DirectShare
A Fast and Secure Peer to Peer File Sharing Protocol

Student Name: Arpan Jati

IIIT-D-MTech-CS-IS-11-002
July 1, 2013

Indraprastha Institute of Information Technology
New Delhi

<u>Thesis Committee</u>
Somitra Sanadhya (Chair)
Sanjit Kaul
RK Agrawal

Submitted in partial fulfillment of the requirements
for the Degree of M.Tech. in Computer Science,
with specialization in Information Security

©2013 Arpan Jati
All rights reserved

Keywords: P2P networks, protocol design, application development, networking, Secure systems design and implementation, human computer interaction, and real-world implementation and study

Certificate

This is to certify that the thesis titled **"A Fast and Secure Peer to Peer File Sharing Protocol"** submitted by **Arpan Jati** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bonafide work carried out by her him under my guidance and supervision in the Security and Privacy group at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Dr. Somitra Kr. Sanadhya
Indraprastha Institute of Information Technology, New Delhi

Abstract

Many Peer to Peer (P2P) protocols and applications have become popular in the recent years and P2P accounts for a very large portion of the internet traffic. But, most protocols are designed with speed and efficiency in mind, and not security. In this thesis a new protocol *DirectShare* is proposed; it is primarily designed for a large user base but with security in mind.

DirectShare allows to share files in a very efficient manner, at the same time there are provisions to allow for strictly restricted file sharing between a group of users, there are also techniques which guarantees the identity of users. The security is implemented at the base level of the protocol and not as an extension, this allows for better control over the security aspects, and as the protocol can identify users uniquely, incentives can be given in a fair way to the users who provide resources to the network, which will help improve the overall user experience. At the same time DirectShare is extensible and can incorporate new features by addition of commands and responses.

The *DirectShare* protocol has been implemented as an application and is being used in the real world. The the measured performance of *DirectShare* is consistent with other protocol of similar nature.

Acknowledgments

I owe particular thanks to my parents who have always encouraged me in my pursuits, without their help all this would not have been possible.

I would like to thank Dr. Somitra Kumar Sanadhya, for being my mentor not only during this thesis, but for the whole masters course. He has inspired me to do good work, and have supported me all along the way.

I would like to thank Dr. Donghoon Chang, we had many discussions regarding the several security aspects of this thesis, he is very spirited, enthusiastic and always helpful.

I would also like to thank Dr. Sanjit Kaul and Dr. RK Agrawal for their comments; they have helped to improve the thesis.

I would like to thank the so many friends who enthusiastically installed the client application during development and provided me with valuable tips to improve the protocol. Without, their help and feedback, it would not have been possible to design such a complex a protocol. The bug and error reports (sometimes in the form of a door knock in the middle of the night) helped me immensely while writing the client application.

I would like to thank Dr. Pankaj Jalote for making IIIT-Delhi such a wonderful place to work.

I would like to thank the IT staff for very quick resolution of IT related problems, and catering to many of the specific needs for setting up and running of the the server.

Lastly, I would like to thank my batch-mates, seniors and juniors for providing with such a good environment for fruitful discussions, many of which helped me immensely.

Arpan Jati

Contents

1 Introduction **1**

1.1 Background . . . 1

1.2 Problem description . . . 1

1.2.1 Protocol Features . . . 2

1.2.2 Client Features . . . 2

2 General Concepts **3**

2.1 Peer-to-peer (P2P) Networks . . . 3

2.1.1 Distributed Hash Tables (DHTs) . . . 3

2.1.2 Centralized P2P File Sharing Networks . . . 4

2.2 Hash Trees . . . 5

2.3 Cryptographic Hash Function . . . 6

2.4 RSA . . . 6

2.4.1 Key generation . . . 6

2.4.2 Encryption . . . 7

2.4.3 Decryption . . . 7

2.4.4 OAEP . . . 7

3 Protocol Design and Description **8**

3.1 Client - Server . . . 8

3.1.1 Server to Client packet format . . . 8

3.1.2 Client to Server packet format . . . 8

3.1.3 Identity of Clients . . . 9

3.1.3.1 Public Registered ID (PRID) . . . 9

3.1.3.2 Client Secret (CSID) . . . 9

3.1.3.3 Session Authentication Token (ST) . . . 10

3.1.3.4 Session GUID(SGUID) . . . 10

3.1.4 Server to Client packet TYPES . . . 10

3.1.4.1 HELLO . . . 10

3.1.4.2 CONNECT_AUTH_INIT . . . 10
3.1.4.3 ACCOUNT_NON_EXISTENT . . . 10
3.1.4.4 AUTH_RESPONSE . . . 11
3.1.4.5 AUTH_CHALLENGE . . . 11
3.1.4.6 INSUFFICIENT_SHARE_SIZE . . . 12
3.1.4.7 USER_BANNED . . . 12
3.1.4.8 MAIL_RESPONSE . . . 12
3.1.4.9 CONNECT_RESPONSE . . . 13
3.1.4.10 ACCOUNT_CREATE_RESPONSE . . . 13
3.1.4.11 USERLIST . . . 13
3.1.4.12 CHAT_BROADCAST . . . 14
3.1.4.13 PERS_CHAT_FWD . . . 15
3.1.4.14 CLIENT_VERSION . . . 16
3.1.4.15 UPDATE_RESPONSE . . . 17
3.1.4.16 REQUEST_AUTH . . . 17
3.1.4.17 SEARCH_REQUEST . . . 18
3.1.4.18 DISCONNECT . . . 18
3.1.5 Client to Server packet TYPES . . . 18
3.1.5.1 HELLO . . . 18
3.1.5.2 CONNECT_REQ . . . 19
3.1.5.3 AUTH_CHALLENGE_RESP . . . 20
3.1.5.4 CLIENT_VERSION . . . 20
3.1.5.5 UPDATE_REQUEST . . . 20
3.1.5.6 ACCOUNT_CREATE_REQUEST . . . 21
3.1.5.7 USERLIST . . . 21
3.1.5.8 CHAT_BROADCAST . . . 21
3.1.5.9 SEARCH_INITIATE . . . 21
3.1.5.10 PERS_CHAT . . . 22
3.2 Client - Client . . . 23
3.2.1 Basic Packet Structure . . . 23
3.2.2 Processing of the Packets . . . 23
3.2.2.1 Packet Description: HELLO . . . 24
3.2.2.2 Packet Description: FILELIST_REQ . . . 24
3.2.2.3 Packet Description: FILELIST_DATA . . . 24
3.2.2.4 Packet Description: DOWNLOAD_REQUEST . . . 25
3.2.2.5 Packet Description: DOWNLOAD_REQUEST_DESCRIPTOR 26
3.2.2.6 Packet Description: DOWNLOAD_DATA . . . 26

3.2.2.7 Packet Description: SEARCH_RESPONSE 27
3.2.2.8 DISCONNECT 27
3.2.2.9 TOO_MANY_CONNECTIONS_FORCE_CLOSE 27
3.2.2.10 FILE_NOT_AVAILABLE 28
3.2.2.11 FILE_PARAMS_INVALID_BUSY 28
3.2.2.12 DOWNLOAD_ACK 28
3.2.2.13 DOWNLOAD_NACK 28
3.3 Algorithm: AUTH_HASH_GEN 29
3.3.1 Required Parameters 29
3.3.2 EXPECTED_HASH Generation 29

4 Protocol Implementation **30**
4.1 File Search 30
4.1.1 Search Algorithm: Damerau−Levenshtein 30
4.1.2 Search Implementation 31
4.2 File Download 32
4.2.1 File Download: Downloader State 33
4.2.2 File Download: Downloader Chunk State 34
4.2.3 File Download: Uploader State 36
4.3 FILELIST 36
4.3.1 FILELIST Download 37
4.4 Create Account 37
4.5 Private Chat 38
4.6 Authenticate User 39

5 Software Design **43**
5.1 Description of some Major Classes 43
5.1.1 IncomingTCPHandler 43
5.1.2 OutgoingTCPHandler 44
5.1.3 UserList 44
5.1.4 ChatCollection 45
5.1.5 TCPUserData 45
5.1.6 DownloadState 46
5.1.7 HashManager 46
5.2 Other software design considerations 46
5.2.1 Dependency 46
5.2.2 Scheduling of Tasks 48
5.2.3 Network Considerations 48

6 **Implementation and Usage** **49**
6.1 Features 49
6.1.1 Share and Search 49
6.1.2 Download 49
6.1.3 Hashing and File Integrity 50
6.1.4 Chat 50
6.1.5 User Management 50
6.2 Usage Instructions 50
6.2.1 First Run 50
6.2.2 Setting the IP Address 50
6.2.3 Account Creation 52
6.2.4 Browsing Shared Files 53
6.3 Download Queue 55
6.4 Finished Queue 56
6.5 File Searching 57
6.6 Sharing Folders 58
6.7 Secure Personal Chat 59
6.7.1 Chat Features 59
6.7.2 Blocking Users 61

7 **Results** **63**
7.1 Statistics (Alpha Testing) 63
7.2 Download Speeds 63
7.2.1 Wi-Fi 63
7.2.2 Gigabit LAN 65
7.3 Comparison with other protocols 67
7.3.1 Protocol Feature Comparison 67
7.3.2 Download Speed Comparison 67
7.3.2.1 Test Setup 68
7.3.2.2 Discussion 68

8 **Conclusion** **69**

9 **Future Work** **70**
9.1 Connection Reuse 70
9.2 Automatic Server Address Resolution 70
9.3 Pure P2P operation in the absence of a server 70
9.4 Direct Streaming of Video and Audio 70

9.5 Implementing Dynamic Chunk sizes . . . 71
9.6 NAT Support . . . 71
9.7 Bandwidth Management . . . 71
9.8 Supported Platforms . . . 71

List of Figures

2.1 Hash Tree 5

4.1 File Search Flow 31
4.2 File Search State 31
4.3 Download Flow 33
4.4 File Download: Downloader State 33
4.5 File Download: Downloader Chunk State 35
4.6 File Download: Uploader State 35
4.7 File List: Download Flow 36
4.8 Create Account Flow 37
4.9 Private Chat Flow 39
4.10 Authentication Sequence 40
4.11 Authenticate User State 41

5.1 Class: IncomingTCPHandler 43
5.2 Class: OutgoingTCPHandler 44
5.3 Classes: UserList and ChatCollection 45
5.4 Classes: TCPUserData and DownloadState 45
5.5 Class: HashManager 46
5.6 Class Dependency Diagram 47

6.1 First Run 51
6.2 Setting the IP Address 51
6.3 File->Create New Account Menu 52
6.4 Preferences Pane 53
6.5 E-mail Format (Copy everything in the body of the mail.) 53
6.6 User List showing connected users 54
6.7 Get User File List 54
6.8 Connection List 54
6.9 Browse User List 55

6.10 Add files to download queue. 55
6.11 Download Queue 56
6.12 Finished Queue 57
6.13 File Searching 57
6.14 Shared Files 58
6.15 Hashing Progress 59
6.16 Start Personal Chat 59
6.17 Secure Chat 60
6.18 RSA Public Keys of the Users 60
6.19 Blocking Users 61
6.20 Blocking Users 61

7.1 Wi-Fi (802.11g) Network: High Utilization, 2 Connections, 256 KB Buffer 1655 KBps 64
7.2 Wi-Fi (802.11g) Network: Low Utilization, 2 Connections, 256 KB Buffer, 1792 KBps 64
7.3 Gigabit LAN, 2 Connections, 256 KB Buffer 65
7.4 Gigabit LAN, 2 MB Buffer 65
7.5 Gigabit LAN, 16/24 Connections, 512KB/2MB Buffer 66
7.6 Gigabit LAN, 16 Connections, 1 MB Buffer, 52.88 MBps 66

Chapter 1

Introduction

1.1 Background

Over the years several peer-to-peer (P2P) protocols and applications have evolved. The basic need is to overcome the limitations of centralized servers. P2P file sharing has become very popular because it is cost effective, fault tolerant and has good performance. The most common protocols for P2P file sharing over the internet are the BitTorrent [4] , Gnutella [9] and eDonkey [16]. BitTorrent works particularly well over the internet with millions of users. But, in a closed network it is not effective because of certain properties of BitTorrent. To take care of these problems certain protocols were developed, 'Direct Connect' [7] is a common protocol and has many implementations; most importantly DC++, it is centralized and is used in thousands of networks worldwide.

But, 'Direct Connect' was not originally designed with security in mind. And, as a result it has some problems which cannot be directly addressed. The problems include:

- Inability to guarantee the identity of a user
- Authentication of users
- User bans are not guaranteed
- Limiting shared files to a set of users
- Personal chat is not fully secure

1.2 Problem description

To address some of the issues of security, and to add to the feature set of the previous protocols, a new protocol, '**Direct Share**' is designed with security, speed and efficiency in mind. It addresses many of the issues with sharing files effectively over a network.

1.2.1 Protocol Features

Direct Share Protocol has the following features:

- Proper authentication and identity management
- Secure communications and chat
- Parallel multi-part download for huge files
- Hashing using SHA-1 [5]/SHA256 [6] for error free file downloads
- Server-Client protocol is currently text based
- Client-Client protocol is binary with fields
- On the fly Compression for better network usage

1.2.2 Client Features

Direct Share Client has the following features:

- Fast and easy sharing of files and folders
- Real-time file sharing
- Chunk based downloading; multiple parts are downloaded form different users if available and the parts are joined to get the complete file.
- Fast file searching using tokenized Levenshtein [11] distance based matching algorithm
- Proper file hash verification and re-downloads for failed chunks
- Quick-LZ [12] based compression for file lists and other compressible packets
- Very secure encrypted and authenticated private chat
- Credits system to promote user behavior and better network utilization

After protocol design, a full reference implementation of a client and server is also done. The implementation of both the server and client is done in `C#`, using .NET Framework 3.5.

Chapter 2

General Concepts

2.1 Peer-to-peer (P2P) Networks

There are two types of P2P networks structured and unstructured. In this thesis we will focus on the unstructured P2P networks. Unstructured P2P networks have no restriction on the overlay of the network. Any peer can connect to any other and there are no central nodes. But, practical unstructured P2P networks have some degree of centralization.

- **Pure P2P systems:** In these networks all nodes are considered equal and there are no preferred nodes. Examples: Freenet [3], early Gnutella [9]
- **Centralized P2P systems :** In these networks a central server does the indexing of the users; i.e. it tells the users about the other nodes in the network. After that, the users communicate between themselves. Normally, the users don't need to have any structure. Examples: Napster
- **Hybrid P2P systems:** These systems are normally decentralized, but, allow to have some central nodes for network, these nodes forward download and search requests for other nodes. Examples: Kazaa, Gnutella2

2.1.1 Distributed Hash Tables (DHTs)

Distributed Hash Tables (DHTs) are decentralized distributed systems which is like a large hash table distributed over a large number of participating nodes. Any node can query the DHT and retrieve the value corresponding to a key. The benefit is that this process is very efficient and does not put a large load on the network. DHTs are decentralized and the distributed and replicated over a large number of nodes. As the hash table is distributed, any node addition or removal causes very small disruption, which can be easily handled. This feature allows very large self-sufficient networks to be formed.

2.1.2 Centralized P2P File Sharing Networks

In a centralized P2P system a central node is responsible for indexing the nodes and telling the nodes about each others presence. The nodes then coordinate among themselves to download content. The search is partially controlled by the central node (server).

The different functions and features of a P2P file sharing network:

- **Listing Peers:** In larger networks spread over the internet, listing is not an option as there may be a very large number of users. But, in networks aimed at smaller number of users like DC/ADC [7] clients, listing nodes/peers is pretty straightforward. The client just needs to send the server a command and the server will respond with the list of all connected nodes.

- **Search:** Search queries are normally handled by the server. For search there exists two techniques.

  - **Centralized Index:** The server maintains a centralized index for all files for all the client nodes. Whenever a search request arrives at the server, it searches for the file in the stored index, If results are found, it sends a response as a list of files and corresponding users (client nodes having those files). Such design is very hard to scale up.
  - **Distributed:** In a distributed search. The server forwards the search request to all or a subset of the connected clients which then processes the request and replies to the original sender. The reply consists of a list of files; which the requesting client can use to get the search result. The whole process is distributed, but, caused a lot of traffic over the network.

- **Downloading Files:**

  For downloading files the following is required:

  - File Name and directory information
  - File Size and File Hash / File Hash Tree
  - Peers having the file and their connection information (IP/Port)

  A file list / search response contains all of the required information. To start a file download, the peer has to connect with other peers having the file and request the file using a predefined command set. Normally the file is downloaded in parts and the different parts are requested from different peers (in case multiple peers are available). After a part is downloaded, the part is hashed and the hash is verified with the expected hash in the file descriptor. If hash verification fails, the part is re-downloaded.

- **Chat:** There are normally two types of chat.

- **Global Chat / IRC Chat:** In this type of chat, the chat is sent to the central server and the central server broadcasts the chat to all the connected peers.
- **Personal Chat:** Personal chat can be of two types.
  * **Peer-Server-Peer:** In this mode the chat is sent to the server, which then forwards it to the required destination peer.
  * **Peer-Peer:** In this mode a connection is made between the two clients which then share the chats in a normal manner.

- **Listing of shared files:** File listing can be done in certain networks, there are two ways:
  - **Centralized:** In this, the central server maintains a list of file shared by each of the clients. Whenever, any of the peer needs a file listing for any other peer; it can request the server for the file list, and the server will respond with the file list.
  - **Client-Client:** The server does not contain the list of files. The peer can directly request any other peer on the network, which will then respond with its file list.

2.2 Hash Trees

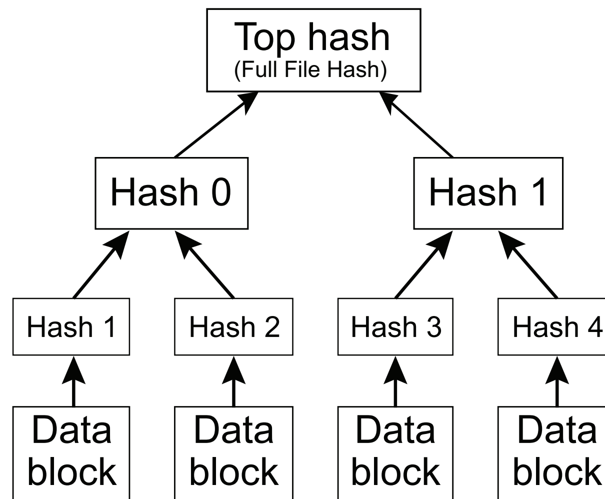

Figure 2.1: Hash Tree

Hash trees are trees in which all the non leaf nodes are labeled with the hash of its child nodes. The root node is called the 'Top Hash'. Any method can be used to join the hashes from the child nodes; methods like concatenation or binary addition may be used.

Hash trees are very useful in verifying the integrity of very large content or data structures. The benefit is that, the hashes from the data blocks can be calculated in parallel or separately, and then the individual hashes can be joined to get the hash of the complete data.

2.3 Cryptographic Hash Function

A cryptographic hash function is any algorithm which transforms an input of a variable length to an output of a fixed length bit string. The functions are designed in such a way that a single bit change in the input bit results in a drastic change in the output bits. The hash functions have the following properties.

- **Collision Resistance:** It is computationally in-feasible to find any two inputs i and i' which hash to give the same output. $H(i) = H(i')$.
- **Pre-image Resistance:** For any given hash value j it is computationally in-feasible to find an input i such that $H(i) = j$.
- **Second Pre-image Resistance:** For a given input i it is computationally in-feasible to find another input i' which hash to give the same output such that $H(i) = H(i')$.

Examples of cryptographic hash functions include MD5, SHA1, SHA2 etc.

