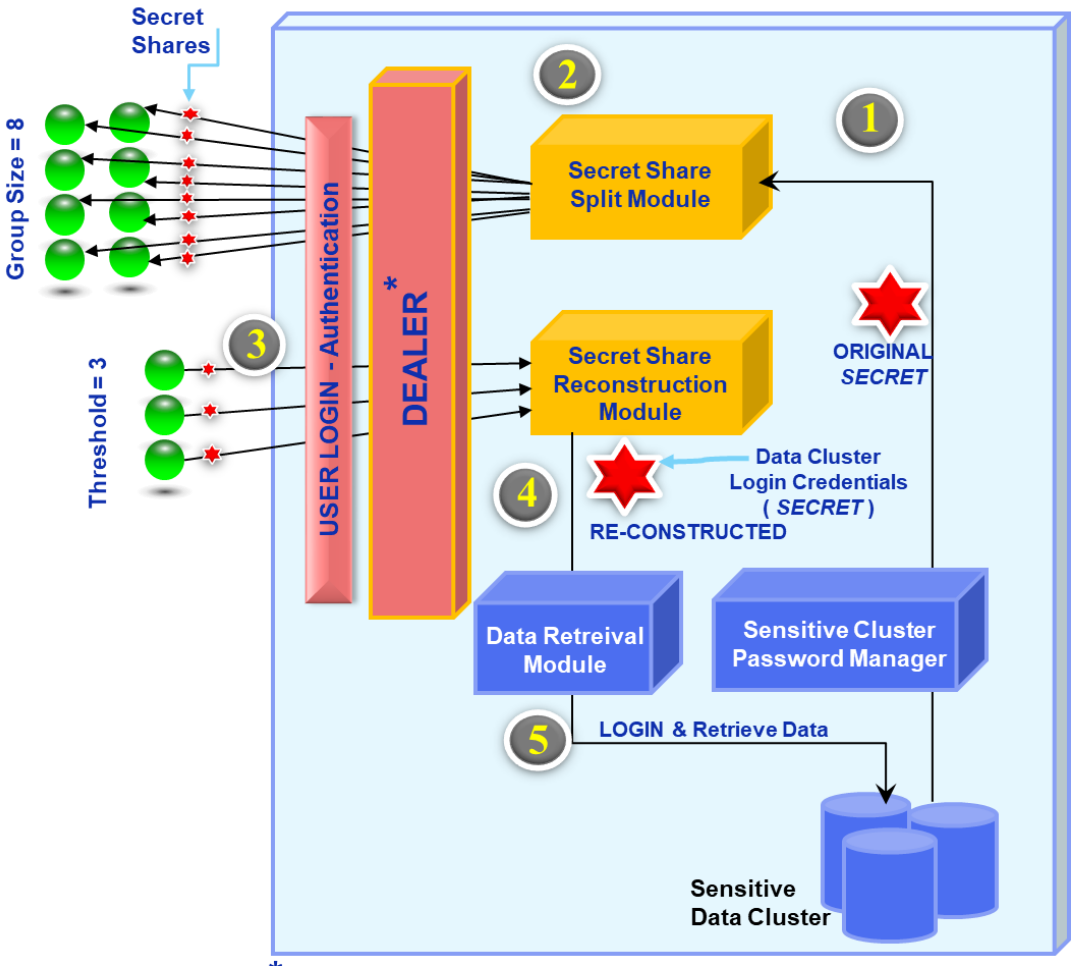
Shamir's Secret Sharing scheme also provides elegant solutions to situations demanding group authorization and separation of responsibilities, wherein login credentials required to access to a sensitive resource need to be secured, but storing the credentials either at a single location (server, cloud, database etc.) or sharing with any single individual results in a security compromise.

For example, joint venture projects working on sensitive product designs usually involve either multiple divisions in the enterprise or may span across multiple organizations. Login access to the server hosting the product designs, cannot be either stored in a single location or shared with an individual (system or database administrators, project managers etc.). Splitting the server login credentials (ID or Password or Digital Key etc.) using the Secret Sharing scheme and distributing the secret shares across the authorized group of individuals constituted from each division, provides a secure and effective solution.

Consider a group of engineers (belonging to various companies) supporting a shared network cluster bridging communications across the world, and no one engineer can be given login credentials to the network cluster. Secret Sharing technique can provide an effective solution in this situation without exposing and compromising on the security of the network cluster login credentials.

Given below in *Figure 3.1*, is a typical implementation of *Secret Sharing* scheme to safeguard login credentials of a sensitive data cluster storing information from various business units in an enterprise with worldwide operations. For obvious security reasons, the login credentials (*SECRET*) can neither be stored nor shared with database or network administrators. The login credentials can be used transiently either on-demand or when needed. As indicated in the diagram, the *Secret Sharing* scheme for this case provides a simple, effective and elegant solution.

Figure 3.1 Typical Application of
Group Authorization to Access Sensitive Data Cluster
Using Secret Sharing Scheme



*Detailed functions of Dealer are listed in *Figure 3.3*

1. Initial handshake to retrieve the secure login credentials of the data cluster (*SECRET*)
2. Split the *SECRET* into *Secret Shares* and distribute amongst the group participants (group size = 8)
3. Obtain the *Secret Shares* from any 3 (threshold size = 3) of the group participants
4. Reconstruct the *SECRET* and pass-on to a data retrieval module
5. Data Retrieval Module uses the *SECRET* (login credentials) to log-in to the Sensitive Data Cluster and performs the required action

3.3 Group Administration and Password Management

Group administration and Password management are integral part of any application that incorporates Shamir's Secret Sharing scheme discussed in the sections above.

Dealer Authority (DA) provides the group administration tasks: group creation, membership management, distribution of secret shares to the group, collection of shares from the threshold group, and group destruction.

Password Authority (PA) generates the secret for the required level of security, and provides the handshake to the system requesting the password. Further, after reconstructing the password using Shamir's Scheme, and prior to sending the password, PA also verifies the hash value of the password originally calculated and stored for the group.

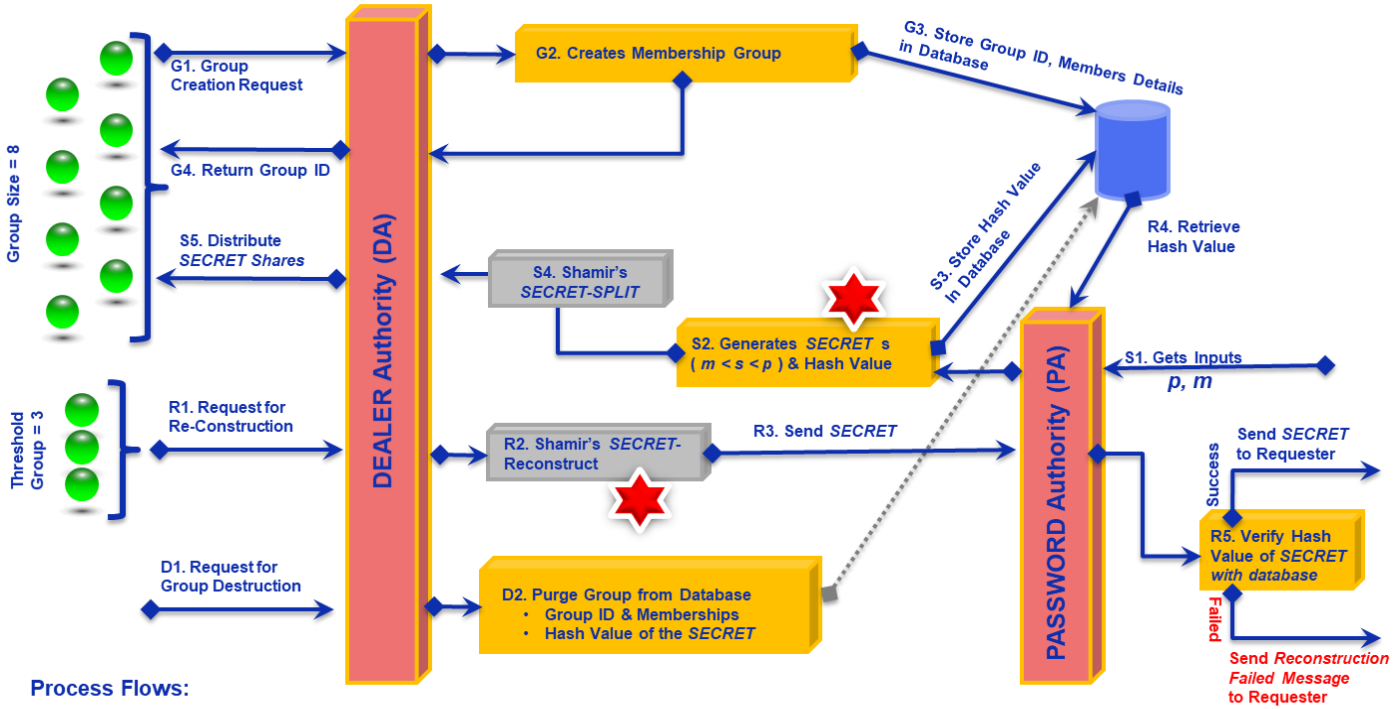
Figure 3.2 describes the process flow details for

- DEALER - Group Creation Process: G1, G2, G3, G4
- PA Secret Generation, DEALER Secret Share Distribution Process: S1, S2, S3, S4, S5
- DEALER Secret Share Collection, and Re-Construction Process: R1, R2, R3, R4, R5
- DEALER – Group Destruction Process: D1, D2

The above processes work in conjunction with Shamir's secret-Split and secret-Reconstruction processes described in detail in Chapter 2 (*Figure 2.1* and *Figure 2.2).*

It is important to note that in the proposed application, neither the password nor the secret shares are stored anywhere in the system, and only the users know their own individual shares. This happens to be the primary benefit of Shamir's Secret Sharing scheme.

Figure 3.2 Group Management, Secret Share Distribution and Secret Reconstruction Process Flows



Process Flows:

- DEALER - Group Creation Process :: G1, G2, G3, G4
  On request from group (administrator), DA Creates the Group, stores the Group ID & Details in Database, and returns the Group ID to the members of the group
- PA Secret Generation, and DEALER Secret Share Distribution Process :: S1, S2, S3, S4, S5
  Using *p, m* (prime number, min. password number), PA generates the *SECRET* (m < s < p), split by Shamir's secretSPLIT function, and the shares are distributed by DA
- DEALER Secret Share Collection, and Re-Construction Process :: R1, R2, R3, R4, R5
  DA collects the shares from the threshold group, Shamir's secretReconstruct function reconstructs the *SECRET*, PA verifies the *SECRET*'s Hash Value with the Hash Value stored in Database, and upon successful verification PA sends the *SECRET* to the requester
- DEALER – Group Destruction Process :: D1, D2
  On request from the group (administrator), DA triggers the group Purge and deletes Group ID, Membership details and Hash Value of the *SECRET* from the database

Details of the pseudo code functions for the Dealer Authority and Password Authority are given in **Algorithm 2** below;

- **Dealer Authority:** GROUP-CREATION(), GROUP-DESTRUCTION(), SHARE-DISTRIBUTION() and SHARE-COLLECTION ()
- **Password Authority:** GENERATE-SECRET ()

Algorithm 2: Dealer Authority and Password Authority

START

1. dealer **receives** request from users
2. **switch()**
3. **case GC**
4. GROUP-CREATION ()
5. dealer **requests** the password management authority for a secret
6. GENERATE-SECRET ()
7. dealer **stores** *p, hash* values in the database
8. SECRET-SPLIT (*t, n, p, s*)
9. dealer **destroys** the secret *s*
10. SHARE-DISTRIBUTE(*shares*)
11. **case AR**
12. SHARE-COLLECTION ()
13. dealer **receives** token
14. **if** token == 1 **then**
15. dealer **sends** authorization to users
16. **else**
17. dealer **rejects** authorization to users
18. **case RG**
19. GROUP-DESTRUCTION ()
20. **end switch**

END

GROUP-CREATION ()

1. **create** group with the list of members & their authentication credentials

2. **issue** a *groupID*
3. **select** the threshold value *t*
4. **store** *groupID* and group members details in database
5. **distribute** *groupID* to the members

GENERATE-SECRET ()

1. **select** a minimum value for the secret *m*
2. **select** a prime number *p* of size $(m \leq p \leq 2^{256})$
3. **generate** a random value between *m* and *p*
4. **assign** the random value to secret *s*
5. **create** *hash* value for the secret
6. **return** *p, hash, s* to the dealer

SECRET-SPLIT (*t, n, p, s*)

1. **calculates** *shares*
2. **delete** *s*
3. **return** *shares* to the dealer

SHARE-DISTRIBUTE (*shares*)

1. dealer **distributes** *shares* to the group members
2. **delete** *shares* and *s*

SHARE-COLLECTION ()

1. **collect** *shares* from members
2. SECRET-RECONSTRUCT(*shares)*
3. **return** token (1 or 0) to dealer

SECRET-RECONSTRUCT (*shares)*

1. **reconstruct** the secret *s*
2. VERIFY-VALUES(*s*)
3. **return** token (1 or 0)

VERIFY-VALUES (*s*)

1. **creates** *hash* value for the reconstructed secret *s*
2. **verifies** the created *hash* value with the stored *hash* in the database
3. **return** token (1 or 0)

GROUP-DESTRUCTION ()

1. **purge** from database *groupID*, group memberships, *hash, p*

*****The application developed as part of this thesis work uses server memory instead of the database to store all group credentials, hash value etc.