2.4 RSA

RSA [13] being a public key crypto-system has two keys, the Public key and the Private key. The Encryption is done using one and the decryption is done using the other. Normally, the encryption is done using the Public key and the decryption is done using the Private key. The RSA modulus (explained below) length is called the key length of the cipher. The currently largest factored prime number had 768 bit. As the security of RSA depends on the factoring problem, using a modulus of 1024 bits is a bare minimum. It is recommended to use at least 2048 bits for good security. 4096 bit is pretty much unbreakable, anything beyond 4096 bits is over the top and would also be very slow.

2.4.1 Key generation

1. Choose two large random prime numbers P and Q of similar length.
2. Compute $N = P * Q$. N is the modulus for both the Public and Private keys.
3. $\Psi = (P - 1)(Q - 1)$, Ψ is also called the Euler's totient function.
4. Choose an integer E, such that $1 < E < \Psi$, making sure that E and Ψ are co-prime. E is the Public key exponent.
5. Calculate $D = E^{-1}(mod\Psi)$, normally using Extended Euclidean algorithm. D is the Private key exponent.

2.4.2 Encryption

1. Convert the data bytes to be encrypted, to a large integer called $PlainText$.
2. $CipherText = PlainText^E (modN)$
3. Convert the integer, $CipherText$ to a byte array, which is the result of the encryption operation.

2.4.3 Decryption

1. Convert encrypted data bytes to a large integer called $CipherText$.
2. $PlainText = CipherText^D (modN)$
3. Convert the integer, $PlainText$ to a byte array, which is the result of the decryption operation.

2.4.4 OAEP

RSA cannot be used directly, we need to perform padding to make sure the data is secure. OAEP (Optimal Asymmetric Encryption Padding) [1] is used to encrypt all the data. It was introduced by Bellare and Rogaway and standardized as RFC2437.

Chapter 3

Protocol Design and Description

In a centralized P2P network there can be two types of communication

- Server-Client
- Client-Client

The server-client communication is text based and the client-client communication is binary and uses a completely different structure. The protocol description will be done in two parts.

3.1 Client - Server

- All the server to client communication is text based, the terminator is the newline character (codepoint 0x0A).
- The parameters are separated by the "|" pipe character.
- The DATA parameter contains the payload for the packet (optional).
- The encoding used is UTF-8

3.1.1 Server to Client packet format

These are the packets sent from the server to the clients.

0	1	2	3	4	5
TYPE	PRID	MAC	NAME	DATA	EOP

3.1.2 Client to Server packet format

These are the packets sent from the clients to the server.

0	1	2	3	4	5	6	7
TYPE	PRID	NAME	MAC	DATA	LISTEN	TOKEN	EOP

Parameter	Description
TYPE	It defines the type of packet, i.e. it designates what the message should do. Its a character string.
PRID	It is the unique 20 byte identifier for the client. The identifier is created during application installation. It is used to recognize all the nodes in the network. Every client application must have an unique identifier associated with them.
NAME	The nick of the client.
MAC	MAC address of the client
DATA	The payload. The format depends on the TYPE field.
LISTEN	The listen port of the client. *Default Value : 54800*
TOKEN	A 20 byte token encoded in Base64 for authenticating communications; before authentication this field is a null string.
EOP	3 byte string, "EOP".

3.1.3 Identity of Clients

Public Registered ID (PRID), Client Secret (CSID), Session Authentication Token (ST) and Session GUID(SGUID) are used for determination of client identity. The protocol has two modes, "Secure Mode" and "Normal Mode".

3.1.3.1 Public Registered ID (PRID)

PRID's are global and public. They identify a client uniquely in the network. PRID is generated while application installation using a Cryptographically Secure Random Number Generator. The Client Secret later assigned is assigned on a per-PRID manner by the server. The length of PRID is 20 bytes.

3.1.3.2 Client Secret (CSID)

- **Secure Mode:** In this mode all users need to have an account associated with an E-Mail address. To keep the system secure and have unique identity, a CSID is generated during account creation and is associated with the E-Mail address. This association is stored in a database permanently. The CSID is communicated securely to the client by E-Mail. During authentication the client does a random hash based handshake with the server and an authentication token is generated and used for the rest of the session.
- **Normal Mode:** CSID is not required.

3.1.3.3 Session Authentication Token (ST)

- **Secure Mode:** This token is a 32 byte token generated by the server and issued to the client after completion of the authentication operation. The client is required to send this token with every command for which authentication is required. This token is generated and issued at every authentication request. The validity is currently infinite. But, only one single token can be valid for every PRID. Multiple instances cannot use the same token.
- **Normal Mode:** ST is not required.

3.1.3.4 Session GUID(SGUID)

All server-client connections are initiated by the client. The client generates a 32 byte random GUID. This GUID is used for the entire duration of the session; this is mandatory. SGUID must be generated randomly, but, there is no requirement for it to be unique across the network.

3.1.4 Server to Client packet TYPES

3.1.4.1 HELLO

TYPE: COMMAND
RESPONSE: HELLO

These are periodic packets sent form the server to make sure the client is connected, and are send every 30 seconds. If a client does not respond to a hello command after 2 successive requests, it is removed from the currently connected users and all the data associated with the current session for that user is purged.

The response is a HELLO response with the required HELLO_DATA.

3.1.4.2 CONNECT_AUTH_INIT

TYPE: COMMAND
RESPONSE: CONNECT_REQ

Whenever a client gets a CONNECT_AUTH_INIT command, it means that the server is operating in the "Secure Mode" and is requesting the client to authenticate. If the client supports the "Secure Mode" it should send a CONNECT_REQ response. Otherwise, it should inform the user that the secure mode is not supported and disconnect. If the client stays connected it will not be able to perform any other than the very basic commands which don't need authentication.

3.1.4.3 ACCOUNT_NON_EXISTENT

TYPE: COMMAND
RESPONSE: ACCOUNT_CREATE_REQUEST (User prompted)

This is sent from the server in "Secure Mode", if the account for the corresponding PRID is not created. The client should prompt the user to create a new account by sending the ACCOUNT_CREATE_REQUEST command.

3.1.4.4 AUTH_RESPONSE

TYPE: RESPONSE
DATA: AUTH_RESPONSE_DATA

This is response sent from the server after a AUTH_CHALLENGE_RESP (Authentication Challenge Response) command from the client.

There are two possible responses.

3.1.4.4.1 Authentication Passed

Index	Parameter Name	Description	Data Type
0	PASS	"PASS"	STRING
1	GUID	32 byte GUID for AUTH_CHALLENGE_RESP	HEX
2	AUTH_TOKEN	32 byte AUTH_TOKEN for the session.	HEX

3.1.4.4.2 Authentication Failed

Index	Parameter Name	Description	Data Type
0	FAIL	String "FAIL"	STRING
1	CONN_STATUS	"CONNECTED" if the client is connected and has already received a CONNECT_REQ, "NO_CONNECT" otherwise.	STRING
2	<NULL>	String "<NULL>"	STRING

3.1.4.5 AUTH_CHALLENGE

TYPE: RESPONSE
DATA: AUTH_CHALLENGE_DATA
RESPONSE: AUTH_CHALLENGE_RESP

This is the response sent from the server after a CONNECT_REQ (Connect Request) command from the client. The format of AUTH_CHALLENGE_DATA is as follows. The separator for the parameters is the comma "," character.

3.1.4.5.1 AUTH_CHALLENGE_DATA

Index	Parameter Name	Description	Data Type
0	GUID	GUID for the current authentication operation	HEX
1	AUTH_RANDOM	32 bytes of randomly generated data	HEX
2	SERVER_PUBLIC_KEY	Microsoft RSA CSP Compliant 2048 bit RSA Public Key BLOB for the server's public key encoded using BASE64	BASE64

After receiving the AUTH_CHALLENGE_DATA the client is expected to use the PRID, AUTH_RANDOM, CSID and E_MAIL with the AUTH_HASH_GEN (Authentication Hash Generation algorithm) (given below) to generate the EXPECTED_HASH and reply with an AUTH_CHALLENGE_RESP packet. The further communications are expected to be encrypted using SERVER_PUBLIC_KEY.

3.1.4.6 INSUFFICIENT_SHARE_SIZE

TYPE: INFO

DATA: Required Share Size, INT64

This command is sent from the server to notify the client that the share size is less than the minimum set by the server admin. This may imply that certain features will not be available until more files are shared. The client application is supposed to notify the user in some way.

3.1.4.7 USER_BANNED

TYPE: INFO

This command is sent from the server to notify that the client is banned from connecting. This means the client should inform the user and disconnect form the server.

3.1.4.8 MAIL_RESPONSE

TYPE: INFO

DATA: MAIL_RESPONSE_DATA, BASE64 encoded UTF-8 String.

After an ACCOUNT_CREATE_REQUEST an ACCOUNT_CREATE_RESPONSE is quickly sent from the server informing about the state of the user account. If a mail is required to be sent, the sending is asynchronously queued, the MAIL_RESPONSE_DATA contains a base-64 encoded UTF-8 string informing whether the mail sending was a success, failure or any other information.

3.1.4.9 CONNECT_RESPONSE

TYPE: INFO

DATA: CONNECT_RESPONSE_DATA, GUID

This command is sent from the server to notify the proper reception of the CONNECT_REQ command. This informs the client that the basic connection is complete. If the server is working in "Secure Mode", the server will send an AUTH_CHALLENGE after this packet. The CONNECT_RESPONSE_DATA contains the session GUID in HEX format.

3.1.4.10 ACCOUNT_CREATE_RESPONSE

TYPE: INFO

DATA: Response String (UTF-8)

This command is sent from the server to notify the result of an ACCOUNT_CREATE_REQUEST. The ACCOUNT_CREATE_RESPONSE_DATA contains one of the following string values; the meanings are intuitive.

Success, AccountExists, EmailExists, OtherException, InvalidEmail or InvalidPRID

3.1.4.11 USERLIST

TYPE: INFO, RESPONSE

DATA: USERLIST_DATA , BASE64-QUICKLZ

This lists all the users in the current network. The USERLIST_DATA contains a UTF-8 encoded info string which is compressed using QuickLZ compression and then encoded using base-64.

USERLIST_DATA contains information about multiple users in the form of USERLIST_ENTRY entities separated by a "|" pipe character.

3.1.4.11.1 USERLIST_ENTRY

The different parameters are explained below, the parameters are separated by a "," comma character. To get the correct value of time, the values LAST_LOGIN, LAST_SEEN & JOIN_DATE should be converted to local time.

Index	Parameter Name	Description	Data Type
0	PRID	PRID of the user	HEX
1	NICK	Nick name / Screen name	STRING
2	MAC	MAC Address	STRING
3	SHARE	Share size in bytes	INT64
4	IP	IP Address	STRING
5	LISTEN	Listen Port	INT32
6	UPLOAD	Total upload size (Bytes)	INT64
7	DOWNLOAD	Total download size (Bytes)	INT64
8	CREDITS	Total credits (Score)	INT64
9	LAST_LOGIN	Last login date in UTC long	INT64
10	LAST_SEEN	Last seen date in UTC long	INT64
11	JOIN_DATE	Join date UTC in long	INT64
12	RSA_PUBLIC	RSA Public Key (Microsoft RSA CSP Compliant 2048 bit RSA Public Key BLOB)	BASE64

3.1.4.12 CHAT_BROADCAST

TYPE: RESPONSE

DATA: CHAT_BROADCAST_DATA , BASE64

This packet is used to broadcast a chat to the clients. The client is expected to display the chat in a single window, appending every chat at the end.

3.1.4.12.1 CHAT_BROADCAST_DATA

The different parameters are explained below, the parameters are separated by a "|" pipe character. The parameter description is as follows:

Index	Parameter Name	Description	Data Type
0	TYPE	TYPE of chat	CHAR
1	SENDER_NICK	Nick name / Screen name	STRING
2	GUID	32 Byte GUID	HEX
3	CHAT_BASE64	Chat in UTF8 encoded as BASE64	BASE64

The description for the TYPE field in CHAT_BROADCAST_DATA is as follows:

CHAR	TYPE	Description
"0"	NORMAL	Standard Chat
"1"	ADMIN	Chat from Admin
"2"	SUPERUSER	Chat form a Super User

3.1.4.13 PERS_CHAT_FWD

TYPE: RESPONSE

DATA: PERS_CHAT_FWD_DATA , BASE64

It is a personal chat from a sender forwarded by the server.

3.1.4.13.1 PERS_CHAT_FWD_DATA

The different parameters are explained below, the parameters are separated by a "|" pipe character. The parameter description is as follows:

Index	Parameter Name	Description	Data Type
0	GUID	GUID for the chat	HEX
1	FORMATTED_SERVER_PAYLOAD	Private chat payload to be decoded	BASE64

3.1.4.13.2 FORMATTED_SERVER_PAYLOAD

The different fields in the FORMATTED_SERVER_PAYLOAD structure

Index	Parameter	Length	Data Type
0	SERVER_DATA_R_LENGTH	2	BYTE
1	SERVER_DATA_R	SERVER_DATA_LENGTH	OCTETS
2	CHAT_DATA_LENGTH	2	BYTE
3	CHAT_DATA	DEST_PAYLOAD_LENGTH	OCTETS

3.1.4.13.3 SERVER_DATA_R

The different fields in the SERVER_DATA_R structure. This structure must firstly be decrypted using the recipient private key while decoding. This data is added to the chat packet by the server. Email ID is added if it is requested by the sender. Else only the sender PRID is sent.

Index	Parameter	Description	Data Type
0	PRID_SENDER	PRID of the sender	OCTETS
1	ID_STRING	Email ID of sender (On demand)	STRING

3.1.4.13.4 CHAT_DATA

The different fields in the CHAT_DATA structure.

Index	Parameter	Length	Data Type
0	SIGN_DATA_LENGTH	2	BYTE
1	SIGN_DATA	SIGN_DATA_LENGTH	OCTETS
2	ENCR_CHAT_LENGTH	2	BYTE
3	ENCR_CHAT	ENCR_CHAT_LENGTH	OCTETS

3.1.4.13.4.1 SIGN_DATA

Every chat is signed by the sender and is verified at the recipient against alteration. The SIGN_DATA field contains the signature corresponding to the current chat. The signature is a standard RSA2048 signature, the hash for the signature is done using SHA256. The data to be verified is obtained by concatenating the recipient PRID and the ENCR_CHAT.

3.1.4.13.4.2 ENCR_CHAT

This contains the ENCR_CHAT structure. Prior to decryption, it must be decrypted using the recipient RSA private key.

Index	Parameter	Description	Data Type
0	SHARE_EMAIL	Specifies whether mail sharing was enabled. 0xA5 (Enabled), 0x5A (Disabled)	BYTE
1	CHAT_STATE	Current client state	BYTE
2	CHAT_TEXT	Chat Text in ASCII	STRING

3.1.4.13.4.3 CHAT_STATE

This is the CHAT_STATE enum. This specifies the current state of the client.

Value	Parameter	Description
1	Typing	The user is currently typing.
2	Waiting	No activity for the last 10 seconds.
4	Active	User online and active but no typing.
8	Inactive	User online, but no activity for 3 minutes.
16	Seen	User online chat panel is visible (top of screen).
32	Received	Chat was received and decoded by the client. This is used as acknowledgement.

3.1.4.14 CLIENT_VERSION

TYPE: INFO, RESPONSE

DATA: CLIENT_VERSION_DATA , XML string encoded with BASE64

This informs the client about the current version. If the installed version is lower than the current, the client must upgrade.

3.1.4.14.1 CLIENT_VERSION_DATA

The standard format is provided below.

```xml
<VersionInfo ClientVersion="1.07" ProtocolVersion="1" UpdateInfo="Released : 8th March 2013">
    <File Name="DirectShareClient.exe"/>
    <File Name="Readme.rtf"/>
</VersionInfo>
```

Attribute	Description
ClientVersion	The current version of the client.
ProtocolVersion	The current version of the protocol.
UpdateInfo	Update notes regarding the current update.
Name	Name of the file to be files liable to up-gradation.

After receiving this packet, of the current client version is lower than specified in the XML, an UPDATE_REQUEST is sent to the server. The server would then respond with a UPDATE_RESPONSE.

3.1.4.15 UPDATE_RESPONSE

TYPE: RESPONSE

DATA: UPDATE_RESPONSE_DATA , XML string encoded with BASE64

This packet contains information and files to upgrade the client to the latest version.

The client is expected to save the contents of the string file in a local file, and launch a program which can decode this packet and finish the upgrade process.

The format of the UPDATE_RESPONSE_DATA string is :

```
<VersionInfo ClientVersion="1.08" ProtocolVersion="1" UpdateInfo="Released : 8th March 2013">
<File Name="DirectShareClient.exe" Hash="3B704D562E75B07EEC406DF76AE42DE1C29DAF3A">BASE64 ENCODED FILE</File>
<File Name="Readme.rtf" Hash="F01D916D413D9B6B109D191E026BD9E3BBC2CEF3">BASE64 ENCODED FILE</File>
</VersionInfo>
```

Attrib / Content	Description
ClientVersion	The current version of the client.
ProtocolVersion	The current version of the protocol.
UpdateInfo	Update notes regarding the current update.
Name	Name of the file to be files liable to up-gradation.
Hash	The SHA-1 hash of the contents of the file.
Content	The content of the file to be upgraded in BASE64 compressed using QuickLZ

The updater application should verify the hashes before upgrade. Only, if the hashes are different, the file should be upgraded. After writing the file to disk. The application must recheck the hash. If any discrepancy is found, the user should be informed.

3.1.4.16 REQUEST_AUTH

TYPE: INFO

DATA: FAIL_COMMAND, STRING

This packet informs that the client is currently not authenticated. As a result it is not able to execute the command FAIL_COMMAND. This means the client should try to inform the user to authenticate or create a new account if it does not previously exists.

3.1.4.17 SEARCH_REQUEST

TYPE: COMMAND

DATA: SEARCH_REQUEST_DATA, STRING encoded using BASE64

The format of the string in SEARCH_REQUEST_DATA is defined as:

[(**PRID:IP:PORT)|BASE64(TYPE|TERM|GUID)**]

The parameters have the following meanings:

Parameter	Description	Format
PRID	PRID of the client.	HEX
IP	IP address of the search requester. Example: 192.168.52.255	STRING
PORT	Listen port of the requester.	INT32
TYPE	Type of search.	STRING
TERM	The search query.	STRING
GUID	GUID of the search. Used to uniquely identify the search requests and responses. All the transmitted packets corresponding to a single search will have the same GUID. This includes packets broadcast-ed from the server and the replies from different clients.	HEX

3.1.4.17.1 TYPE

The possible values of the TYPE parameter:
All, Documents, Programs, Video, Music, Compressed, Folder, Picture, Executable & FileHash. The client is expected to filter out files based on the TYPE for search. **FileHash** is a special type; it is the "Top-Hash" of the hash tree of the file. It is used to query for the file directly without initiating a full search. In this case the TERM is expected to contain the HEX string of the hash. As a HashTable is expected to be maintained for all shared files, the execution is very fast.

3.1.4.18 DISCONNECT

TYPE: COMMAND

This means the server wants the client to be disconnected. No reason is provided for the same. The client must follow this and close the connection to the server.

3.1.5 Client to Server packet TYPES

3.1.5.1 HELLO

TYPE: COMMAND,RESPONSE

DATA: HELLO_DATA

The response to a HELLO from the server is is a HELLO response with the required HELLO_DATA.

Hello packets are used to show change in status and also any pending upload/download updates. The design rationale for making the hello packet so complex is that, the hello packet needs to be sent every 30 seconds, sending the common updates with this packet will avoid the sending to multiple packets for updates, leading to lower network load.

If the client is connected, but, not authenticated, and the server is running in the "Secure Mode"; a CONNECT_AUTH_INIT should be sent, so that the client can initiate the authentication process.

3.1.5.1.1 HELLO_DATA

Index	Parameter Name	Description	Data Type
0	TOTAL_SHARE	Total size of shared files	INT64
1	COMM_PORT_LISTEN	Client listen port	INT32
2	SGUID	GUID for the Server-Client	HEX
3	DOWNLOAD_ACK [DATA]	See Client-Client Section	STRING

3.1.5.1.2 DOWNLOAD_ACK: Download Acknowledgment Packet

This is used to acknowledge the successful download of a packet. This packet is sent from the *downloader* to the *uploader* who then sends it to the server along with the HELLO packet. The parameters are separated by the ":" colon character.

Index	Parameter Name	Description	Data Type
0	PRID	PRID of the user downloading the part	HEX
1	START	Start Index in the file	INT64
2	LENGTH	Length of the packet	INT32
3	ROOT_HASH	ROOT_HASH of the concerned file	HEX

3.1.5.1.3 DOWNLOAD_ACK [DATA]: Download Acknowledgment Data

Its an array of DOWNLOAD_ACKs. The individual DOWNLOAD_ACKs are re-encoded as BASE64 strings and concatenated, they are separated by the "|" pipe character.

Example : BASE64(ACK-1) | BASE64(ACK-2) | BASE64(ACK-3) . . .

3.1.5.2 CONNECT_REQ

TYPE: COMMAND

DATA: GUID of the connection, HEX

After receiving the CONNECT_REQ command, the server checks whether the client has enough share size. If not, it sends the INSUFFICIENT_SHARE_SIZE command and stops the authentication process.

Then, the server checks if an USER_ACCOUNT corresponding to the PRID already exists:

- **If YES:** The server generates a AUTH_RANDOM using a Cryptographically Secure Random Number Generator. Then it creates the AUTH_CHALLENGE packet and sends it to the client. Meanwhile the server also generates the AUTH_TOKEN and calculates the EXPECTED_HASH using the AUTH_HASH_GEN algorithm and maintains them for use in the next part of client authentication.
- **If NO:** The server sends the ACCOUNT_NON_EXISTENT to the client and stops the authentication process.

3.1.5.3 AUTH_CHALLENGE_RESP

TYPE: RESPONSE
DATA: AUTH_CHALLENGE_RESP_DATA, STRING

This packet is the response to the AUTH_CHALLENGE sent from the server.
The AUTH_CHALLENGE_RESP_DATA is decoded and the client is authenticated. The parameters are separated using a "," comma character. The parameters details are:

Index	Parameter Name	Description	Data Type
0	GUID	GUID for the current authentication operation	HEX
1	RESULT_HASH	20 bytes of response to the hash challenge	HEX
2	CLIENT_PUBLIC_KEY	Microsoft RSA CSP Compliant 2048 bit RSA Public Key BLOB for the server's public key encoded using BASE64	BASE64

If the RESULT_HASH returned by the user matches the previously calculated value, the user is authenticated successfully; the AUTH_RESPONSE packet is send correspondingly to complete the authentication.

3.1.5.4 CLIENT_VERSION

TYPE: COMMAND

This packet means that the client has requested for the latest client version data.
The CLIENT_VERSION response is prepared and sent in the format as described previously.

3.1.5.5 UPDATE_REQUEST

TYPE: COMMAND

This packet means that the client is requesting to upgrade to the current version.
An UPDATE_RESPONSE packet is sent in the format previously described as a result.

3.1.5.6 ACCOUNT_CREATE_REQUEST

TYPE: COMMAND
DATA: ACCOUNT_CREATE_REQUEST_DATA, STRING

This is a request to create a new account for the associated user. The DATA field contains the E-Mail address of the client.

A new user account is created if required and the result is sent as an ACCOUNT_CREATE_RESPONSE packet. An E-Mail is also sent containing the AUTH_DATA.

3.1.5.7 USERLIST

TYPE: COMMAND, VALIDATED

This means the client is requesting a new user list. As a response a USERLIST is created and sent to the client.

3.1.5.8 CHAT_BROADCAST

TYPE: COMMAND, VALIDATED
DATA: CHAT_BROADCAST_DATA, STRING

This is a chat which is required to be sent to all the connected clients. The AUTH_TOKEN is validated and the chat is processed and sent.

3.1.5.8.0.1 CHAT_BROADCAST_DATA

This is the CHAT_BROADCAST_DATA structure. This contains the chat. The separator for the parameters is the "|" pipe character.

Index	Parameter Name	Description	Data Type
0	CHAT_GUID	GUID for the chat	HEX
1	CHAT_STRING	The actual chat string in ASCII encoded in BASE64	BASE64

3.1.5.9 SEARCH_INITIATE

TYPE: COMMAND, VALIDATED
DATA: SEARCH_INITIATE_DATA, STRING

The format for the SEARCH_INITIATE_DATA is: (TYPE|TERM|GUID)

The parameters have the following meanings:

Parameter	Description	Format
TYPE	Type of search.	STRING
TERM	The search query.	STRING
GUID	GUID of the search.	HEX

3.1.5.10 PERS_CHAT

TYPE: COMMAND, VALIDATED

DATA: PERS_CHAT_DATA, STRING

This packet corresponds to a Private Message. The packet has to be decoded, decrypted, interpreted and the resultant chat data should be forwarded to the required client. After doing all the steps a PERS_CHAT_FWD packet is sent to the client in the format previously specified.

3.1.5.10.1 PERS_CHAT_DATA

The different parameters are explained below, the parameters are separated by a "|" pipe character. The parameter description is as follows:

Index	Parameter Name	Description	Data Type
0	GUID	GUID for the chat	HEX
1	FORMATTED_CHAT_DATA	Private chat payload to be decoded	BASE64

3.1.5.10.2 FORMATTED_CHAT_DATA

The different fields in the FORMATTED_CHAT_DATA structure

Index	Parameter	Length	Data Type
0	SERVER_DATA_S_LENGTH	2	BYTE
1	SERVER_DATA_S	SERVER_DATA_LENGTH	OCTETS
2	CHAT_DATA_LENGTH	2	BYTE
3	CHAT_DATA	DEST_PAYLOAD_LENGTH	OCTETS

3.1.5.10.2.1 SERVER_DATA_S

The different fields in the SERVER_DATA_S structure. This structure must firstly be decrypted using the server private key while decoding.

Index	Parameter	Description	Data Type
0	PRID_RECEIVER	PRID of the receiver	OCTETS
1	SHARE_EMAIL	Specifies whether mail sharing was enabled. 0xA5 (Enabled), 0x5A (Disabled)	BYTE

3.1.5.10.2.2 CHAT_DATA

This part contains the signature and chat data encrypted using the receiver's private key. So, its impossible to decrypt for the server. This should be re-packetized and forwarded to the receiver.

3.2 Client - Client

This section deals with the packet structure and protocol for all client-client communications. The features of the client-client communication protocol are:

- Client-Client communications are binary, based on a fixed structure.
- Both the clients maintain a state-machine associated with each connection.

3.2.1 Basic Packet Structure

All the client-client communications take place using binary packets. The packets have the following structure.

0	1	2	3	4	5	6	7	8
ATDS	TYPE	PRID	MAC	LISTEN	NAME	DATA_SZ	DATA	EOP

The fields are described in the table below:

Start	**End**	**Parameter**	**Description**	**Octets**	**Type**
0	3	ATDS	"ATDS" in ASCII	4	STRING
4	4	TYPE	TYPE Identifier Enumeration	1	BYTE
5	24	PRID	PRID of sender	20	BYTE [ ]
25	30	MAC	MAC of sender	6	BYTE [ ]
31	34	LISTEN	Listen port of sender	4	INT32 [1]
35	50	NAME	Name of sender	16	STRING
51	54	DATA_SZ	Payload length	4	INT32
55	54+D [2]	DATA	Actual Payload	D	BYTE [ ]
55+D	57+D	EOP	"EOP" in ASCII	3	STRING

3.2.2 Processing of the Packets

The processing of the packets is based solely on the TYPE field. The TYPE specifies what kind of operation is to be done. That means, the decoding and processing of the DATA field depends on TYPE.

The following kinds of TYPE packets are defined:

[1] Int32 packed as 4 bytes, lower byte first
[2] D is the length of payload

Value	Parameter
0	HELLO
1	FILELIST_REQ
2	FILELIST_DATA
3	DOWNLOAD_REQUEST
4	DOWNLOAD_REQUEST_DESCRIPTOR
5	DOWNLOAD_DATA
6	SEARCH_RESPONSE
7	DISCONNECT
8	TOO_MANY_CONNECTIONS_FORCE_CLOSE
9	FILE_NOT_AVAILABLE
10	FILE_PARAMS_INVALID_BUSY
11	DOWNLOAD_ACK
12	DOWNLOAD_NACK

3.2.2.1 Packet Description: HELLO

TYPE: COMMAND

DATA: HELLO_DATA, STRING

The HELLO packets are sent for the sole purpose of keeping the connection alive. These packets are sent every 5 seconds for an open and active connection. If HELLO packets are not received for more than 15 seconds, the connection is dropped and the resources released.

3.2.2.1.1 HELLO_DATA

This is the HELLO_DATA structure. It contains two strings separated by a "," comma character.

Index	Parameter Name	Description	Data Type
0	TOTAL_SHARE	GUID for the chat	INT64
1	COMM_PORT_LISTEN	Listen Port for incoming connections	INT32

3.2.2.2 Packet Description: FILELIST_REQ

TYPE: COMMAND

This is a request from one client to another for a list of files it is currently sharing. It has no data fields. The response would be a FILELIST_DATA XML string containing the list of shared files. The XML should follow the FILELIST format specified below.

3.2.2.3 Packet Description: FILELIST_DATA

TYPE: RESPONSE

This is a response to a FILELIST_REQ command. After receiving this packet and making sure that it was actually requested, the client should decode and show the list to the user in the form of a TreeView / or a ListView.

3.2.2.3.1 FILELIST Format

The FILELIST is defined as an XML string containing the list of files.

The following XML is an example FILELIST containing 2 files:

```
<F Name="S:\File.iso" NHash="A55BD5F7DA90642696B8A1414855D49EFCC7394F" Size="18791980"
Date="129867100127656250" Root="6417C7E5BC9B2E6DD82E8DF2219C8F1A7B99F73E" >
</F>
<F Name="P:\Downloads\Movie.mp4" NHash="6CBFDDB56B53EF8E8168651D448EBAAD7244532C" Size="97605692"
 Date="129959188445670880" Root="36C1BE1D0E5E12A9EAF171293E19267E56627534" >
<P>0CD6BB09AEFDA13CA6CA8666CCF707FBB0B1C82C</P>
<P>3A170514A0A3B3954C3BF74FF2EE2185E6D3BD18</P>
</F>
```

Attrib / Content	Description
Name	File name with path.
NHash	SHA-1 Hash of Name
Size	FileSize in bytes
Date	File Date in Int64
Root	SHA-1 hash root of the file.

The <P>or part entries contains the chunks. If the file size is less than the CHUNK_SIZE, then the <P> entries are not required to be specified.

3.2.2.4 Packet Description: DOWNLOAD_REQUEST

TYPE: COMMAND
DATA: DOWNLOAD_REQUEST_DATA

All downloads start with one client sending a DOWNLOAD_REQUEST command to another. The serving client is expected to decode the request and send a corresponding DOWNLOAD_DATA packet containing the requested data.

3.2.2.4.1 DOWNLOAD_REQUEST_DATA

This is the DOWNLOAD_REQUEST_DATA structure. It contains parameter strings separated by a "|" pipe character.

Index	Parameter Name	Description	Data Type
0	ROOT_HASH	Top hash of the hash tree of the concerned file	HEX
1	START_INDEX	Position in the file to start reading	INT64
2	LENGTH	Length of data required, starting from START_INDEX	INT64

3.2.2.5 Packet Description: DOWNLOAD_REQUEST_DESCRIPTOR

TYPE: RESPONSE
DATA: DOWNLOAD_REQUEST_DESCRIPTOR_DATA

Because there is no way to know what is going to be done while an outgoing connection is established, and downloading takes place in small chunks, a DOWNLOAD_REQUEST_DESCRIPTOR packet tells the uploading client about nature of connection during a file transfer. This way, the client can show proper progress during the transfer.

This packet is optional.

3.2.2.5.1 DOWNLOAD_REQUEST_DESCRIPTOR_DATA

The format is the same as that of the DOWNLOAD_REQUEST_DATA structure.

3.2.2.6 Packet Description: DOWNLOAD_DATA

TYPE: RESPONSE
DATA: DOWNLOAD_DATA_DATA

This packet is a response to a DOWNLOAD_REQUEST packet. It contains the requested data for a file.

3.2.2.6.1 DOWNLOAD_DATA_DATA

0	1	2	3	4	5
PLD	FILE_HASH	START	LENGTH	CONTENT	EOP

The fields are described in the table below:

Start	End	Parameter	Description	Octets	Type
0	2	PLD	"PLD" in ASCII	3	STRING
3	23	FILE_HASH	Top hash of the hash tree	20	BYTE [ ]
24	31	START	Start index in the file	8	INT64 [3]
32	35	LENGTH	Length of data requested	4	INT32
35	34+D [4]	CONTENT	Payload Data	D	BYTE [ ]
35+D	37+D	EOP	"EOP" in ASCII	3	STRING

3.2.2.7 Packet Description: SEARCH_RESPONSE

TYPE: RESPONSE

DATA: SEARCH_RESPONSE_DATA

In the current version of the protocol, all search requests have to pass from the server. Whenever a search request is sent from the server. It has to be processed and if any results are obtained, this packet is sent as a result to the original client directly.

3.2.2.7.1 SEARCH_RESPONSE_DATA

This is the SEARCH_RESPONSE_DATA structure. It contains parameter strings separated by a "|" pipe character.

Index	Parameter Name	Description	Data Type
0	USER_HASH	PRID of the sender	HEX
1	FILES_XML	Listen Port for incoming connections	BASE64 STRING
2	SEARCH_GUID	GUID for the search query	HEX

The XML format for FILES_XML is same as for the FILELIST. The client also must make sure that the GUID matches any of the previously sent search queries.

3.2.2.8 DISCONNECT

TYPE: COMMAND

This means that connection is no longer required, and informs the client to disconnect and release the resources allocated for the connection.

3.2.2.9 TOO_MANY_CONNECTIONS_FORCE_CLOSE

TYPE: INFO

This means that the other client is busy responding to multiple requests and cannot currently handle newer requests. It is expected for the connecting client to try after a predetermined

[3] Int64 packed as 8 bytes, lower byte first

[4] D is the length of payload

amount of time. On reception of such a request, the client must disconnect.

3.2.2.10 FILE_NOT_AVAILABLE

TYPE: INFO

This means that the requested file does not currently exist with the client, this may be due to a recent file system change, or it may have been removed from the share explicitly by the user. In any case, the responding user should be removed from the list of potential users having the file.

3.2.2.11 FILE_PARAMS_INVALID_BUSY

TYPE: INFO

This means that the requested file exists, but, there is a problem accessing the file part, the possible reasons are:

- **Busy File System:** Multiple read or write operations may cause longer delays in retrieving the file.
- **Invalid File Parameters:** This may happen if the request is outside file boundaries. Example: The file is having a length of 12949 bytes, but, the specified start index is greater than that value. This may happen if the file has changed after hashing.

In any case, the client should try for a few times with some time delays between operations, before failing and removing the connected client as a potential user for the file in question.

3.2.2.12 DOWNLOAD_ACK

TYPE: INFO

This is used to acknowledge the successful download of a packet. The packet is decoded and validated and is forwarded to the server on proper validation.

3.2.2.13 DOWNLOAD_NACK

TYPE: INFO

This is used to acknowledge the unsuccessful download of a packet. The packet is meant to inform the uploader that the transferred part is either invalid or the hash verification fails. If many such packets are received, the client may decide to inform the user and rehash the file to make sure the integrity of the file and hash is maintained.

3.3 Algorithm: AUTH_HASH_GEN

This algorithm is used to generate EXPECTED_HASH from multiple parameters.

3.3.1 Required Parameters

1. $EMAIL_ID$: It is the E-Mail id of the user.
2. $PRID$: PRID of the User.
3. $AUTH_SECRET$: Authorization Secret for the User (Sent using mail)
4. $AUTH_RANDOM$:Random number used for the authentication operation.

3.3.2 EXPECTED_HASH Generation

The length of the *Required Parameters* may vary. As a result the 20 first bytes from all the parameters are considered; rest are ignored.

1. $EMAIL_HASH = H(EMAIL_ID)$
2. $PRE_HASH = EMAIL_HASH \oplus PRID \oplus AUTH_SECRET \oplus AUTH_RANDOM$
3. $EXPECTED_HASH = H(PRE_HASH)$

may be either SHA1/SHA2

Chapter 4

Protocol Implementation

This chapter will focus on how to use the protocol to implement a full featured file sharing system, implementation logic of search, download, authentication, chat, and various other features of the protocol.

4.1 File Search

The search in *Direct Share* is distributed; because of the focus on medium sized networks, simple broadcast based search makes sense. Every search takes place using the following steps.

1. **User Command:** The user enters a query, selects the properties and type of search and initiates the search.
2. **Request:** The client application sends a request to the server containing the search term and properties in a format specified in the SEARCH_REQUEST command.
3. **Broadcast:** The server broadcasts the search request to all the clients. The clients then process the query independently, in parallel and create a sorted search result set.
4. **Response:** The clients who have got valid search results send the results to the original searcher as a SEARCH_RESPONSE packet.
5. **Display:** The client application creates a dynamic list and adds all the search results and displays the results as they come in. Because, every search has an associated GUID, search results for every search request can be uniquely displayed.

4.1.1 Search Algorithm: Damerau−Levenshtein

Any technique can be used to implement the search. In the current implementation *Damerau−Levenshtein [11] distance* algorithm is being used. Damerau−Levenshtein is based on edit distance between

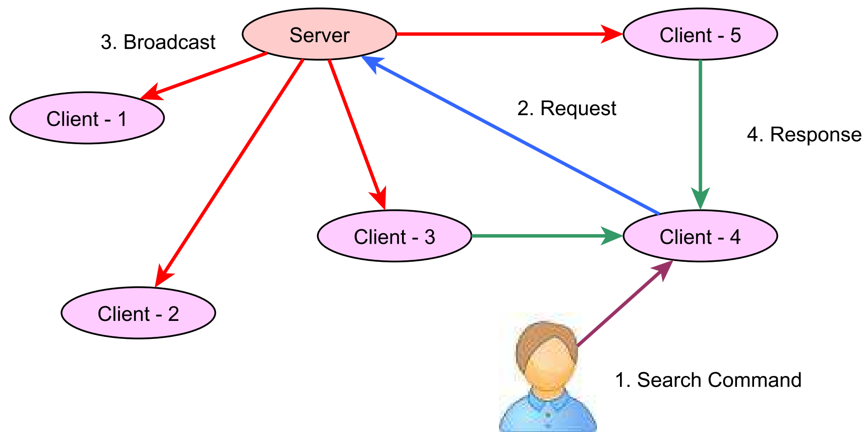

Figure 4.1: File Search Flow

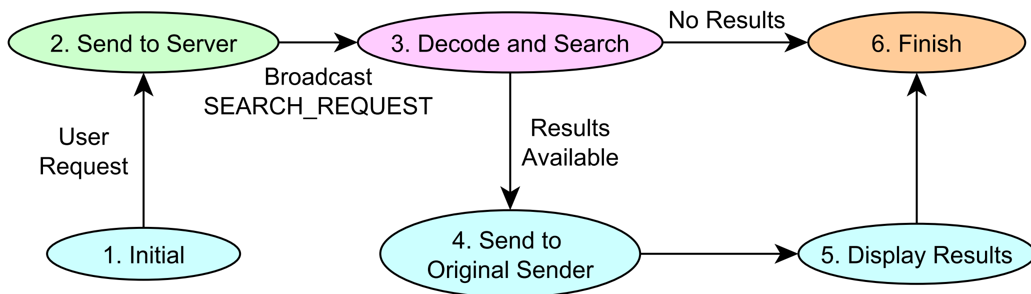

Figure 4.2: File Search State

two strings. In other words, how many *edit* operations are required to get one string from the other. The allowed *edit* operations are *insertion*, *deletion*, *substitution* of a single character or the *transposition* of two adjacent characters.