Algorithm 3: User

START

1. **switch()**
2. **case GC**
3. user **sends** request to the dealer for creation of the group
4. user **receives** *groupID* from the dealer
5. user **receives** a *secret share* from the dealer
6. **case AR**
7. user **sends** authorization request to the dealer
8. user **receives** request for the *secret share* from the dealer
9. user **sends** the *secret share* to the dealer
10. user **receives** response from the dealer
11. **if** the response is positive **then**
12. user is granted authorization
13. **else**
14. user is denied authorization
15. **case RG**
16. user **sends** request to the dealer for removal of the group
17. **end switch**

END

Chapter 4 : Application Details

4.1 Application

In this thesis I have developed a multithreaded client / server application. On the server side, when the **SecretServer.JAVA** module is executed, the server starts, a socket is created and the server starts listening for client connections on port number 1111. On the client side, when the Clients.JAVA module is executed, a frame pops out with a text field for the client to provide IP address of the server and the UserID of the client. If the provided details are correct, the server accepts the connection, a thread is created and assigned to the client. In this multithreaded environment whenever a client is connected, a new thread is created and assigned to the client. After the thread assignment, the client is provided with four choices as explained below.

1. IN (Initialization of the process) – To create a group which participates in the secret sharing process.

2. SS (Secret Share generation request) – When the client gives this response, the server prompts for the group details. If the details provided are correct, then the secret sharing process is initiated.

3. AR (Authorization Request) – When the client chooses this response, the server prompts for the details of the members who are going to participate in the secret reconstruction process. This procedure is followed for every client to ensure that all the clients present in the group have agreed for the process.

4. TG (Terminate Group) – When the client chooses this response, the server prompts for the group details. If the provided details are correct, the server disconnects with all the clients present in that group and deletes their details.

The **SServer.JAVA** module is where the shares generation and secret reconstruction process are performed. When the **SS** response is chosen, the **secretSplit ()** method is called. In this method the secret is generated using a pre-defined prime as basis. After the secret is generated, the shares are calculated and are distributed among the clients present in the group using the **recivShare ()** method. When the **AR** response is chosen, and when all the shares are received by the server, these shares are received using **sendShare ()** method. After all the shares are received, **secretReconstruct ()** method is called. This method uses Lagrange interpolation to reconstruct the secret, if the reconstructed secret is correct '1' is returned to the client, else '0' is returned.

Pseudo code details of the Secret Server Module and the Client Module are given below in **Algorithm 3** and **Algorithm 4** respectively.

Refer to **Appendix 1** for full listing of the JAVA modules (code) developed as part of this thesis work.

Algorithm 4: Secret Server Module

START

1. *port* ← 1111
2. **open** *Server socket*
3. **bind** *socket* to the declared *port*
4. **start** listening for *Client* connections
5. **receive** request from *Client*
6. **assign** thread to the connected *Client*
7. **open** an *input stream* and *output stream*
8. **get** *ClientID* from the *Client*
9. **if** *ClientID* is correct **then**
10. **add** *Client* to the *Client's List*
11. **send** message "select an option: IN or SS or AR or TG" to *Client* for response
12. **get** response **IN or SS or AR or TG** from *Client*

```
13.     switch(response)
14.         case IN
15.             prompt Client for n and t
16.             prompt Client for the group details
17.             send message to Client "Group Created"
18.         case SS
19.             prompt Client for the group details
20.             if group details are correct then
21.                 instantiate the SSERVER Object
22.                 call secretSplit(n,t)
23.                 send Clients their Shares
24.         case AR
25.             prompt Client for n and t
26.             prompt Client for the group details
27.             if group details are correct then
28.                 prompt Client for their Share
29.             else
30.                 send message "Invalid details…try again"
31.             receive at least t Shares from Clients
32.             call secretReconstruct(t)
33.             receive value from secretReconstruct(t)
34.             if value == 1 then
35.                 send message to Clients "secret
                    reconstruction successful"
36.             else
37.                 send message to Clients "secret
                    reconstruction unsuccessful…..try again"
38.         case TG
39.             prompt Client for the group details
40.             if group details are correct then
41.                 send message to Clients "group removed"
42.                 delete group details
43.                 close Server socket
44.if Client exits
45.remove Client from Client's List

END
```

Algorithm 5: Client Module (executed by USER)

START

1. **open** *Client* socket
2. **open** frame
3. **prompt** the *Client* for *Server's IP Address*
4. **if** *Server's IP Address is correct* **then**
5. **prompt** the *Client* for *ClientID*
6. **if** *ClientID* is correct **then**
7. **open** a frame with text field and message field
8. **else**
9. **prompt** the *Client* for valid *ClientID*
10. **else**
11. **prompt** the *Client* for *Server's IP Address*
12. **if** server message STARTS with "MESSAGE:" **then**
13. **display** message in message field
14. **click** on close button
15. **close** frame

END

4.2 Flow-Chart

Given below is the flow chart of both client and server modules of the Java program developed to study and illustrate Shamir's secret sharing algorithm.