4.1.2 Search Implementation

The file search is implemented as a state machine. For every search, a new thread is initialized to handle that particular request. Every request has its own associated GUID which is randomly generated. This GUID is used to differentiate between multiple search queries.

In the current implementation, the search takes place in the following steps:

1. **User Request:** Whenever the user enters a query and starts the search, a new thread and a corresponding search GUID is created and the search begins.

2. **Send to Server:** A SEARCH_REQUEST is created from the query and selected options and send to the server. The server then broadcasts the SEARCH_REQUEST to all the connected clients.

3. **Decode and Search:** A client on receipt of a SEARCH_REQUEST packet decodes it and searches the query as per predefined techniques. Currently, a Damerau−Levenshtein based shortest distance algorithm is implemented, which allows for small errors in spellings. The results are sorted based on match score before sending.
4. **Send to Original Sender:** If results are available for the query, a SEARCH_RESPONSE packet is created, containing the results in the form of *FILELIST*.
5. **Display Results:** The client dynamically accumulates all the SEARCH_RESPONSE packets and displays the results in real-time as the results arrive.
6. **Finish:** The client waits for a fixed time listening for incoming results. After that time if no more results arrive, the search is deemed finished.

4.2 File Download

File download is one of the most important and complex operation in the protocol. The complexity is due to the requirement for reliable download of very huge files from multiple sources. The parallel downloads, state management, status updates, feedback, hash verification, and handling of failure makes it a fairly complex operation.

File download is a client-client operation, the files are downloaded in chunks, and the default chunk size is 50 MiB (*CHUNK_SIZE*). The following steps take place during simplified file download operation having a single client having the file.

1. **Initialize:** Because the download of a large file involves multiple chunks. Proper data structures must be initialized to maintain the state of individual chunks. Thread safety has to be taken into account as multiple threads may be involved, downloading the file from multiple sources.
2. **Request:** The client which wants to download the file sends a request to another client having the file by sending a DOWNLOAD_REQUEST packet. The packet contains all the information to look for the file, read, and send.
3. **Response:** The client getting a DOWNLOAD_REQUEST packet decodes it and checks whether it has the requested file. If the file is present, it is read at the required index, packetized and sent to the sender as a DOWNLOAD_DATA packet.
4. **Receive:** When a DOWNLOAD_DATA packet is received, it is decoded, and if it corresponds to a file in the download queue. The payload data is written to a buffer in sequence pre-initialized for the chunk. When all the parts on the chunk are written; a hash, CALCULATED_HASH is generated using the selected hash function. The hash is then matched with the EXPECTED_HASH. If the hash match is valid, the chunk is written to disk/storage at the proper index in the file, otherwise the chunk is discarded and re-downloaded.

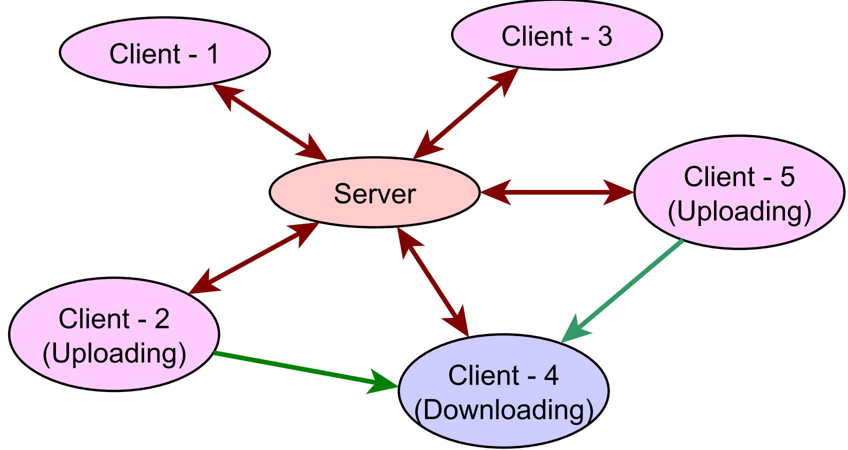

Figure 4.3: Download Flow

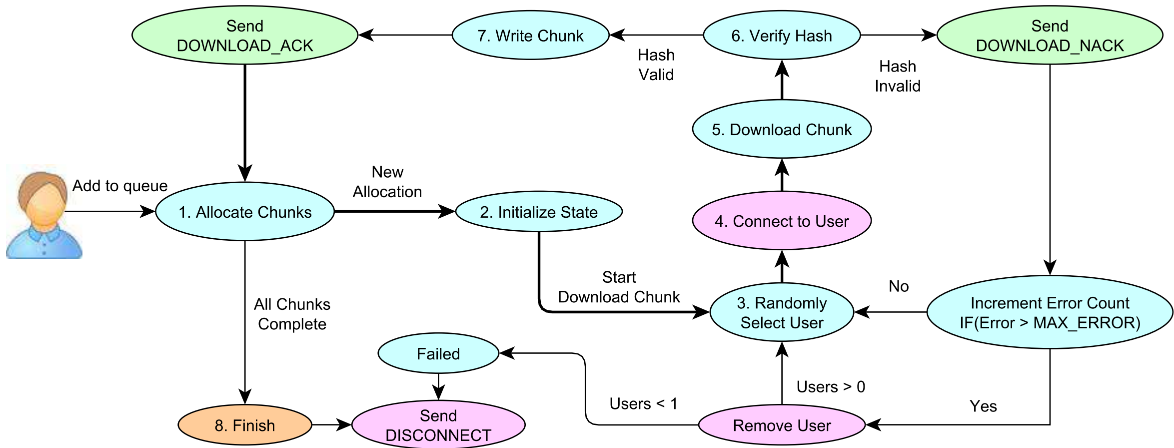

Figure 4.4: File Download: Downloader State

Two state machines are required to be maintained for downloading a file, one at the downloader side and one at the uploader side. The file is divided into *chunks* of size CHUNK_SIZE (50 MiB), the chunks are then themselves divided into *parts*. The PART_SIZE is dynamic, and is decided by the client (downloader) based on the type of network being used. Before downloading, the client initializes data structures to maintain the status and unwritten chunks.

4.2.1 File Download: Downloader State

The basic operation at the downloader end is explained as follows:

1. **Allocate Chunks:** Whenever a file is added to the queue and started. The first thing is the setting up of data structures to maintain the download of chunks. Thread safety has to be taken into account as multiple parts may be downloaded simultaneously. After initialization chunks are allocated to downloading threads in a sequential manner depending

on the number of available download slots.

2. **Initialize State:** After a chunk is allocated to a thread, parts are allocated as per available connection parameters. It is important to note that all the downloads takes place in in the form of *PARTS*, which are later joined to form the *CHUNK*.

3. **Randomly Select User:** If there are multiple users, any of of them is chosen at random.

4. **Connect to User:** After selection, the user is connected and informed about the current download requirements.

5. **Download Chunk:** The chunk that was allocated is downloaded. The *chunk* downloaded further involves the download of *parts* which is explained later. During the download, the SHA-1/SHA256 hash state is progressively updated as the parts come in.

6. **Verify Hash:** After the expected number of parts are received, the hash is calculated and matched with the expected hash.

7. **Write Chunk:** If the calculated hash matches, with the expected hash, the chunk is written to the file at the proper index.

8. **Finish:** The allocation and download of chunks continues until all the chunks are downloaded. After which, some optional operations like updating file attributes take place and the file is finally closed.

4.2.2 File Download: Downloader Chunk State

Every *chunk* is downloaded in smaller segments called the *part*. This is a sequential process The *chunk* level download takes place in the following steps.

1. **Initial:** After a download is queued, the chunks are assigned to a downloading thread. The *chunk* is a segment of the file of size CHUNK_SIZE starting from an index in the file. This position is predetermined during the file hashing.

2. **Get Next Part Index:** Every time a part of the chunk is downloaded the next part has to start from a different index. This process gives the next valid index for the chunk. The part size should depend on the current network speed; if the speed is greater then 10 MB/s, then this size can be as high as 4 MB, or as low a 128 KB for very slow wireless notworks. It should be dynamically assigned for best performance.

3. **Send DOWNLOAD_REQUEST:** Based on the assigned part size and index in the previous step, a DOWNLOAD_REQUEST is sent to request for the next part. The state machine then waits for the next response for the *uploader*.

4. **Process Response:** Depending on the response, multiple things may happen, but, if all goes well, the *uploader* responds with a DOWLOAD_DATA packet containing the contents requested.

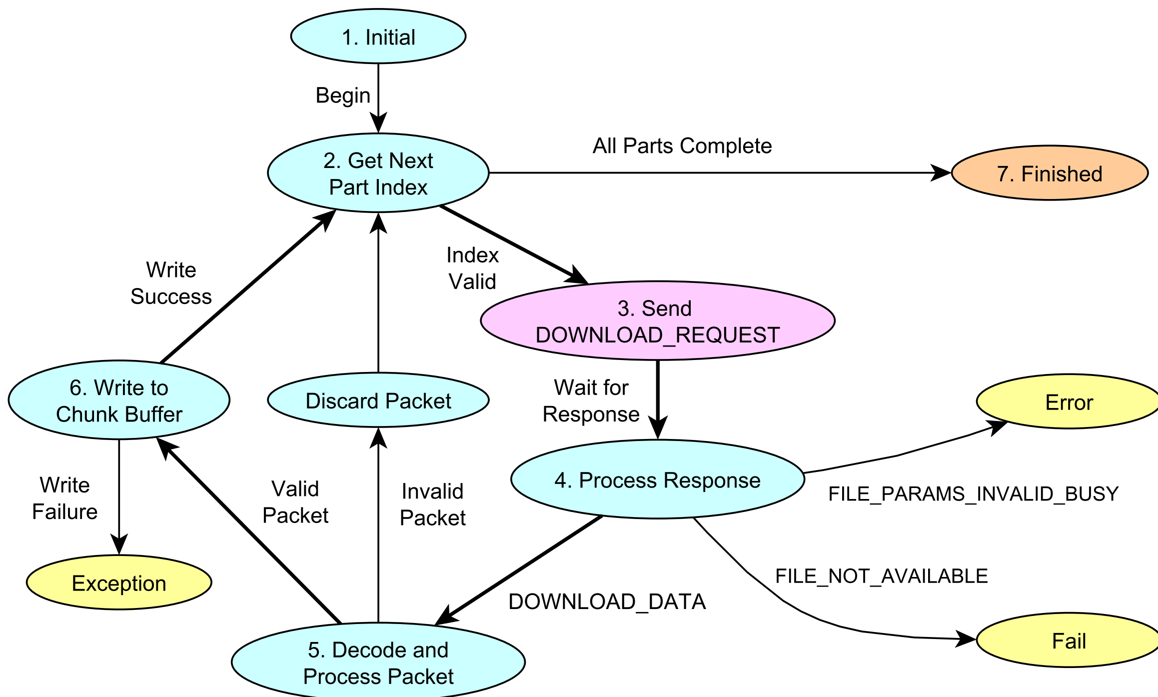

Figure 4.5: File Download: Downloader Chunk State

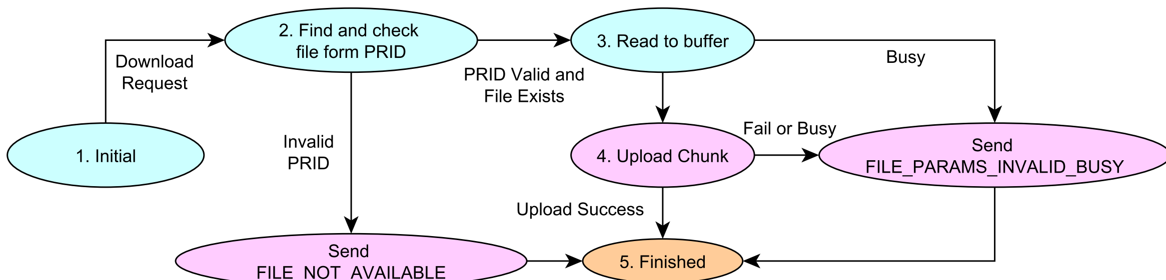

Figure 4.6: File Download: Uploader State

5. **Decode and Process Packet:** The contents of the packet is decoded and checked if it conforms to the expected format.

6. **Write to Chunk Buffer:** If the packet in the previous step is in the proper format, the data is written to the chunk buffer and corresponding data structures are updated correspondingly. If the write is a success, the SHA-1 hash which is maintained may be updated simultaneously. This process continues till all the parts complete.

7. **Finish:** After all the parts are completed successfully, the subroutine returns; and an optional DISCONNECT command may be sent to the *uploader* if it is not needed anymore.

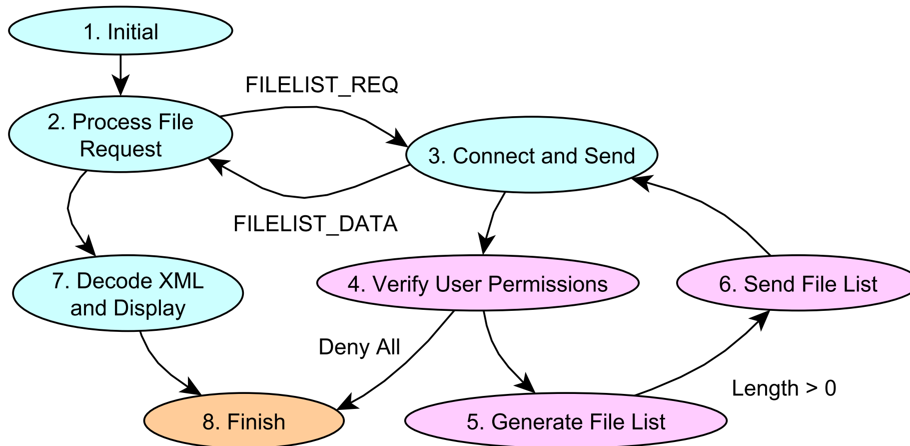

Figure 4.7: File List: Download Flow

4.2.3 File Download: Uploader State

The upload of a file takes place in the following steps:

1. **Initial:** A new connection is established for every download. After connecting the downloader sends a DOWNLOAD_REQUEST_DESCRIPTOR packet, containing information about the chunk to be downloaded. Using the packet the internal structures are updated.
2. **Find and check file from PRID:** The client maintains a list of all the files currently being shared, the requested PRID can be checked, if it does not exist in the listing or physically in the file-system, FILE_NOT_AVAILABLE is sent as a response.
3. **Read to Buffer:** If the file is available, the required chunk is read to the file buffer. Max read size can be CHUNK_SIZE.
4. **Upload Chunk:** After reading the chunk, it is uploaded in smaller units called *part* in a sequential manner. For every *part* the downloader sends a different DOWNLOAD_REQUEST command which is replied with a DOWNLOAD_DATA packet containing the data.
5. **Finish:** If all the parts are successfully uploaded, the chunk completes. The client then can disconnect or keep the connection alive for more chunks.

4.3 FILELIST

A list of file can be obtained from any client. The list contains all the shared files. The user can then see and download any file or folder from that list.

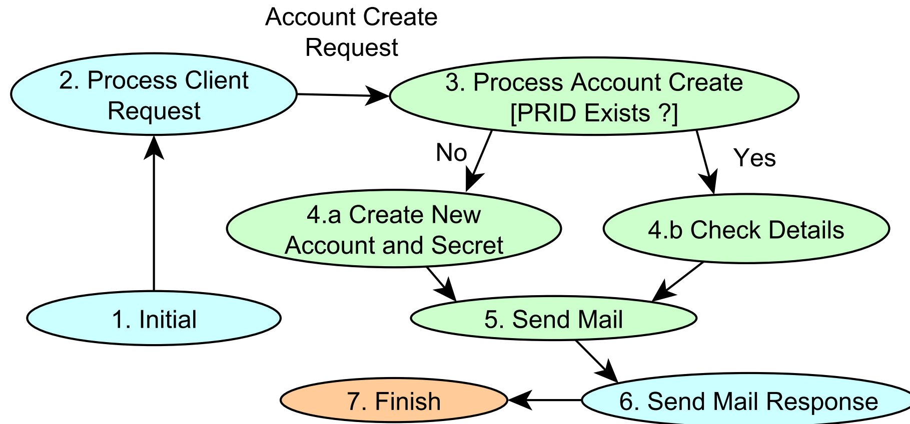

Figure 4.8: Create Account Flow

4.3.1 FILELIST Download

The downloading of the *FILELIST* is done in the following manner.

1. **Initial:** The information like IP and Port about the user to be connected is gathered.
2. **Process File Request:** The client is connected and a FILELIST_REQ command is sent to the connected user.
3. **Connect and Send:** The user makes the connection and starts processing the request.
4. **Verify User Permissions:** As a user can be blocked access to certain files. The permissions are checked for the user.
5. **Generate File List:** A file list is generated which contains the files which are explicitly shared for the user, this ensures that limited file sharing is possible. Or, a file can be shared to a limited set of users.
6. **Send File List:** The file list converted to a transportable XML format (FILELIST format is specified before), and is then sent to the client requesting.
7. **Decode XML and Display:** The application then decodes the XML and generates a TreeView or ListView or a combination of both.
8. **Finish:** The list is displayed to the user and the process ends.

4.4 Create Account

The creation of an user account takes place in the following manner:

1. **Initial:** The user manually starts the account creation process.

2. **Process Client Request:** The client creates a ACCOUNT_CREATE_REQUEST packet and sends it to the server. This packet contains the PRID of the user.

3. **Process Account Create:** The request is decoded by the server and the PRID is checked against already existing accounts in the database.

4. (a) **Create New Account and Secret:** If the PRID corresponding to the user does not exist. A new *Secret* is generated randomly and assigned to the user and the databese id updated with the information.

   (b) **Check Details:** If the Account exists the validity is checked and reconfirmed.

5. **Send Mail:** An E-mail containing the AUTH_DATA (Authentication Data) is sent to the client's E-mail address.

6. **Send Mail Response:** Mail sending is an asynchronous process and it takes some time to send the mail. After the mail sending process ends. A reply in the form of an ACCOUNT_CREATE_RESPONSE is sent to the user. The client application then displays it to the user.

7. **Finish:** This ends the account creation process.

4.5 Private Chat

Private chat allows two clients to chat securely. Every chat is encrypted, signed and verified. Because of this, the chat is very secure and even the server is not able to decrypt the chat messages between the users.

The following steps are followed to transfer a chat from one client to another.

1. The user sending the chat enters the chat and starts the process. The chat GUID is generated rendomly.

2. According to the settings selected by the user, a SERVER_DATA_S packet is created and encrypted with the Server_Public_Key.

3. The chat is encoded and encrypted and signed, for signing Client_Private_Key of the sender is used, the chat and the signature is used encrypted with the Client_Public_Key of the receiver. All this is used to make the CLIENT_DATA.