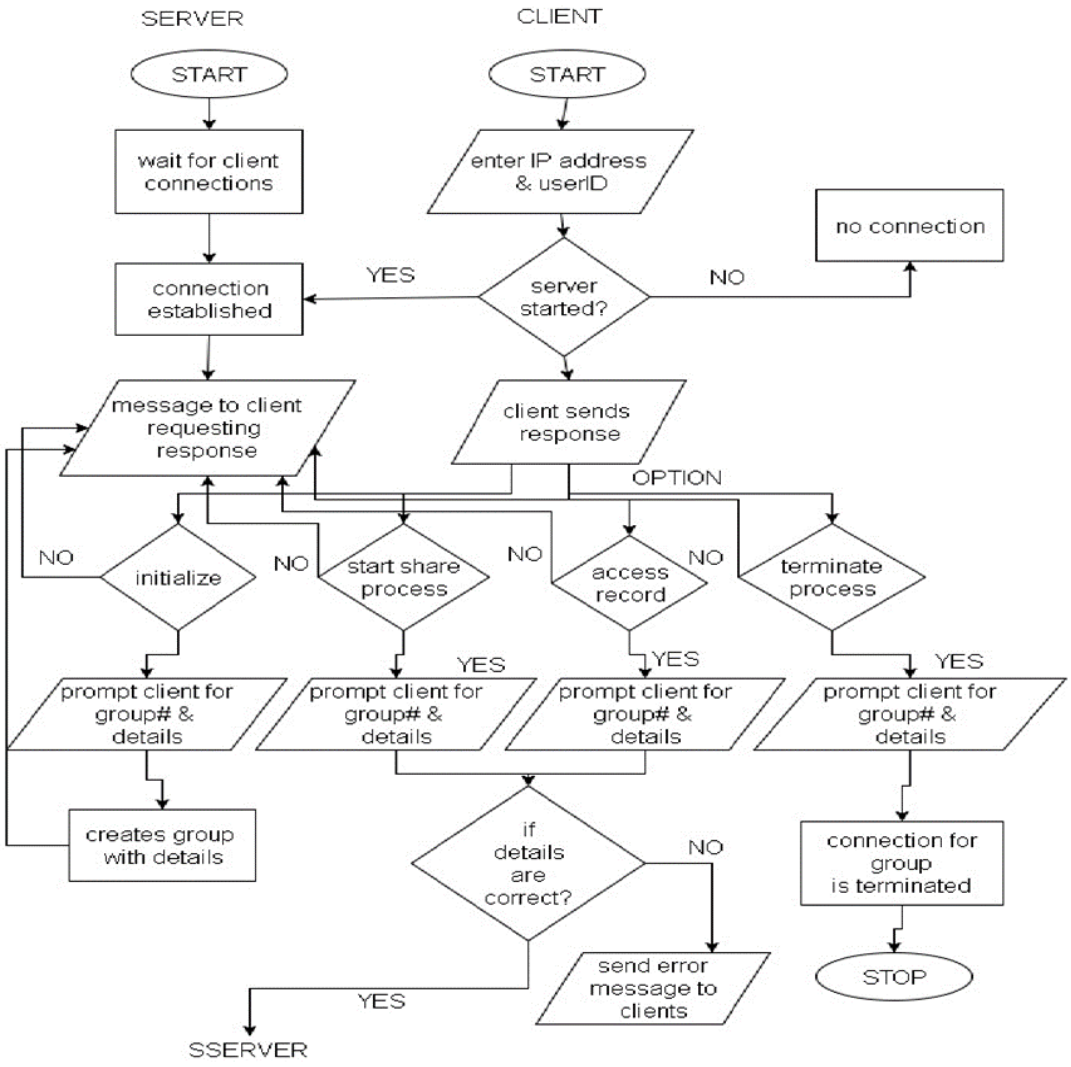


Figure 4.1 Flowchart of Client & Server modules

The ***SServer*** module below splits the secret into multiple shares.

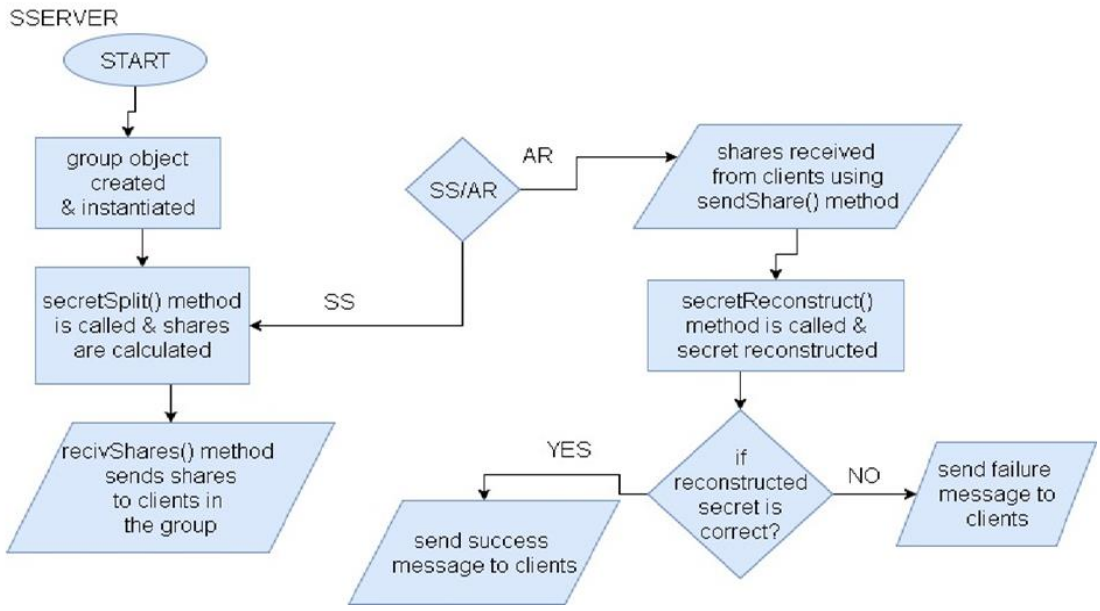


Figure 4.2 Flowchart of Secret sharing process

4.3 JAVA Modules

Given below is a sample java module function *secretSplit(),* which splits the secret into corresponding shares based on a given number of participants *n*, threshold value *t*, and the participant group number *gnum*. In this method the generated secret is split into shares using the parameters provided, and the pre-defined prime number. After the shares are generated the hash value of the secret is calculated and stored in a variable named 'hvalue', later the secret variable is made 0.

```
/*Method to generate a secret using rndBigInt method,
  and splitting it into shares using particNum and threshold.
  After shares are calculated,a hash value of the secret is created
  and the secret is set to 0*/
public static void secretSplit(int particNum, int threshold, int groupNum)
{
   System.out.println("Prime Number: " + prime);

   final SecureRandom random = new SecureRandom();

   secret = rndBigInt(prime);

   final BigInteger[] coeff = new BigInteger[particNum];

   coeff[0] = secret;

   for (int i = 1; i < particNum; i++)
   {
      coeff[i] = rndBigInt(prime);
   }

   for (int x = 1; x <= threshold; x++)
   {
      temp = secret;

      for (int exp = 1; exp < particNum; exp++)
      {
         temp = (temp.add(coeff[exp].multiply((BigInteger.valueOf(x).pow(exp))
                  .mod(prime))).mod(prime)).mod(prime);
      }
      shares[x-1][1] = temp;

      shares[x-1][0] = BigInteger.valueOf(x);

      System.out.println("Share " + shares[x-1][1]);
   }

   hvalue = BigInteger.valueOf(secret.hashCode());

   secret = BigInteger.ZERO;

   temp = BigInteger.ZERO;

}
```

Figure 4.3 secretSplit() method

The method *secretReconstruct()* accepts the threshold number *t*, and the group number *gnum* as the parameters, and in this method the secret is reconstructed using the shares provided and the pre-defined prime number using Lagrange interpolation. After the secret

is reconstructed, it's hash value is compared with the already stored hash value during the shares generation. If the compared values are equal 1 is returned which sends a message saying, "reconstruction is successful", else 0 is returned by sending a message "wrong shares…please try again".

```
/*Method to reconstruct the secret using the received threshold value and
  shares, after the secret is reconstructed a hash value is generated and
  compared with the calculated hash value of the secret in the secret spliting method*/
public static int secretReconstruct( int threshold, int groupNum)
{
  int k;

  for(int formula = 0; formula < threshold; formula++)
  {
     BigInteger numer = BigInteger.ONE;

     BigInteger denom = BigInteger.ONE;

     System.out.println("Number of shares is: "+threshold);

     for(int count = 0; count < threshold; count++)
     {
        if(formula == count)
           continue;

        BigInteger sposition = shares[formula][0];

        BigInteger nposition = shares[count][0];

        numer = numer.multiply(nposition.negate()).mod(prime);

        denom = denom.multiply((sposition.subtract(nposition))).mod(prime);
     }
     System.out.println("The share is"+ share[formula]);

     BigInteger tmp = share[formula].multiply(numer) . multiply(denom.modInverse(prime));

     secret = prime.add(secret).add(tmp) . mod(prime);
  }

  System.out.println("The secret is: " + secret + "\n");

  if(hvalue.compareTo(BigInteger.valueOf(secret.hashCode()))==0)
     k=1;
  else
     k=0;

  return k;
}
```