4. The SERVER_DATA_S and CLIENT_DATA is packaged to make the PERS_CHAT packet. Thereafter it is sent to the server.

5. The server decrypts the SERVER_DATA_S part of the PERS_CHAT and checks the value of SHARE_EMAIL.

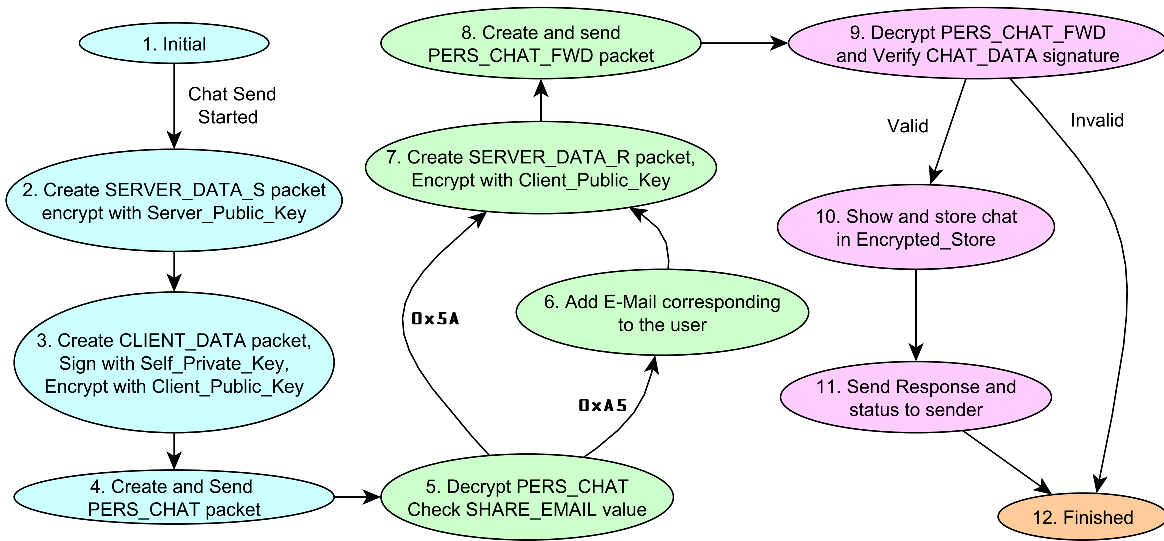

Figure 4.9: Private Chat Flow

6. If the SHARE_EMAIL part of the SERVER_DATA has the value of 0xA5 then the E-Mail id of the sender is attached with the response.

7. A SERVER_DATA_R packet is created using the response data encrypted with Client_Public_Key of the receiver.

8. A PERS_CHAT_FWD packet is created using SERVER_DATA_R and CLIENT_DATA and sent to the receiver.

9. The PERS_CHAT_FWD packet is decrypted using the Client_Private_Key of the receiver and CHAT_DATA signature is verified.

10. If the verification is successful the chat is stored in an encrypted store and displayed to the user at the same time.

11. The response containing a Received tag is sent to the user corresponding to the GUID, so that the sender knows the valid reception of the chat.

12. This marks the end of the chat sending process.

4.6 Authenticate User

The user authentication is one of the most critical operations in the functioning of *DirectShare*. It makes sure that all the users in the network are valid and prevents unauthorized access by attackers and disallowed users.

The sequence of events are shown in the figure "Authentication Sequence".

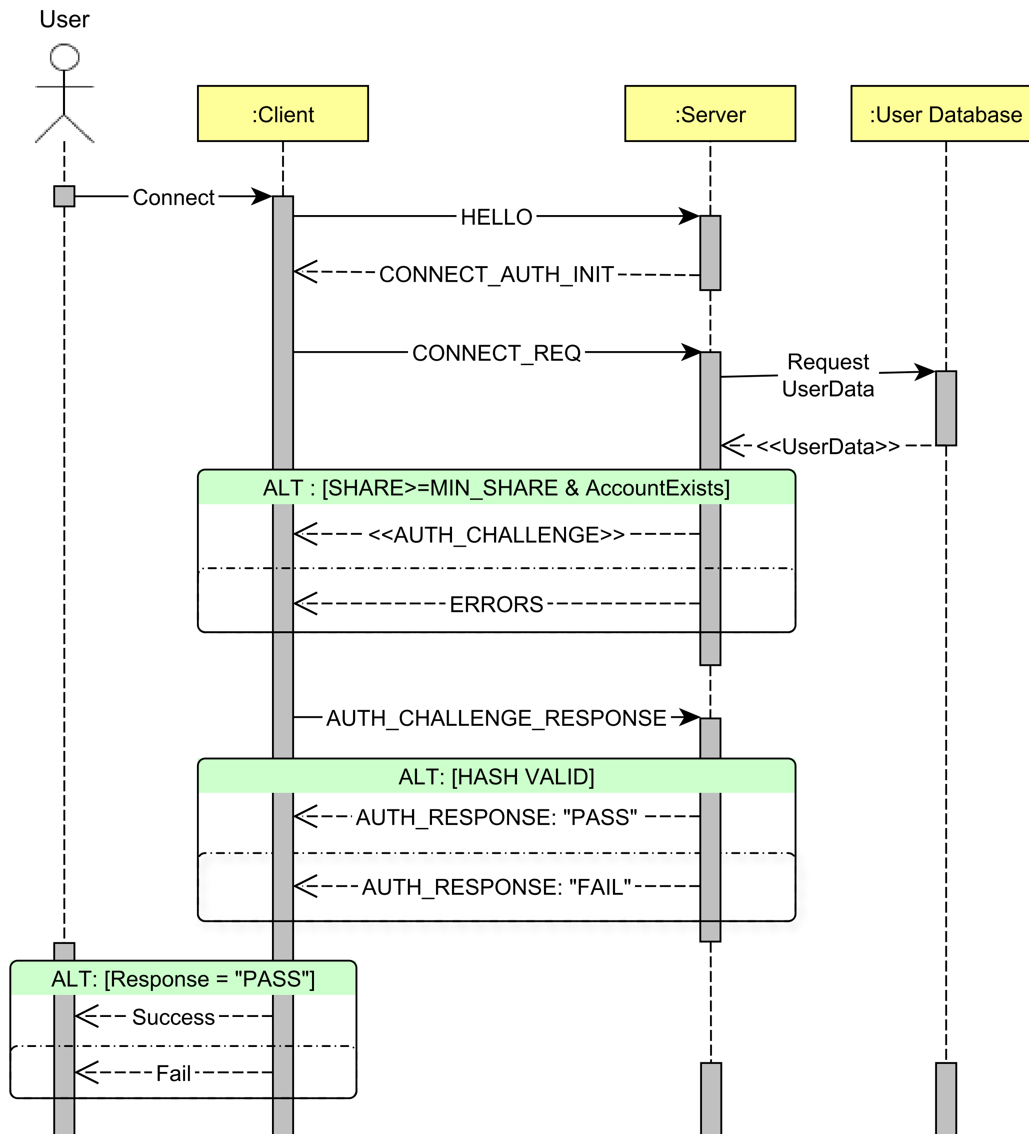

Figure 4.10: Authentication Sequence

The authentication is quite complex compared to the other operations. For it to work, two state machines are needed, one at the client side and one at the server side.

1. **Send CONNECT_AUTH_INIT:** Whenever a new client connects to the server the server checks whether the client is already authenticated. If not, it sends the CONNECT_AUTH_INIT command. This for the client means that the server is using *Secure Mode* and hence the client should start the authentication process.

2. **Initialize Variables and send CONNECT_REQ:** After the CONNECT_AUTH_INIT is received by the client, it starts the authentication process by sending the CONNECT_REQ command to the server.

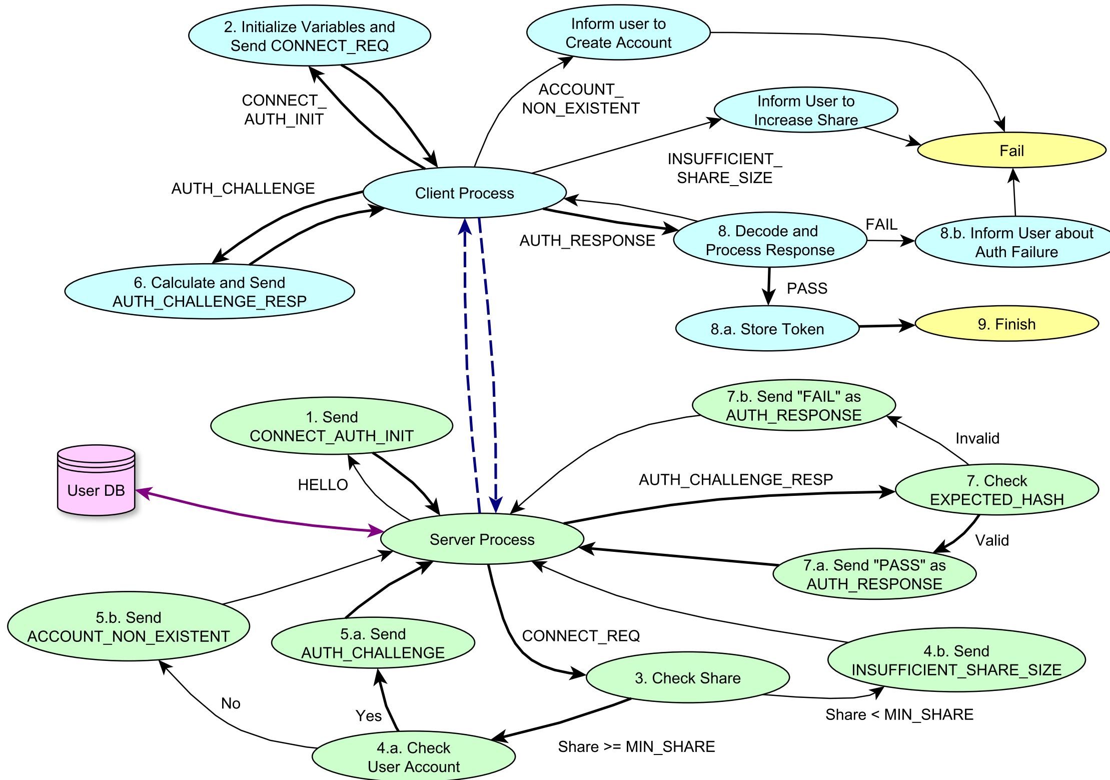

Figure 4.11: Authenticate User State

3. **Check State:** On receipt of a CONNECT_REQ command, the server checks whether whether the client's share size is greater then minimum limits.

4. (a) **Check User Account:** If the *Share Size* is greater than the MIN_SHARE set by the server the validity of the user account is checked.

   (b) **Send INSUFFICIENT_SHARE_SIZE:** If it is found that the client's share size is less than the minimum set. The client is informed by using an INSUFFICIENT_SHARE_SIZE command and disconnected.

5. (a) **Send AUTH_CHALLENGE:** If the user account information is valid. And the account is in good standing. The server generates the required session variables for the authentication process and sends a AUTH_CHALLENGE packet to the client.

   (b) **Send ACCOUNT_NON_EXISTENT:** If the account does not exist, then ACCOUNT_NON_EXISTENT message is sent to the client and the connection is terminated.

6. **Calculate and Send AUTH_CHALLENGE_RESP:** After the client receives the AUTH_CHALLENGE command, it decodes the packet and obtains the challenge in the form of a random number. The random number must be then processed and hashed as per the AUTH_HASH_GEN algorithm. After calculating the RESULT_HASH the client sends result as a AUTH_CHALLENGE_RESP packet to the server.

7. **Check RESULT_HASH:** The hash is verified to the previously generated result EXPECTED_HASH if they match "PASS" is sent else "FAIL" is sent as a AUTH_RESPONSE packet. In case the result is a match, an AUTH_TOKEN is also sent with the packet. This token is to be used for all future communications.

8. **Decode and Process Response:** The client on receipt of a AUTH_RESPONSE packet from the server decodes it and takes steps as per the packet contents:

   (a) **"PASS":** In this case the AUTH_TOKEN is stored for future use. And the authentication process finished.

   (b) **"FAIL":** In this case the user is informed and the connection is closed.

9. **Finish:** This marks the end of the authentication process.

Chapter 5

Software Design

This chapter deals with the software design aspects of the project. Due to the large size of the code base, this chapter is divided into sections which will explain the working and implementation of some of the main classes.

Both the client and server is written entirely in `C#`. The code is fully managed and uses no unmanaged code.

5.1 Description of some Major Classes

This section will show and explain some of the major classes used to maintain state and help in implementing the state machines for the implementation of the *DirectShare* protocol.

5.1.1 IncomingTCPHandler

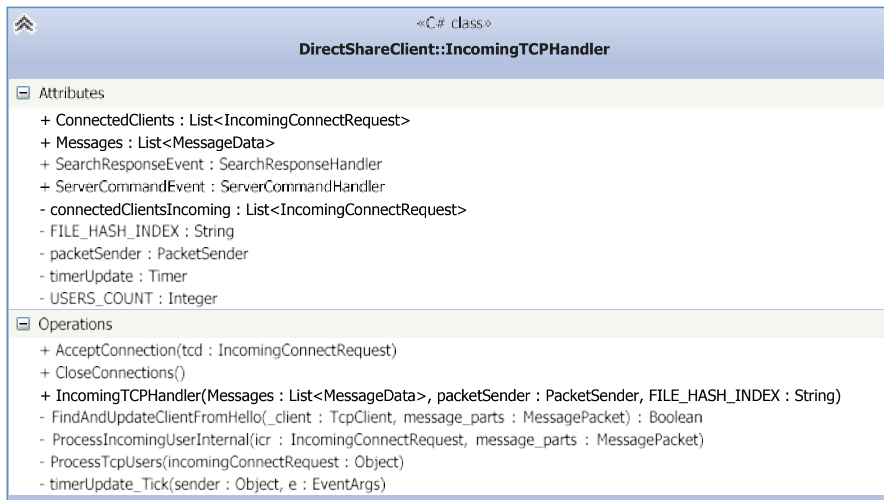

Figure 5.1: Class: IncomingTCPHandler

The *IncomingTCPHandler* class handles all the incoming connections. New threads are spawned for every new connection and individual state machines are maintained for all the connections. The state machines themselves follow the client-client protocol. A global timer event is used to process the queue of the new incoming connections and for assigning threads.

5.1.2 OutgoingTCPHandler

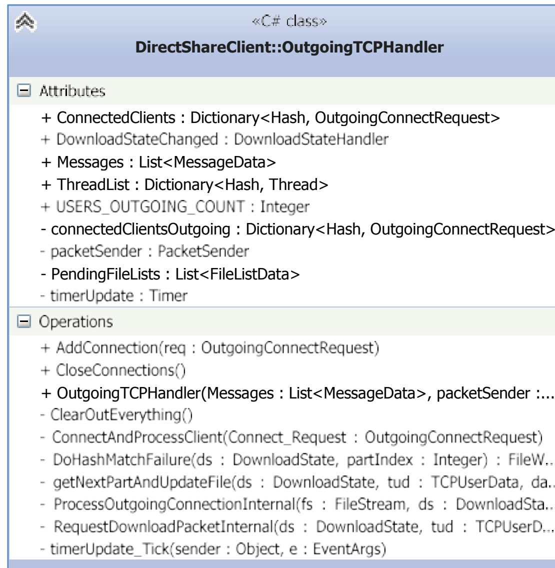

Figure 5.2: Class: OutgoingTCPHandler

The *OutgoingTCPHandler* class handles all outgoing connections. Every new outgoing connection to other clients is controlled by this class. For every connection a new thread is spawned which handles the communications. There are data structures for state management, and assignment of new threads.

A large part of download management is also implemented in this class. For downloading every chunk, a new connection has to be made, and the state machine run in the proper way. All kind of error handling, hash verification, re-downloads and other functions regarding download or any other kind of data exchange between clients is handled here.

5.1.3 UserList

UserList is a data structure which is used to maintain the state of users. It contains all the data concerning any user. It is normally sent from the server in a fixed format and decoded and initialized to be used by all the components of the software.

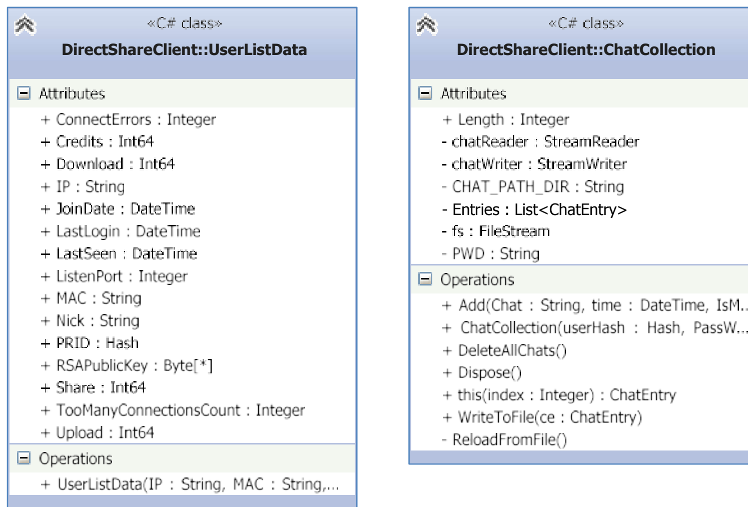

Figure 5.3: Classes: UserList and ChatCollection

5.1.4 ChatCollection

The ChatCollection class is used to to manage the chat state of users. It keeps the chat as an enumeration, at the same time it handles the encryption and storage of chats in the file system.

5.1.5 TCPUserData

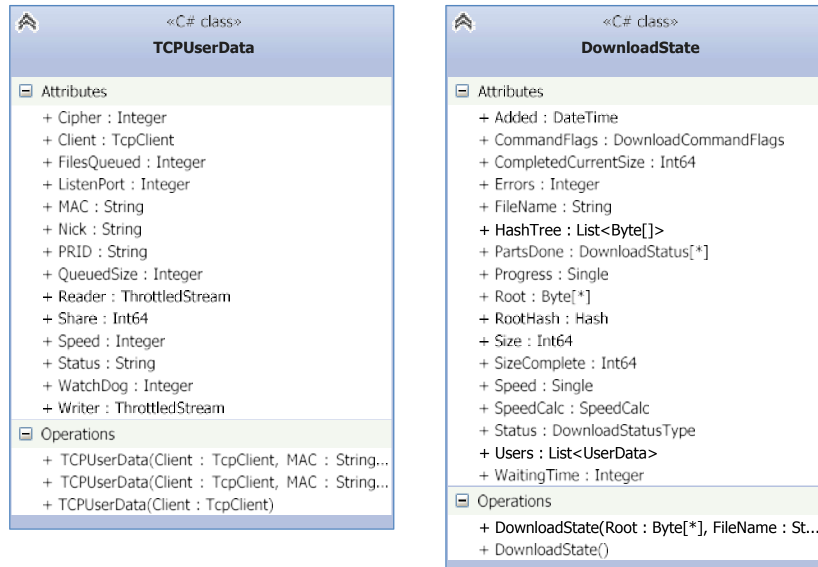

Figure 5.4: Classes: TCPUserData and DownloadState

The TCPUserData class maintains all the information regarding the connections for any user. It is kept in a HashMap for quick access and is used in parallel by multiple threads.

5.1.6 DownloadState

The DownloadState class keeps the state of all the parts and their progress for all the files queued for download. It maintains the expected hashes to be verified after part download. It is also thread safe and used by multiple threads in parallel. This class, and all the fields in this class are also serializable, and is converted to an XML format for saving the download state in case of a program exit. So that when the application starts again, the unfinished downloads can continue.