Figure 4.4 secretReconstruction() method

Chapter 5 : Test Cases & Results

5.1 Test Case - Experiment #1

The following numerical test cases were used to illustrate the length variance of the secret and the corresponding calculated length of the shares. In other words, the data type of the secret varies from Integer (int) to BigInteger (BigInt class), and the corresponding calculated share size (number of digits) is tabulated. In all the cases, the calculated share size is either very close or equal to the size of the secret itself.

#	SECRET			# of Shares	Threshold	Share Size
	Basis (Prime)	Data Type	# of digits			
1	12347	int	5 digits	20	11	4-5 digits
2	8765437	int	7 digits	15	7	7 digits
3	987654323	int	9 digits	12	3	9 digits
4	98765441	int	8 digits	10	4	8 digits
5	2147483629 (<2^31-1)	int	10 digits	10	2	10 digits
6	2147483659 (>2^31-1)	long	10 digits	10	8	8 digits
7	18 digits	long	18 digits	3	2	18 digits
8	18 digits	long	18 digits	20	15	16-18 digits
9	19 digits	long	19 digits	8	4	19 digits
10	19 digits (<2^63-1)	long	19 digits	4	2	19 digits
11	19 digits	BigInt class	19 digits	20	15	19 digits
12	38 digits	BigInt class	37 digits	15	10	37 digits
13	49 digits	BigInt class	48 digits	7	7	48 digits
14	61 digits	BigInt class	61 digits	10	8	61 digits
15	78 digits	BigInt class	76 digits	10	6	76 digits

TABLE 1

Note: *Here are the lengths of the data types as per the System/JAVA compiler used in developing the program. Additional system details are listed in Appendix 2.*

Data Type	Byte Length	# of Maximum Number limit
int	4 bytes	2^{31}-1
long	8 bytes	2^{63}-1
BigInt class	Internal array	2,147,483,647 digits (or 2^{31} – 1 digits)

5.2 Test Case - Experiment #2

The following table (Table 2) captures the time taken for the generating the secret shares and corresponding reconstruction of the secret shares for different lengths of the secret considered for testing (prime number is taken as the secret). In this experiment, the prime number length ranges from 1 digit to 1million digits, and the calculation time is captured in milliseconds (ms).

Password Length (Prime # of digits)
(vs)
Secret Share Generation, Reconstruction CPU time

	Prime Number		Secret Shares		
#	# of digits	Generation Time (ms)	# of Shares	Generation Time (ms)	Reconstruction Time (ms)
1	1	4	6	0	0
2	10	5	6	0	0
3	100	75	6	15	0
4	1000	772	6	16	31
5	1332	M_{4423}	6	31	47
6	13395	M_{44497}	6	266	469
7	227832	M_{756839}	6	3754	103363
8	909526	$M_{3021377}$	6	15069	1560307
	Mersenne Prime Number			$M_n = 2^n - 1$	

TABLE 2

Note: *For prime numbers with lengths above 1000 digits, Mersenne prime numbers were generated using the formula cited above.*

Figure 5.1 is a graph plotted with # of digits of prime number on x-axis and the share generation time, secret reconstruction time (milliseconds) on the y-axis. The graph clearly indicates an increase in secret share generation time as the length of the secret (# of digits) increases. Further, the graph also shows a dramatic increase in the corresponding secret

reconstruction time, which would adversely impact the response time and would be a serious limiting factor for any practical implementation of the algorithm.

For secret lengths below 1 KB the share generation and reconstruction times are approximately 16 to 31 milliseconds, which are within acceptable limits for most realtime implementations in enterprise data centers.

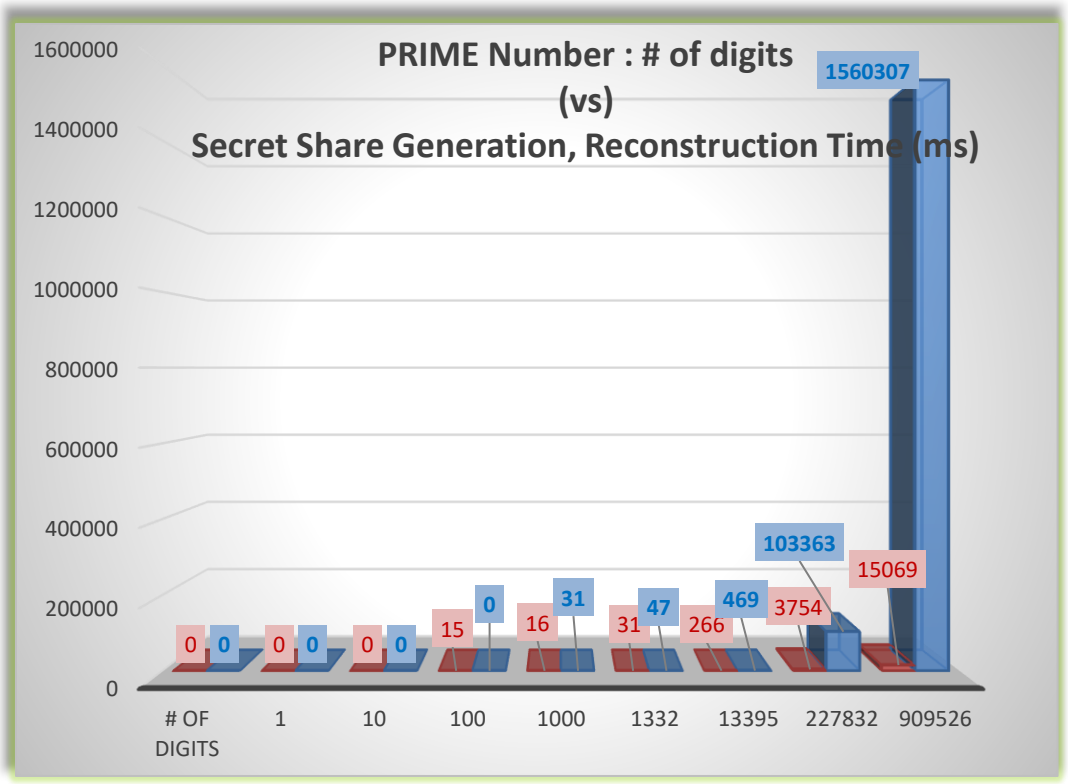


Figure 5.1 Plotted graph from the results of experiment #2 (# of digits vs share generation, reconstruction time (ms))

Chapter 6 : Conclusion

In conclusion, Shamir's Secret Sharing scheme offers a strong fundamental basis for group-oriented login (passwords, secret keys, PIN #'s etc.) algorithm, and can effectively secure highly sensitive information assets.