5.1.7 HashManager

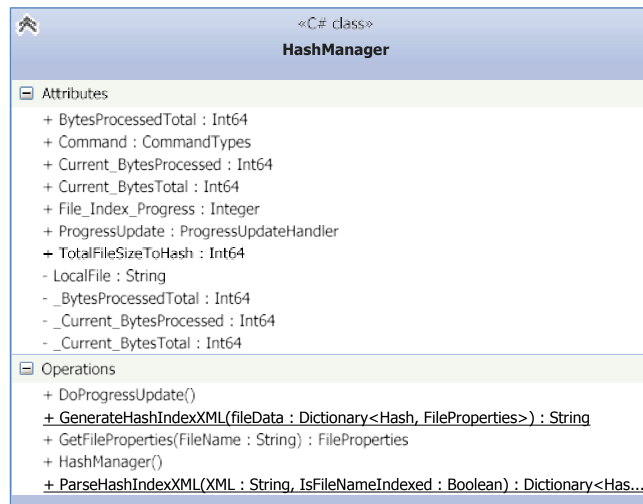

Figure 5.5: Class: HashManager

The HashManager class is responsible for performing the Hashing of all the shared files and the building of the hash trees, which are shared and used for file verification and integrity checks. This class allows for efficient hashing with pause and resume functions.

5.2 Other software design considerations

5.2.1 Dependency

By having a hierarchical design an attempt has been made to keep the dependency low. But, still some of the classes have a large number of dependencies on other classes. The figure 5.6 shows the dependency between some of the major classes for the *DirectShareClient* implementation. But, there are no cyclic dependencies.

Figure 5.6: Class Dependency Diagram

5.2.2 Scheduling of Tasks

`C#` allows for simple task scheduling using Timers. A considerable amount of timers have been used to handle the generation and maintenance of the multiple threads. The application is UI Based, as a result all the major operations have to be done in threads, because doing anything in the UI thread will the make the application non-responsive. Other than that the download process is required to be multi-threaded, because at a time multiple connections are opened which download parts of multiple files from multiple users; and the updating of the various lists of users and other data structures needs to be done in parallel.

Almost all the data structures and classes in *DirectShareClient* are thread safe.

While uploading or downloading it is made sure that the threads are allocated as per the preset preferences, if enough free slots are not available, the thread allocation is paused till some thread finishes execution.

5.2.3 Network Considerations

- Because of the connection oriented nature of the *DirectShare* protocol, TCP is used as the transport protocol.
- For all operations, TCP connections are established between either *client-server* or *client-client*.
- The server and clients have pre-determined port addresses for listening to incoming connections. Current values are:
  - Server Listen Port: 1597
  - Client Listen Port: 54800
- All the outgoing and incoming connections are buffered, the default TCP buffer size is taken to be 512 KB after some real world experiments. These buffer sizes have a major effect on the maximum possible download speeds. Incase of Wi-Fi, 512 KB is adequate; but, in order to maximize the download speeds for Gigabit Ethernet adapters, its necessary to increase the buffer sizes to 2-4 MB.

Chapter 6

Implementation and Usage

This chapter focuses on the implementation details of the *Direct Share* protocol and using the client software which was developed as a part of this thesis.

The chapter will start with the explanation of the features of *Direct Share Client* and then go into the details of actually using it.

6.1 Features

The implemented features are as follows:

6.1.1 Share and Search

- Quickly search several Terabytes of shared content; the search is distributed.
- Every folder you share; must be explicitly added to the shared list and hashed before its visible to other others.
- Very fast and efficient searching of files using Levenshtein algorithm distributed over all the clients.
- Its possible to view the other user's complete shared collection in the form of a directory browser.

6.1.2 Download

- Fast multipart download, files are downloaded from multiple users as chunks in parallel.
- Pause and resume functionality for downloads.
- Automatic download management, automatic requests to new users for file incase the original owner is missing.

6.1.3 Hashing and File Integrity

- Files are hashed in chunks before sharing; this allows perfect, error free joining of chunks to get the final file.
- Hashing is a one-time business.
- Fast hashing using efficient SHA1, tested up to 140 MB/s on Core i5
- Hashing supports pausing and resuming in case of other intensive tasks at the same time.
- During startup the shared files are verified so that anybody requesting them gets them quickly and reliably.

6.1.4 Chat

- Broadcast style chat. In this all users can see and chat with everyone else in the network.
- Highly secure Peer-to-Peer chat using RSA-2048 encrypted channels.

6.1.5 User Management

- Individual user uploads and downloads statistics
- User banning for abuse.
- Authentication to server using 2-Way SHA1/SHA2 based handshake, this is to prevent abuse and allow for user banning.
- Account creation using automated mails containing authentication data.

6.2 Usage Instructions

6.2.1 First Run

A window shows up on the first application run. This is to take the basic information about the user. Things like nickname for chat, E-Mail ID and the default download folder are needed to set up things. It is important to input the correct E-Mail ID as later this ID would be used to send mails containing authentication data.

6.2.2 Setting the IP Address

Click "File->Connect Manually" in the main menu, and enter the IP address in the provided space.

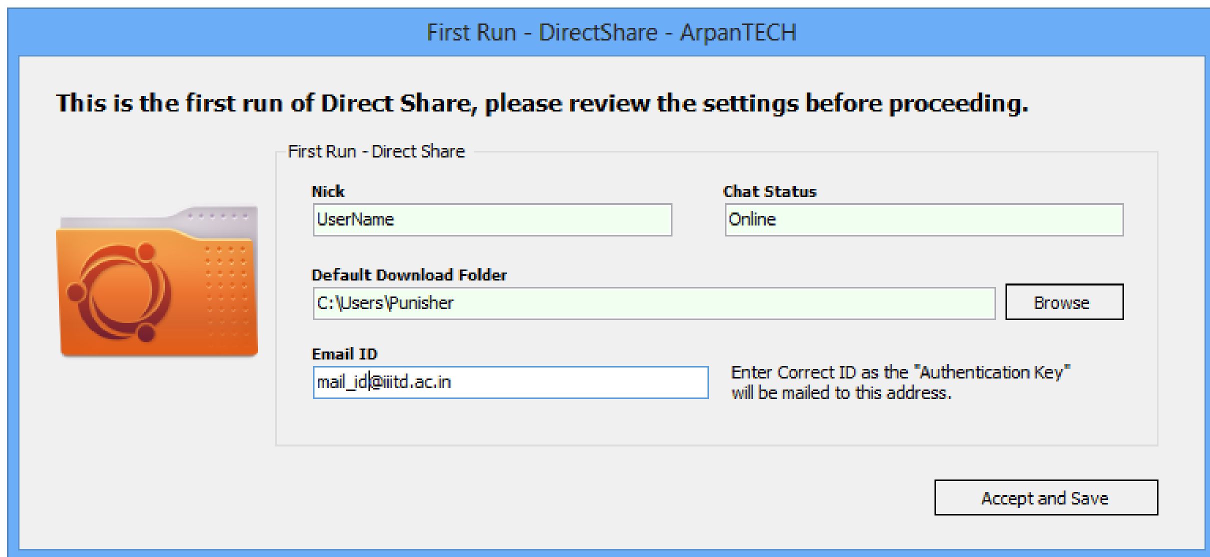

Figure 6.1: First Run

In the current version there is only a single server controlling the network. This is not optimal in case of a power failure or some other outage. There should be some method to tell the clients to join some other server. There should be also a method or command to allow the clients to search for the server by connecting the other clients and obtaining information regarding the server's address. Keeping a history of connected clients would help in this immensely.

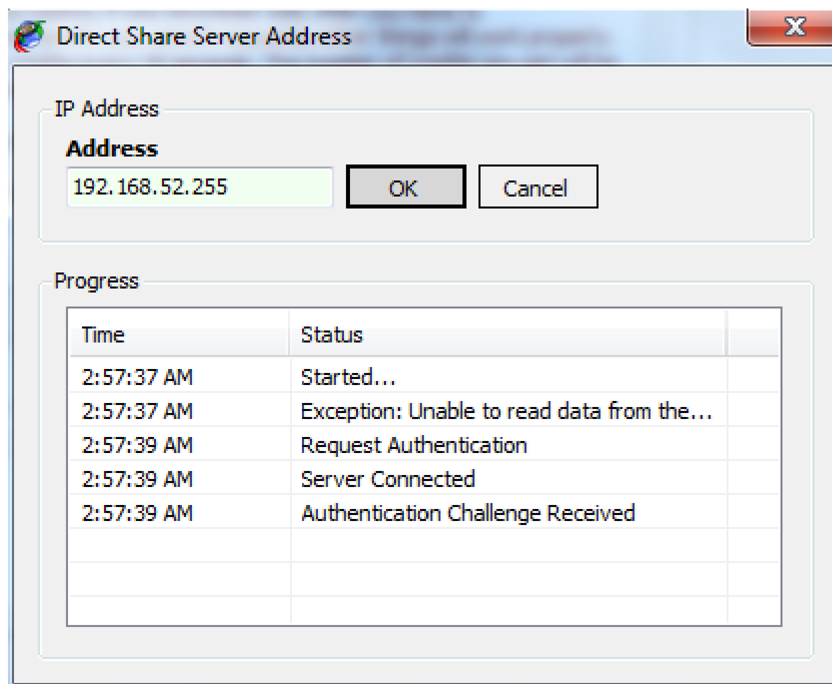

Figure 6.2: Setting the IP Address

6.2.3 Account Creation

If you are a new user, the server will inform that, "Account does not exist" then you can create a new account by clicking on "File->Create New Account" Account creation is a one time process and takes about 2 minutes.

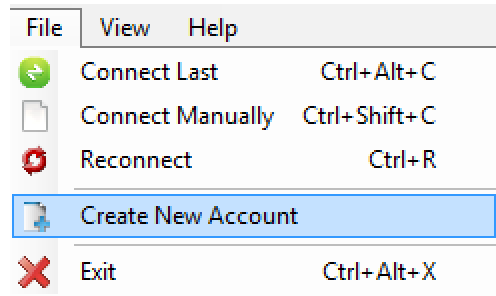

Figure 6.3: File->Create New Account Menu

1. You have to perform a robot test and click on "OK".

   Just click on the "Log Tab" to see what's going on. This is useful as everything is shown in the Log.

2. Please wait for a few seconds for the account creation mail to be sent. The mail will be sent to the address in the "File->Preferences" window; so make sure its correct.

   Normally the mail is sent in a few seconds (if there's proper internet connectivity in the server). Otherwise, please try after some time.

3. The mail will contain the Authentication data in the format:

   Copy over the entire thing to the 'Authentication Data <XML>' textbox and click on OK to save.

4. Before you connect you can select the files you want to share. The current minimum limit is a mere 40 GB. Just head over to the 'Shared Files' and select some folders you would like to share.

   After adding some folders, click on 'Refresh', this will enumerate and Hash your files using SHA1/SHA2.

   The hashing can be paused, but, if you stop it, it will start from the beginning. If you have some folders hashed before; then stopping will cause the old state to be retained. That includes adding and removing of folders.

5. Click on reconnect, to connect to the network with the new authentication data.

6. If all goes well, you will be connected to the server and you will see a green light on the 'Users' tab. And, the user list will be updated with the list of connected users.

7. That concludes the process.

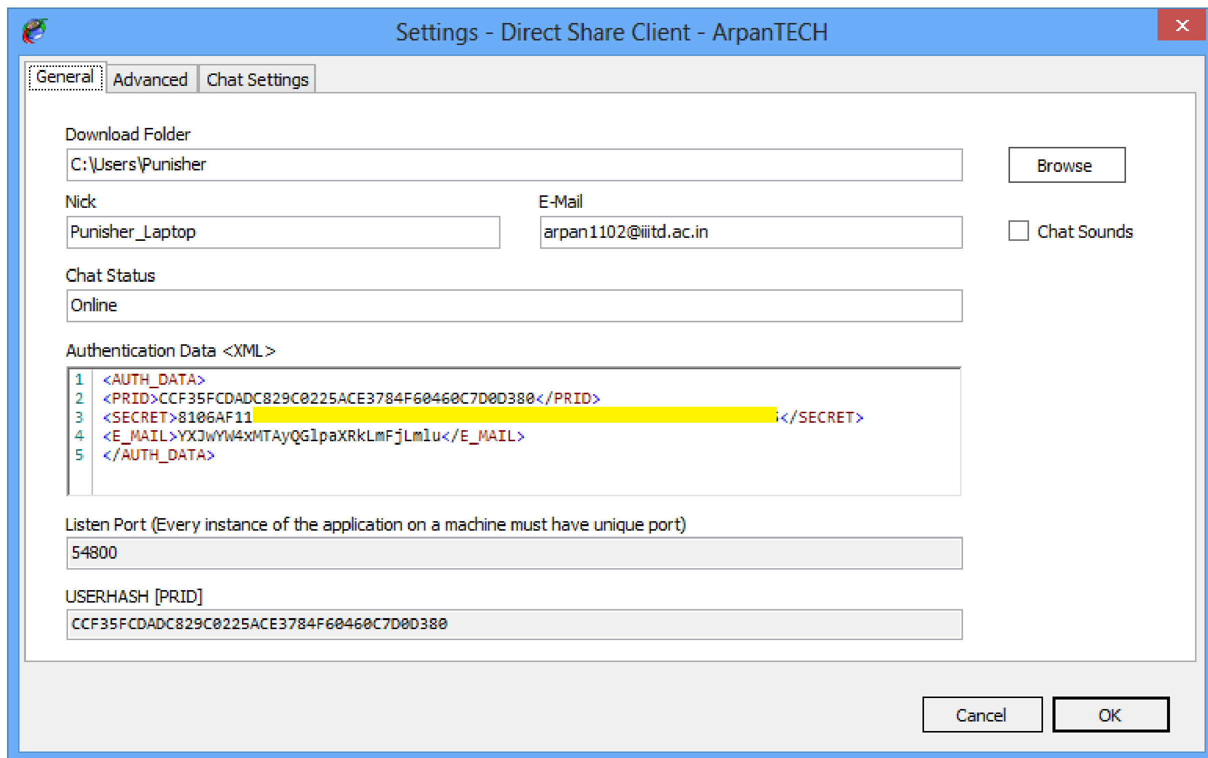

Figure 6.4: Preferences Pane

Figure 6.5: E-mail Format (Copy everything in the body of the mail.)

6.2.4 Browsing Shared Files

1. Browsing is pretty basic. In the users panel just Right Click on the user whom you want to browse and click on 'Get File List'.

2. You will see a new connection in the 'Connections' tab below showing the downloading of the list. It may take a few seconds depending on number of shared files.

   It takes around 10 MB for the file list for 40,000 files / 1.85 TB. So, just wait for a few seconds for the list to download fully.

3. After that is complete the file list will open in a new tab. Just click on the newly created tab to browse the file list.

User	Share	Credits	Last Seen	Upload	Download
Jason	1009.74 MB	0	Online	0 Bytes	7.69 GB
Punisher_Laptop	2.08 GB	183.20 K	Online	42.27 MB	8.39 GB
Punisher_PC	2.57 TB	341.95 M	Online	520.90 GB	30.68 GB
Spartacus	73.69 GB	629.25 K	Online	1.32 GB	67.63 GB
Guru	100.58 GB	1.23 M	Online	2.39 GB	89.15 GB
kanchan	0 Bytes	0	Online	0 Bytes	175.00 MB
nectar_09	0 Bytes	137.40 K	Online	2.30 GB	64.93 GB
Aarav	63.21 GB	169.74 K	Online	1.58 GB	9.83 GB
Dr.Who	72.99 GB	860.48 K	Online	10.17 GB	2.37 GB
power_boy	0 Bytes	867.96 K	Online	174.35 MB	79.09 GB
Anuskha Sharma	202.90 GB	4.49 M	Online	9.61 GB	13.33 GB

Users: 11 | Shared: 3.07 TB (Total), 285.92 GB (Average)

Figure 6.6: User List showing connected users

User	Share	Credits	Last Seen	Uplo
Punisher_Laptop	61.05 GB	716.04 K	Online	9.3
Punisher_PC	1.85 TB	40.67 M	Online	38.

Refresh
Get File List
Start Personal Chat

Figure 6.7: Get User File List

Connections | Log | Close

Incoming Outgoing

User	Status	Speed	IP	Cipher
[Myself]	Downloading File List . . .	0	192.168.48.57	N/I

Figure 6.8: Connection List

4. The file browser is similar to windows style file browsing. A tree view also shows the directory structure.

5. From the browser you can select multiple files / folders, by using the standard ctrl/shift keys. Then you can easily add files to the download queue just by right clicking and selecting 'Queue for Download'. Make sure to use the 'Download Containing Folder' option carefully.

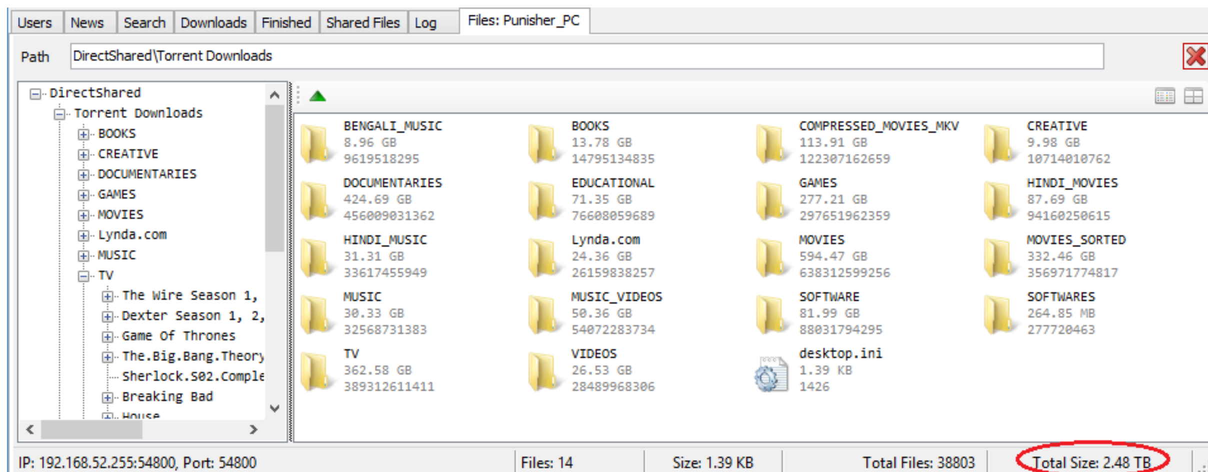

Figure 6.9: Browse User List

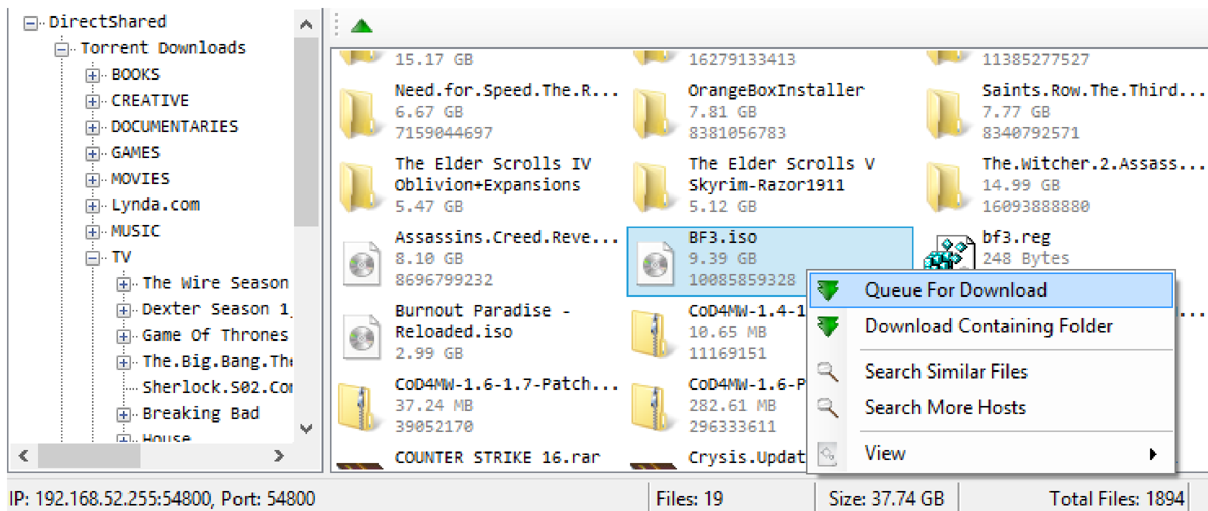

Figure 6.10: Add files to download queue.