As indicated by the results in Chapter 4, during the analysis, I noticed that as the width of the secret increases, for all practical purposes, the vulnerability to brute force attacks are eliminated, but the algorithm takes more time (CPU cycles) during secret splitting and the reconstruction process.

Further, as discussed above the enhancements to Shamir's scheme suggested by Tompa[3], Tartary and Wang[6], Rabin[2] and Krawczyk[4] addresses various vulnerabilities, while retaining the security and efficiency of the original scheme.

References

1. Adi Shamir, 'How to share a secret', Communications of the ACM, vol.22, no.11, November 1979

2. M.Rabin, 'Randomized Byzantine generals', Proc. 24th IEEE Symposium Foundation Computer Science, November 1983

3. Martin Tompa, 'How to Share a Secret with Cheaters', IBM Thomas J. Watson Research Center, 1988

4. Hugo Krawczyk, 'Secret Sharing Made Short', IBM Thomas J. Watson Research Center, 1993

5. Miao Fuyou, Fan Yuanyuan,Wang Xingfu,Yan Xiong and Moaman Badawy, ‘A (t,m,n)-Group Oriented Secret Sharing Scheme’, Chinese Journal of Electronics, 2010

6. Christophe Tartary, Huaxiong Wang, 'Dynamic Threshold and Cheater Resistance for Shamir Secret Sharing Scheme', Proceedings of the 2nd SKLOIS Conference on Information Security and Cryptology (INSCRYPT 2006), Lecture Notes in Computer Science, Springer - Verlag, vol. 4318, pp 103 - 117, 2006

7. S.Goldwasser, S.Micali, and R.Rivest, ‘A “paradoxical solution” to the signature problem’, Proceedings of 25th IEEE Symp. Foundations of Computer Science, Oct. 1984

8. M.Rabin, ‘Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance’, Journal of ACM, vol.36, issue.2, Apr.1989

9. Lein Harn, ‘Group Authentication’, IEEE Transactions on Computers, vol. 62, no. 9, Sep.2013

10. Lein Harn, Changlu Lin, ‘An Efficient Group Authentication for Group Communications’, International Journal of Network Security & Its Applications (IJNSA), Vol.5, No.3, May.2013

Appendix 1

Application Java modules and the code.

SServer.JAVA

```
class SServer
{
   static int maxNumSh=1000, shar=2, maxThre=300;
   //maxThre indicates the max shares used in reconstruction, maxNumSh
is max shares generated

   private static BigInteger[][] shares = new
BigInteger[maxNumSh][shar];
   //Array to store the share number and the generated share

   private static BigInteger[] share = new BigInteger[maxThre];
   //Array to store the collected shares

   private static final BigInteger prime = new
BigInteger("115792089237316195423570985008687907853269984665640564039457
584007913129640233");
   //prime number which is greater than 2^256

   private static BigInteger secret, temp, hvalue;
   //hvalue is to store the calculated hash value of the secret

   /*Method to generate random integers between 1 and the declared
prime number using Random() function*/
   public static BigInteger rndBigInt(BigInteger max)
   {
```

```
        Random rnd = new Random();
        do
        {
            BigInteger i = new BigInteger(max.bitLength(), rnd);

            if (i.compareTo(BigInteger.ZERO)>0 && i.compareTo(max) < 0)
                return i;
        } while (true);
    }

    /*Method to split the secret divided is generated using rndBigInt
method and depending on particNum and threshold the shares are
calculated and the secret is set to 0 after its hash value is
calculated*/
    public static void secretSplit(int particNum, int threshold, int
groupNum)
    {
        System.out.println("Prime Number: " + prime);

        final SecureRandom random = new SecureRandom();

        secret = rndBigInt(prime);

        final BigInteger[] coeff = new BigInteger[particNum];

        coeff[0] = secret;

        for (int i = 1; i < particNum; i++)
        {
```

```
            coeff[i] = rndBigInt(prime);
        }

        for (int x = 1; x <= threshold; x++)
        {
            temp = secret;

            for (int exp = 1; exp < particNum; exp++)
            {
                temp =
(temp.add(coeff[exp].multiply((BigInteger.valueOf(x).pow(exp)).mod(prim
e))).mod(prime)).mod(prime);
            }
            shares[x-1][1] = temp;

            shares[x-1][0] = BigInteger.valueOf(x);

            System.out.println("Share " + shares[x-1][1]);
        }

        hvalue = BigInteger.valueOf(secret.hashCode());

        secret = BigInteger.ZERO;

        temp = BigInteger.ZERO;

    }
```

```
    /*In this method the secret is constructed using the received
threshold value and the shares, after the secret is reconstructed a
hash value is generated and compared with the calculated hash value of
the secret in the secret spliting method*/
    public static int secretReconstruct( int threshold, int groupNum)
    {
      int k;

      for(int formula = 0; formula < threshold; formula++)
      {
         BigInteger numer = BigInteger.ONE;

         BigInteger denom = BigInteger.ONE;

         System.out.println("Number of shares is: "+threshold);

         for(int count = 0; count < threshold; count++)
         {
            if(formula == count)
               continue;

            BigInteger sposition = shares[formula][0];

            BigInteger nposition = shares[count][0];

            numer = numer.multiply(nposition.negate()).mod(prime);

            denom =
denom.multiply((sposition.subtract(nposition))).mod(prime);
```

```
            }
            System.out.println("The share is"+ share[formula]);

            BigInteger tmp = share[formula].multiply(numer) .
multiply(denom.modInverse(prime));

            secret = prime.add(secret).add(tmp) . mod(prime);
         }

         System.out.println("The secret is: " + secret + "\n");

         if(hvalue.compareTo(BigInteger.valueOf(secret.hashCode()))==0)
            k=1;
         else
            k=0;

         return k;
      }


      /*Method to store the received shares from the clients that are
required to reconstruct the secret*/
      public static void sendShare(int j, BigInteger sh)
      {
         share[j]=sh;
      }

      /*Method to send the calculated shares to the clients after the
secret is generated and split*/
```

```
    public static BigInteger recivShare(int j)
    {
        return shares[j][1];
    }

}
```