6.3 Download Queue

This is the place where you can manage all the files being downloading. adding, removing, prioritizing, stopping, pausing and resuming; all the actions can be done here.

1. The 'Download' Tab lists all the files currently in the queue.
2. All the files are downloaded in parts from the different users. If the download is added using the file browser then there will be only single user to whom the file originally belongs.
3. All the files will be downloaded sequentially. You can change the order using the provided Up/Down arrows.
4. After a file is fully downloaded and verified, it will be moved to the *finished queue*.
5. In case the file is added as a result of search then all the users will be added. In some

cases when the user is not able to share file, due to file system corruption, I/O error, or file unavailability, the user may send a NACK which will temporarily remove the user from the list, if all the users are removed then the list may show 'NoUsers' then you can perform a search by right clicking to find other possible users having the same exact file. Sometimes the original user may again show up. General scenario is that the user has removed any external drives from which the file was being shared. Or, the user is doing something very I/O intensive and hence cannot handle requests. Or, the user has stopped sharing the file by removing it from the shared folder.

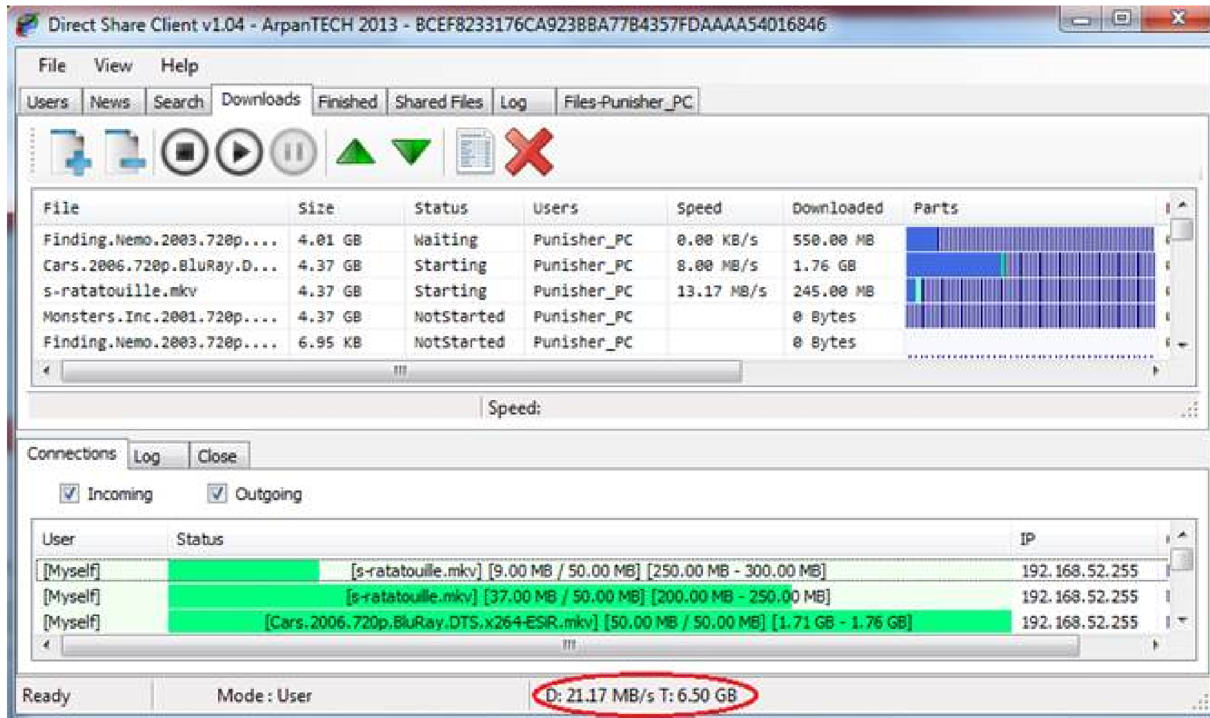

Figure 6.11: Download Queue

6.4 Finished Queue

After the download is complete all the files are move to the finished queue. The finished Queue is a temporary queue. The contents will be removed if the application is closed. Double-Clicking any file in the queue opens the file using the default application for the file. Its equivalent to Double Clicking any file in the Windows Explorer. In the ToolStrip there is an option to open the containing folder also. The folder where the file is saved.

Removing the files from the queue does not change the downloaded files in any way.

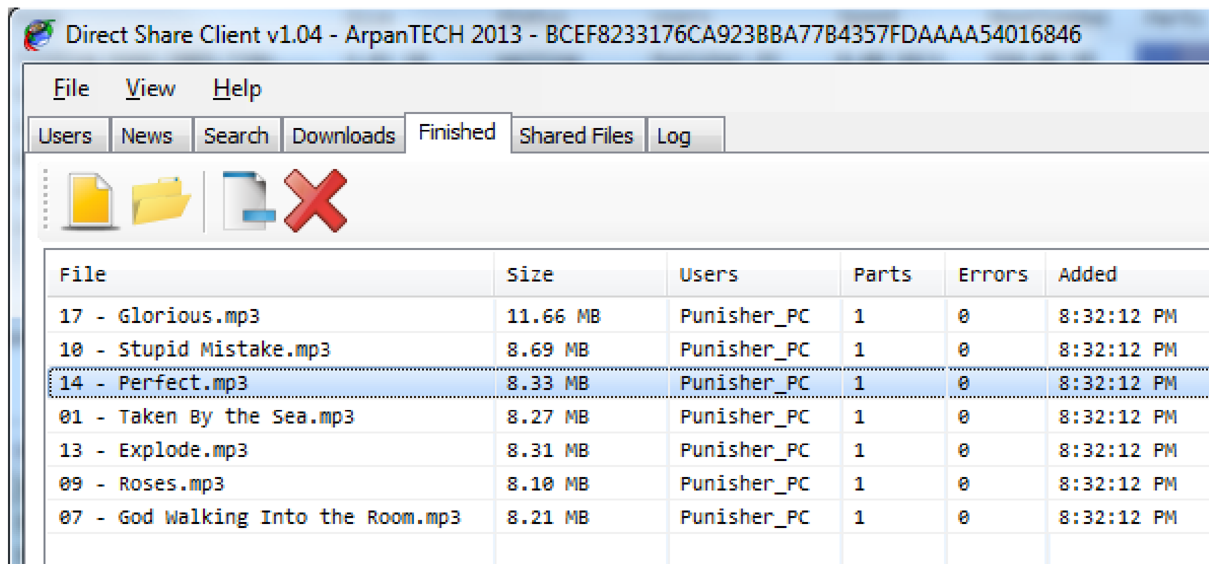

Figure 6.12: Finished Queue

6.5 File Searching

Fast Distributed Searching is one of the most powerful features of Direct Share. Searching enables the user to find files from the shared lists of all the users. This allows you to search several Terabytes of data within a second.

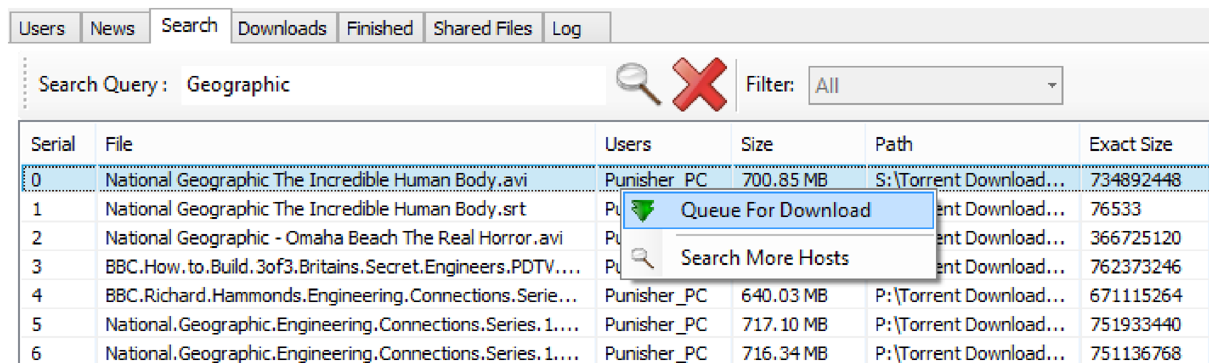

Figure 6.13: File Searching

1. Performing search is done by entering the search query in the TextBox and pressing Enter/Search Button.

2. The files can be sorted according to the FileNames or FileSize by clicking the appropriate column headers.

3. The searching is a distributed process whenever you type in a query, the server broadcasts this query and the way to communicate to the sender to all the users. The users then search the keywords in their shared file list, if some entries are found, the users reply to the original sender directly with the search response.

4. After getting the responses, the client builds a list of all the responses received, and their associated senders. Which is dynamically updated using a 'Virtual List View' and displayed.
5. Just select the file/files you want and Right-Click and click in 'Queue for Download' to add the file to the Download Queue for Downloading.

Tips: Use accurate names, smaller queries will fetch more results. Search is case insensitive.

6.6 Sharing Folders

1. Head over to the 'Shared Files' tab to add folders to be shared.
2. Click on the Add button to select a folder to be shared.
3. After the folder is selected A virtual share name is assigned, the default name is 'Direct-Shared'. You can change the Share Name as per your wish. This is done by selecting the folder and clicking the 'Edit Share' button The Share Names is added so that files from multiple drives can be grouped together. This allows for easier file browsing for users. The drive names and paths are also don't show up this way.
4. After you are done adding the files. You must press the Refresh button to apply the changes.

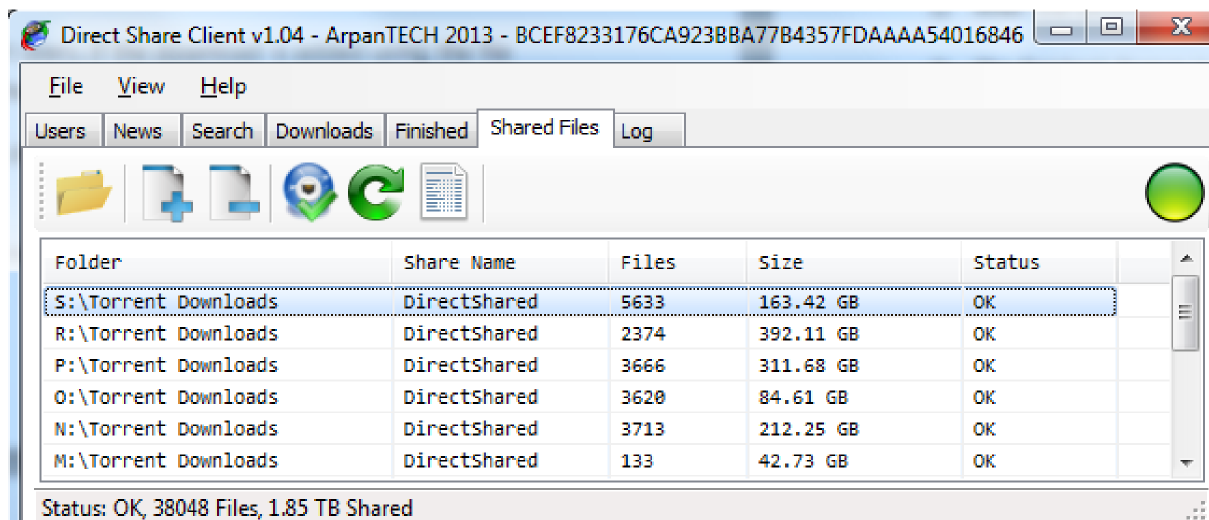

Figure 6.14: Shared Files

5. All the files in your shared folders will firstly be enumerated and then hashed. You can see the hashing progress and speed. Hashing can be paused/stopped. If stopped, the newer changes will not be reflected in the file list. Hashing must complete fully for the hash index to update.

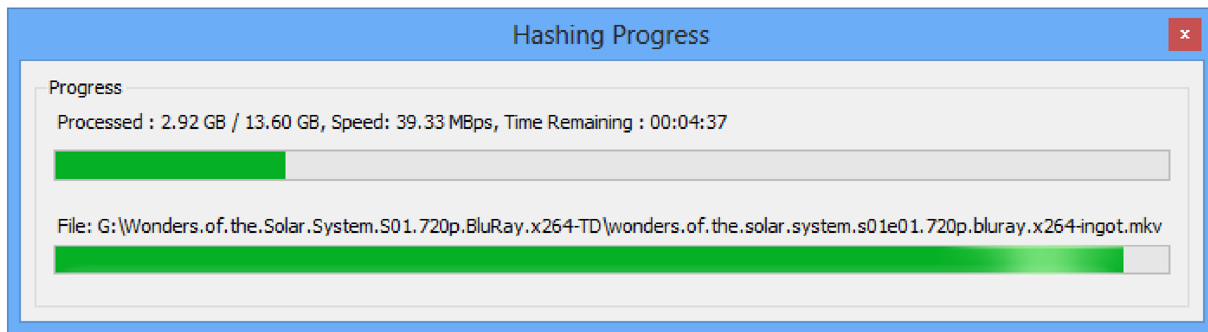

Figure 6.15: Hashing Progress

6. Press the 'Progress' button or View->Indexing Progress to view the current progress.

7. After the hashing is complete, all the files will be available to be searched and obtained through file list.

6.7 Secure Personal Chat

This allows secure chat between two users.

1. Start a chat session by Right Clicking on any user in the 'User List' and clicking 'Personal Chat'

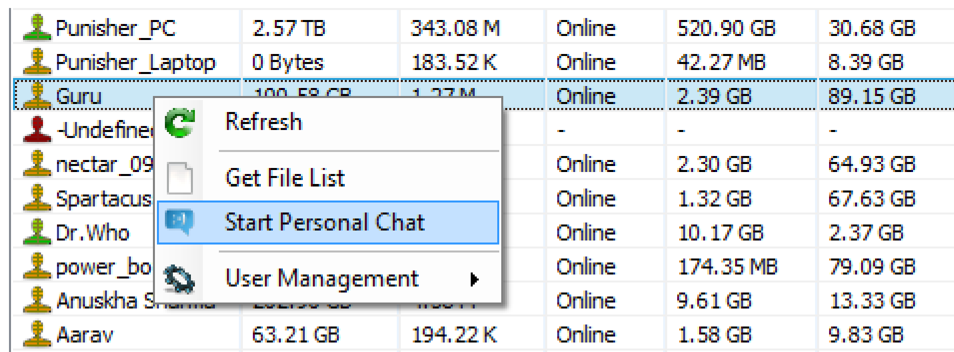

Figure 6.16: Start Personal Chat

2. A new tab will open and you can start chatting.

The chat is very secure. The security aspects are explained below.

6.7.1 Chat Features

Encrypted Personal Chatting: All chats are signed by the sender with a signature and verified at the receiver. Status messages like: Seen, Typing, Waiting etc... Optional sharing of ID's (To confirm the identity of the users). All previous chats are saved in the client machines, in

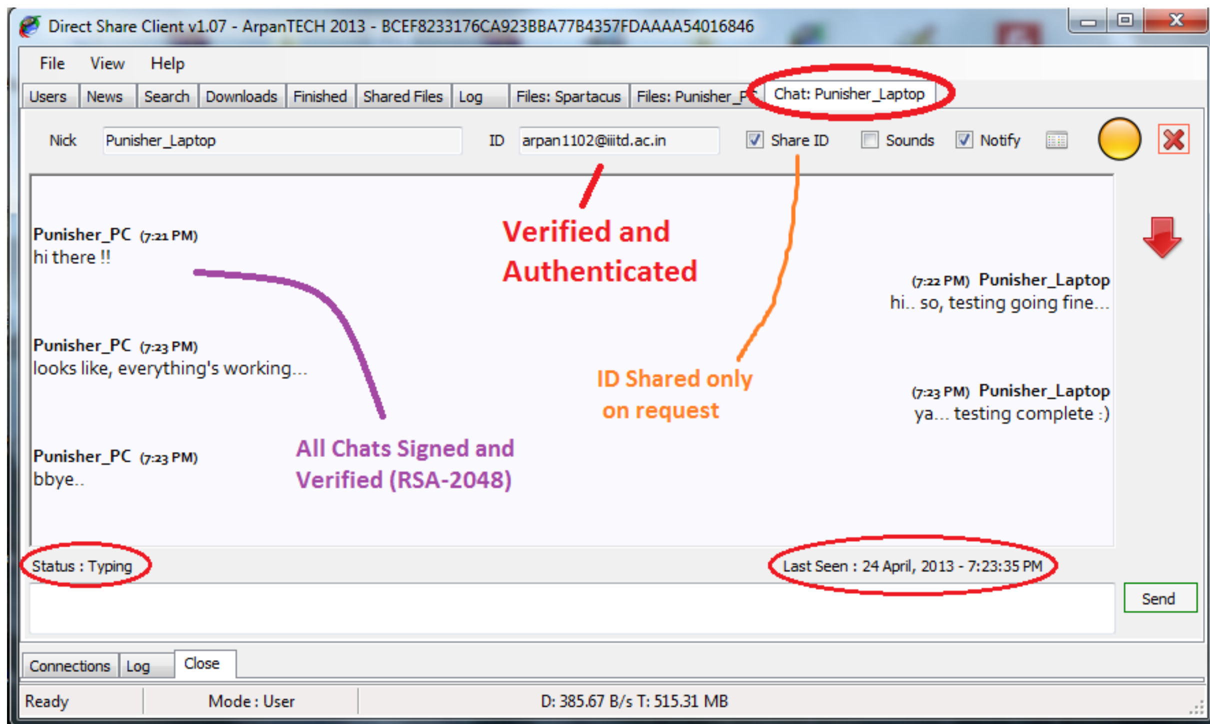

Figure 6.17: Secure Chat

encrypted form (AES-256). As the server just forwards the encrypted packets, there is no way for the server or anyone to decrypt any chat. The private key required to decrypt, rests only with the user receiving the chat and is generated at runtime at application start-up.

Listen Port	Join Date	PRID	RSA Public Key
54800	Thursday, April 18, 2013	66242DD7E10B6F61109E7A000E7ED3...	F6251957EB727E4BD085ACA3833B...
54800	Thursday, April 18, 2013	54A33E7FFD6215864D9FEFF09D4432...	90A9384C57B141A821CE4731CC95...
54800	Wednesday, February 2...	BCEF8233176CA923BBA77B4357FDA...	E7D16FCAE544465CD975CC40CE3...
54800	Tuesday, March 12, 2013	CCF35FCDADC829C0225ACE3784F60...	B000356143DE5BB6467575107A48...
54800	Tuesday, March 26, 2013	C709B97D8BAA4C072D3B337394A95...	CB52B1300473F6D59BB355A80EE5...
54800	Wednesday, March 20, ...	02DD0F0A009F9B27557D666B4F0C9E...	992A949BACF02AEE2EC85F38EEBD...
54800	Friday, April 12, 2013	1F0903A7D06CDCFB2FB277EF0F7F69...	B86ABB3602077A5082E7A33FEB4C...
54800	Sunday, April 14, 2013	2A00909AAF0B6C4C7956FB273006C0...	A04A1A6F30DC30B0FC5AC23026D...
54800	Saturday, April 13, 2013	108675FC7823D0EBC0A62FBFD1E17B...	B59ADD6EE9F1CAC76CB36A2DCF2...
54800	Sunday, April 14, 2013	741F9968D90006B4581EBD3BF44262...	B61C2BE0225FA17ECF526505DEF5...
54800	Tuesday, April 16, 2013	746CB2140FACCB01EB1EABD4BF82D...	9769146A1AA880FA833695A069E1...