SecretServer.JAVA

```
public class SecretServer
{

   private static final int PORT = 1111;

   private static int clientCount=0, n, t,recGrpCount=0, cs=0;

   private static int suc=0, unsuc=0,g, groupCount=0, genSecCount=0;

   static int maxNumGrp=30, maxGrpMem=30;
   //maxNumGrp is max number of groups that can be handled
   //maxGrpMem is max members that can be handled in each group

   private static BigInteger[][] shares = new
BigInteger[maxNumGrp][maxGrpMem];
   //Array to hold the generated shares that are distributed among the
members of a group

   private static BigInteger[] share = new BigInteger[maxGrpMem];
   //Array to collect the shares from members to reconstruct the secret

   static String[][] gnames = new String[maxNumGrp][maxGrpMem];
   //Array to store the names of the clients and their respective group
number

   static String[][] pnames = new String[maxNumGrp][maxGrpMem];
```

```
   //Array to store the names of the clients who are trying to access
the system and their respective group number

   static int[] ct = new int[maxNumGrp];

   static int[] j = new int[maxNumGrp];

   static int time = 300000;
   //int variable

   static int[][] nt = new int[maxNumGrp][2];
   //Array to hold the number of participants and the threshold for the
respective groups

   static SServer[] ss = new SServer[maxNumGrp];
   //An object array where each object is assigned to each group for
the secret sharing process

   private static HashSet<String> names = new HashSet<String>();

   private static HashSet<PrintWriter> writers = new
HashSet<PrintWriter>();



   public static void main(String[] args) throws Exception
   {
      System.out.println("The server is running.");
```

```
        ServerSocket listener = new ServerSocket(PORT);
        try
        {
           while (true)
           {
              new Handler(listener.accept()).start();
           }
        }
        finally
        {
          listener.close();
        }
    }

    /*Method to display the details of the clients connected and the
groups created and other functions*/
    private static void displayStatus()
    {
        int i=0,j=0;
        System.out.println("Groups created: "+groupCount);
        System.out.println("Total clients connected: "+clientCount);
        while(i<30)  //for(int i=0;i<30;i++)
        {
           System.out.println("The group number is:"+i+" clients in the
group are:");
           while(j<30)  //for(int j=0;j<30;j++)
           {
              if(gnames[i][j]!= null)
```

```
            {
               System.out.print(gnames[i][j]);
            }
            else
            {
               i++;
               break;
            }
         }
      }
      System.out.println("Total secrets generated:"+genSecCount);
      System.out.println("Total groups who reconstructed the
secret:"+recGrpCount);
      System.out.println("Successful reconstructions:"+suc);
      System.out.println("Unsuccessful reconstructions:"+unsuc);
   }

   /*Method to check if a particular client is present in the group
that is mentioned by him*/
   private static int checkGroup(int particNum, int threshold, String
particName, int groupNum)
   {
      int num=0;
      for(int i=0;i<particNum;i++)
      {
         if(nt[groupNum][0]==particNum && nt[groupNum][1]==threshold)
         {
            if(particName.compareToIgnoreCase(gnames[groupNum][i])==0)
               num=1;
```

```
            }
        }
        return num;
    }

    /*Method to intialize and start the process of secret sharing if the
group number provided and the members of that particular group are
connected to the server, and the calculated shares are distributed
among the group members*/
    public static void inSProcess(int groupNum)
    {
        int i=0,count=0,l=0;

        for(PrintWriter writer: writers)
        {
            if(names.contains(gnames[groupNum][i++]))
            {
                count++;
            }
        }
        if(count==nt[groupNum][0])
        {
            ss[groupNum].secretSplit( nt[groupNum][1], nt[groupNum][0],
groupNum);
            genSecCount++;
        }
        for(PrintWriter writer: writers)
        {
            if(names.contains(gnames[groupNum][l]))
```

```
            {
                writer.println("SHARE:Your share
is"+ss[groupNum].recivShare(l));
                l++;
            }
        }
    }

    /*A thread handler class to handle the connected clients as
individual threads and provides options to create a group, initialize
the sharing process, reconstructing the secret and giving access to
records and terminated the process for a particular group*/
    private static class Handler extends Thread
    {
        private String name;
        private Socket socket;
        private BufferedReader in;
        private PrintWriter out;

        public Handler(Socket socket)
        {
            this.socket = socket;
        }

        public void run()
        {
            try
            {
                in = new BufferedReader(new InputStreamReader(
```

```
            socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);
            while (true)
            {
               out.println("ID");
               name = in.readLine();
               if(name=="admin")
               {
                  displayStatus();
               }
               if ((!names.contains(name)))
               {
                  names.add(name);
                  clientCount++;
                  break;
               }
            }
            writers.add(out);
            while(true)
            {
               out.println("MESSAGE:Please enter IN to Initialize the
process,");
               out.print("MESSAGE:SS to start Secret Sharing
process,");
               out.print("MESSAGE:AR to Access the system,");
               out.print("MESSAGE:TP to Terminate the process");
               String input = in.readLine();
               System.out.println("Command received");
               switch(toUpperCase(input))
```

```
                {
                    case "IN":
                        out.println("MESSAGE:Please specify the group
number");
                        String gn = in.readLine();
                        g = Integer.parseInt(gn);
                        if(gnames[g][0]==null)
                        {
                            out.println("MESSAGE: Please specify the # of
participants and the threshold");
                            String np = in.readLine();
                            nt[g][0] = Integer.parseInt(np);
                            String tp = in.readLine();
                            nt[g][1] = Integer.parseInt(tp);
                            out.println("MESSAGE:Please provide the names
of the participiants");
                            for(int i=0;i<nt[g][0];i++)
                            {
                                String pname = in.readLine();
                                gnames[g][i] = pname;
                            }
                            out.println("MESSAGE: Group created.");
                            groupCount++;
                        }
                        else
                        {
                            out.println("MESSAGE:Please specify the number
of participants");
                            String p = in.readLine();
```

```
                        int pn = Integer.parseInt(p);
                        out.println("MESSAGE:Please provide the
threshold");
                        String pt = in.readLine();
                        int ptt = Integer.parseInt(pt);
                        if(pn==nt[g][0] && ptt==nt[g][1])
                        {
                           out.println("MESSAGE:Please provide the
names of the participiants");
                           for(int i=0;i<pn;i++)
                           {
                              String nam = in.readLine();
                              if(checkGroup(pn,ptt,nam,g)==1)
                              {
                                 continue;
                              }
                              else
                              {
                                 out.println("MESSAGE: Invalid group.
Please check the group number again.");
                                 break;
                              }
                           }
                           out.println("MESSAGE: Group created.");
                        }
                        else
                           System.out.println("Invalid group count");
                     }
                     break;
```

```
                    case "SS":
                        out.println("MESSAGE:Please provide the group
number:");
                        String n = in.readLine();
                        t = Integer.parseInt(n);
                        inSProcess(t);
                        break;

                    case "AR":
                        socket.setSoTimeout(time);
                        out.println("MESSAGE: Please provide the group
number:");
                        String ga = in.readLine();
                        int gb = Integer.parseInt(ga);
                        while(true)
                        {
                           try
                           {
                              if(gnames[gb][0]!=null)
                              {
                                 if(pnames[gb][0]==null)
                                 {
                                    j[gb]=0;
                                    out.println("MESSAGE: Please provide
the count of the group and their details.");
                                    String gc = in.readLine();
                                    cs = Integer.parseInt(gc);
```

```
                                System.out.println("the count of the
group is:" +cs);
                                if((cs>=nt[gb][1]) && (cs<=nt[gb][0]))
                                {
                                   for(int i=0;i<cs;i++)
                                   {
                                      String groupnames =
in.readLine();
                                      pnames[gb][i]=groupnames;
                                   }
                                   out.println("MESSAGE:Please provide
your share:");
                                   String sj = in.readLine();
                                   BigInteger ha;
                                   ha = new BigInteger(sj);
                                   shares[gb][j[gb]]= ha;
                                   ss[gb].sendShare(j[gb],ha);
                                   j[gb]++;
                                   recGrpCount++;
                                }
                                else
                                {
                                   out.println("MESSAGE: Invalid
threshold. Please try again... ");
                                   }
                             }
                             else
                             {
                                for(int i=0;i<cs;i++)
```

```
                                {
                                    if(pnames[gb][i].contains(name))
                                    {
                                        out.println("MESSAGE: Please
provide your share:");
                                        String sh = in.readLine();
                                        BigInteger h = new
BigInteger(sh);
                                        ct[gb]=j[gb];
                                        shares[gb][j[gb]] = h;

ss[gb].sendShare(j[gb],shares[gb][j[gb]]);
                                        j[gb]++;
                                        System.out.println("the given
share is:"+shares[gb][i]);
                                    }
                                }
                            }
                            else
                            {
                                out.println("MESSAGE:Group not yet
created....Please check again");
                                break;
                            }
                            for(int i=0;i<cs;i++)
                            {
                                System.out.println(pnames[gb][i]);
                            }
```

```
                    if(ct[gb]>=nt[gb][1])
                    {
                        int i=0;

if(ss[gb].secretReconstruct(ct[gb],gb)==1)
                        {
                            suc++;
                            for (PrintWriter writer : writers)
                            {

if(pnames[gb][i].contains(gnames[gb][i]))
                                {
                                    writer.println("MESSAGE:Secret
successfully reconstructed");
                                    i++;
                                }
                            }
                        }
                        else
                        {
                            unsuc++;
                            for (PrintWriter writer : writers)
                            {

if(pnames[gb][i].contains(gnames[gb][i]))
                                {

writer.println("MESSAGE:reconstruction failed. Try again...");
                                    i++;
```

```
                                }
                            }
                        }
                        break;
                    }
                }
                catch(SocketTimeoutException e)
                {
                    for (PrintWriter writer : writers)
                    {
                        writer.println("MESSAGE:Session timedout.
Try again... ");
                        socket.setSoTimeout(0);
                    }
                    break;
                }
                break;
            }

        case "TP":
            out.println("MESSAGE: Please provide the group
number:");
            String gc = in.readLine();
            int gd = Integer.parseInt(gc);
            int l=0;
            for (PrintWriter writer : writers)
            {
                if(names.contains(gnames[gd][l++]))
                {
```

```
                        writer.println("MESSAGE:The process is
terminated and group is deleted");
                        socket.close();
                    }
                }
                for(int i=0;i<nt[gd][0];i++)
                    gnames[gd][i] = null;
                nt[gd][0]=0;
                nt[gd][1]=0;
                break;

            default:
                System.out.println("Invalid command");
                break;
        }
    }
}

catch (IOException e)
{
    System.out.println(e);
}
finally
{
    if (name != null)
    {
        names.remove(name);
    }
    if (out != null)
```

```
                {
                    writers.remove(out);
                }
                try
                {
                    socket.close();
                }
                catch (IOException e)
                {
                }
            }
        }
    }
}
```

Clients.JAVA

```
public class Clients {


   BufferedReader in;
   PrintWriter out;
   JFrame frame = new JFrame("Client");
   JTextField textField = new JTextField(100);
   JTextArea messageArea = new JTextArea(8, 40);


   /*
    Constructs the client by laying out the GUI and registering a
    listener with the textfield so that pressing Return in the
    listener sends the textfield contents to the server.  Note
    however that the textfield is initially NOT editable, and
    only becomes editable AFTER the client receives the NAMEACCEPTED
    message from the server.
   */
   public Clients()
   {
      textField.setEditable(true);
      messageArea.setEditable(false);
      frame.getContentPane().add(textField, "North");
      frame.getContentPane().add(new JScrollPane(messageArea),
"Center");
      frame.pack();


      textField.addActionListener(new ActionListener()
      {
```

```
            /*
             Responds to pressing the enter key in the textfield by
sending
             the contents of the text field to the server.Then clear
             the text area in preparation for the next message.
            */
            public void actionPerformed(ActionEvent e)
            {
               out.println(textField.getText());
               textField.setText("");
            }
         }
         );
      }

      /*Prompt for and return the address of the server.*/
      private String getServerAddress()
      {
         return JOptionPane.showInputDialog(
         frame,
         "Enter IP Address of the Server:",
         JOptionPane.QUESTION_MESSAGE);
      }

      /*Prompt for and return the desired screen name.*/
      private String getName()
      {
         return JOptionPane.showInputDialog(
         frame,
```

```
        "Enter your Id:",
        JOptionPane.PLAIN_MESSAGE);
    }

    /*Connects to the server then enters the processing loop.*/
    private void run() throws IOException
    {

        // Make connection and initialize streams
        String serverAddress = getServerAddress();
        Socket socket = new Socket(serverAddress, 1111);
        in = new BufferedReader(new InputStreamReader(
            socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);

        // Process all messages from server, according to the protocol.
        while (true)
        {
            String line = in.readLine();
            if (line.startsWith("ID"))
            {
                out.println(getName());
            }
            else if (line.startsWith("MESSAGE"))
            {
                messageArea.append(line.substring(8) + "\n");
            }
            else if (line.startsWith("SHARE"))
            {
```

```
                messageArea.append(line.substring(6) + "\n");
            }
        }
    }

    /*Runs the client as an application with a closeable frame.*/
    public static void main(String[] args) throws Exception
    {
        Clients client = new Clients();
        client.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        client.frame.setVisible(true);
        client.run();
    }
}
```

Appendix 2

Details of the PC, OS and Java Version used for the compilation and runtime of both the client and server modules are given below.

- Processor : Intel Core i7
- RAM : 8.00 GB
- Operating System : Windows 10 (64-Bit)
- Compiler : NetBeansIDE 8.2
- Java Runtime : Java SE Runtime Environment 1.8.x