Users: 12 | Shared: 3.07 TB (Total), 261.92 GB (Average)

Figure 6.18: RSA Public Keys of the Users

The security is based on the well known RSA algorithm.

Every individual client broadcasts their RSA public key to the server, and the server broadcasts it to all the users. As a result, everyone has the RSA2048 public key of everyone else. Now whenever any chat is transmitted, it is encapsulated in a packed and encrypted with the RSA

public key. As a result only the receiver can decrypt, and NO one else.

Moreover because of the E-Mail based authentication before, the authenticity of the user can also be guaranteed by the server. Both parties can share there ID's by selecting a CheckBox in the 'Chat Window', doing that, securely tells the server to forward the E-Mail ID to the recipient of the chat. This prevents any type of repudiation attacks.

6.7.2 Blocking Users

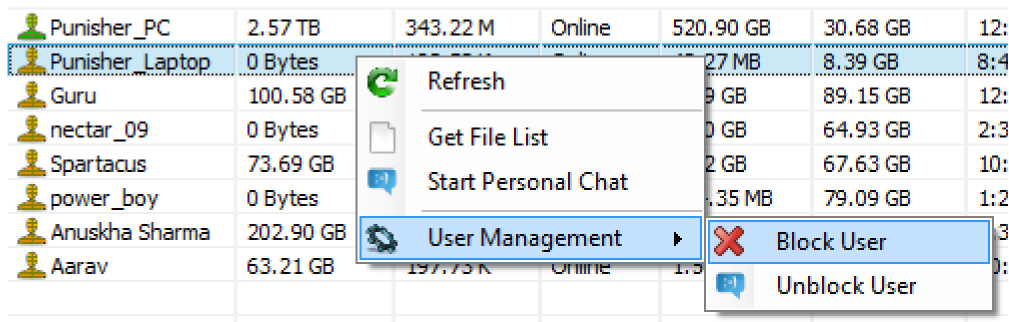

Figure 6.19: Blocking Users

Blocking of users is also possible. Its as simple as it can be. After a user is blocked, all chat attempts are ignored by the client.

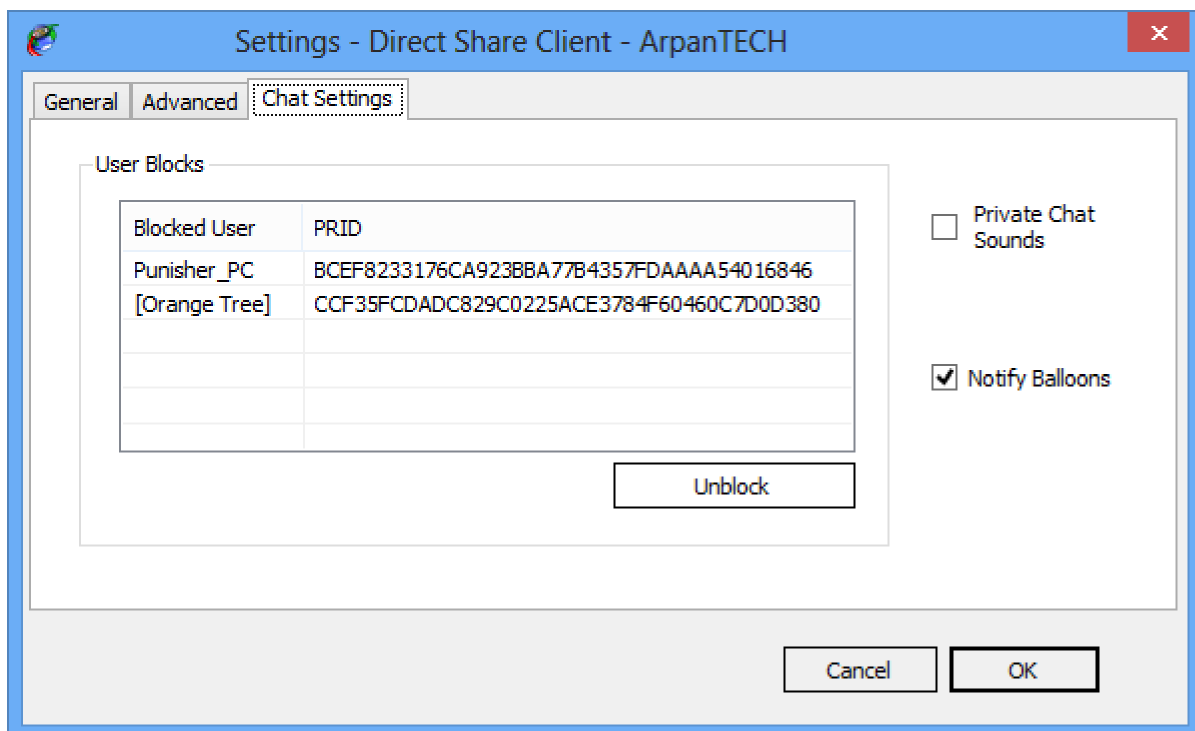

Figure 6.20: Blocking Users

In the 'View->Preferences->Chat Settings' window it is possible to view all the blocked users. Manually unblocking can also be done by selecting the user and clicking on the 'Unblock' button.

In the current implementation, the users are only blocked from sending personal chats. The blocked user can see and download the shared files. There may be an option for this in the later versions.

Chapter 7

Results

7.1 Statistics (Alpha Testing)

- **Number of Registered users / testers:** 25, Regularly active (13-14)
- **Total Files Shared:** 3452 GB
- **Total Data transferred:** 850 GB over 3 weeks
- **Total files in network:** 100,000 +
- **Download speeds:**
  - **Wi-Fi:** 1.5 - 2.5 MB/s
  - **LAN:** 25 - 55 MB/s (Depending on settings)
- **Search speeds:** 0.75 sec / 50,000 Files / User
- **File corruptions:** No instances during testing.

7.2 Download Speeds

The download speed was analyzed for the following two network configurations. The analysis shows that selecting proper TCP buffer sizes and number of simultaneous TCP connections is very critical for optimal network utilization. The plots below were obtained by measuring the download speed for various configurations by logging the current speed every second.

7.2.1 Wi-Fi

The Wi-Fi speed measurements were done by using a wireless router, a switch, a laptop and a desktop. The Switch-Desktop and Switch-Router is connected using Gigabit Ethernet and the

Router-Laptop was working in the 802.11g mode (15-20 meters from the router). As a result, the only speed bottleneck is the wireless connection.

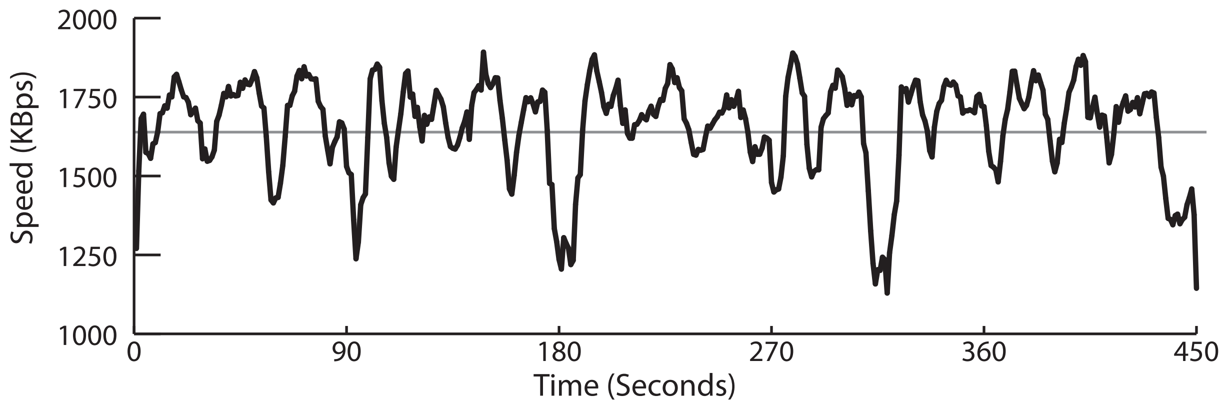

Figure 7.1: Wi-Fi (802.11g) Network: High Utilization, 2 Connections, 256 KB Buffer 1655 KBps

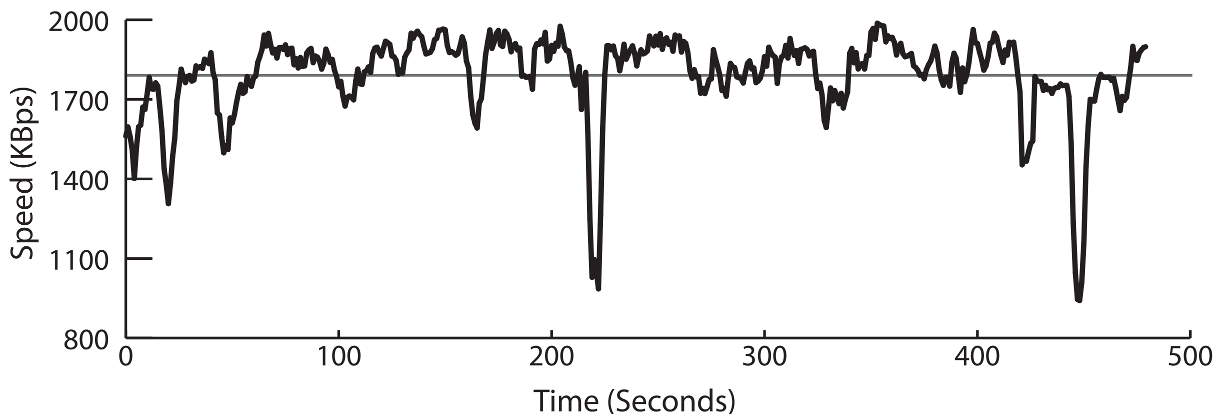

Figure 7.2: Wi-Fi (802.11g) Network: Low Utilization, 2 Connections, 256 KB Buffer, 1792 KBps

By seeing the two plots for the Wi-Fi speeds its clear that in a low utilization environment, the download speeds are much more stable and consistent. There are some periods where the speeds go down while new blocks are being allocated. In the current implementation, the connection allocation speed is slowed down to prevent very sudden increases in RAM usage.

After a lot of testing and analysis, the buffer size of 256 KB was chosen for the Wi-Fi connection profile.

Increasing the buffer size may cause connection drop by timeout. Because the timeout is set to 10 seconds and if the data equal to the buffer size is not transferred in the timeout period, the connection is closed; this is done to increase reliability and faster response times in case of disconnections and errors.

7.2.2 Gigabit LAN

The testing of *DirectShare* over LAN was done by using two PC's with Gigabit Ethernet network adapters. The two PC's were physically separated by four gigabit network switches, including a fiber optic link. The total distance between the two PC's was around 1/2 kilometer. This replicates the network infrastructure in a large campus. Windows 7 was used as the OS and the code was compiled for x86 architecture.

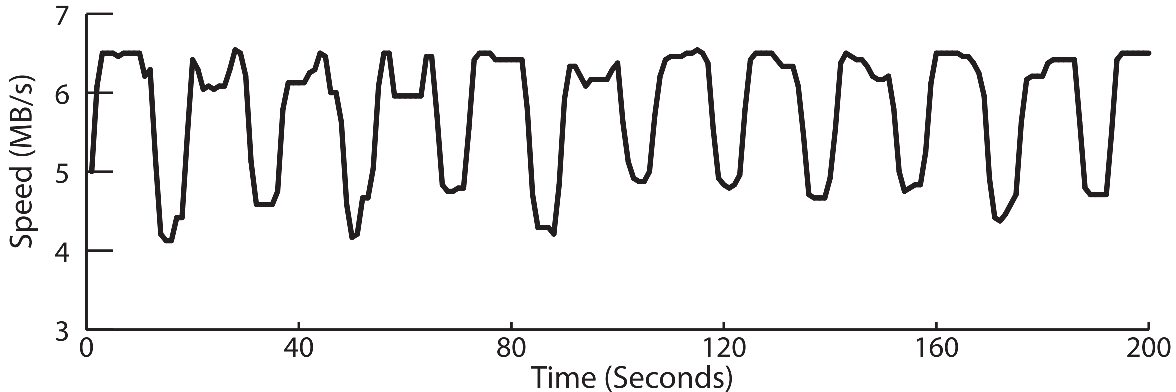

Figure 7.3: Gigabit LAN, 2 Connections, 256 KB Buffer

The figure 7.3 shows the result of operating the protocol in a Gigabit LAN network, but using small 256 KB buffer sizes. The small buffers massively reduce the maximum speed available.

The figure 7.4 shows the download speeds for three different values of *maximum allowable connections*, and the fixed *buffer size* of 2 MB. It is interesting to note that with 2 connections the speed is less but stable; but with 6 connections the speed is much higher, but quite unstable. The basic reason for such a behavior is that the CHUNK_SIZE is only 50 MB, and that much data is transferred in a few seconds, and then the thread has to wait for re-allotment. Instant re-allotment and connection re-use will tremendously help to increase the throughput. Another factor is that; as the connections are not re-used, a large amount of RAM has to be allocated and de-allocated very quickly, and sometimes the thread has to wait until enough free memory is available.

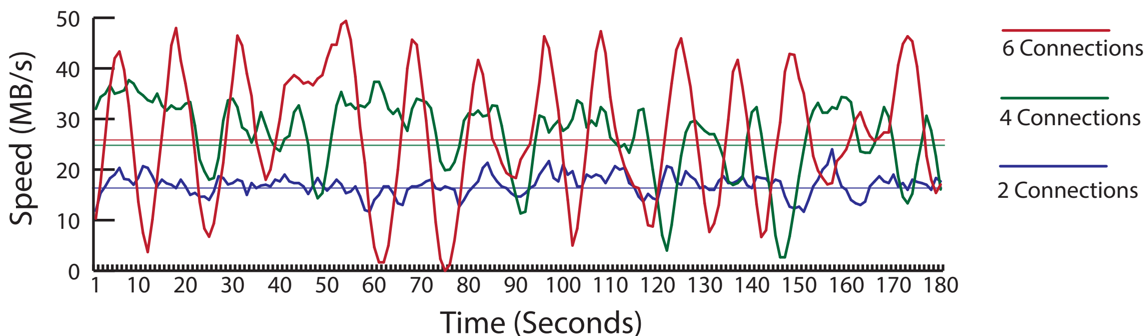

Figure 7.4: Gigabit LAN, 2 MB Buffer

The figure 7.5 and 7.6 shows download speeds when a large number of parallel connections are used. The following cases are considered:

- 2 MB TCP Buffer, 24 Connections
- 1 MB TCP Buffer and 16 Connections
- 512 KB TCP Buffer and 16 Connections

The figures show that, after a fixed number of connections, any more increase in connections don't increase the speed. Larger buffer sizes leads to higher download speeds, but, consistency in speed decreases, primarily due to timed-out threads and waiting threads. After a lot of analysis it was found out that for the current implementation and usage over Gigabit LAN, the optimal number of connections lies between 8 to 16, and the optimal buffer size is 1 MB.

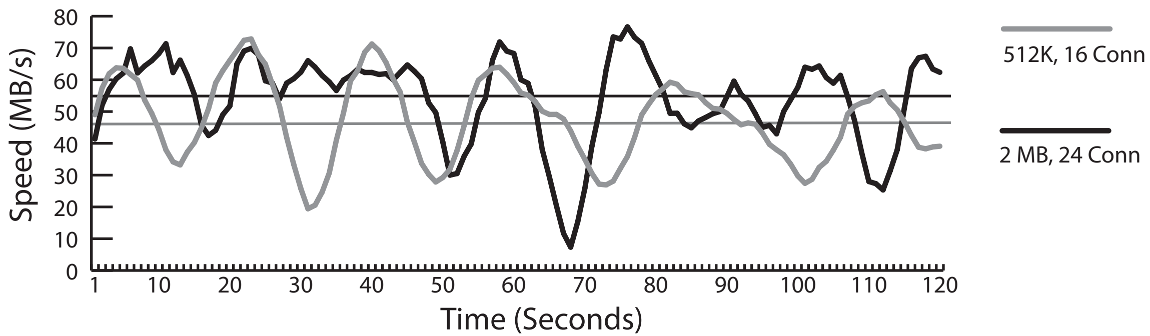

Figure 7.5: Gigabit LAN, 16/24 Connections, 512KB/2MB Buffer

In the figure 7.6, the connections plot shows active connections.

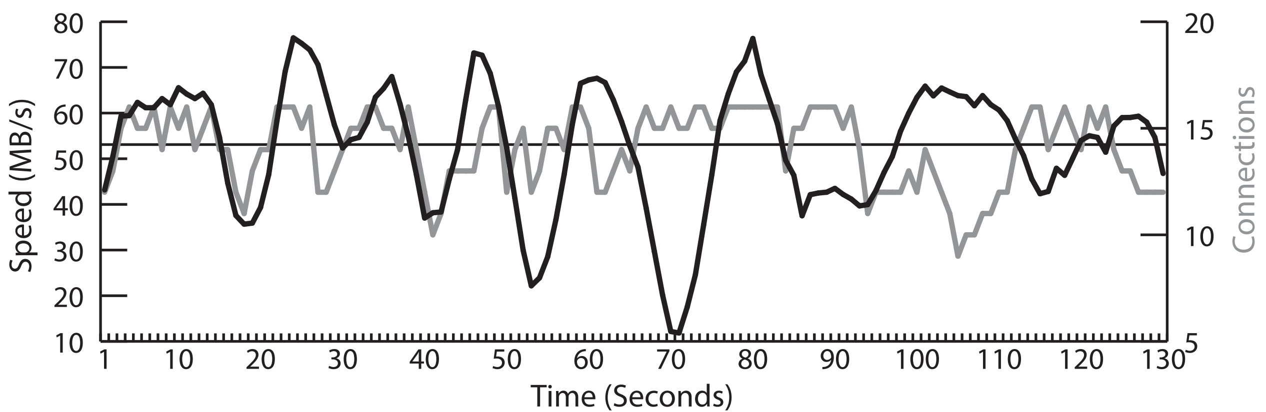

Figure 7.6: Gigabit LAN, 16 Connections, 1 MB Buffer, 52.88 MBps

7.3 Comparison with other protocols

The following table shows the features of three different protocols, the comparison is done with the original *BitTorrent* and *NMDC* (NeoModus Direct Connect) specifications.

7.3.1 Protocol Feature Comparison

Feature	DirectShare	BitTorrent	NMDC
Challenge Based Authentication	Yes	No	No
User Identity	Yes	No	No
User bans	Yes	No	Yes
User bans (Identity)	Yes	No	No
Limiting shared files	Yes	No	No
Parallel multi-part download	Yes	Yes	Yes
Hashing for error free file downloads	Yes	Yes	Yes
Need of index files	No	Yes[1]	No
On-the-fly compression	Yes	No	Yes[2]
Public Chat	Yes	No	Yes
Private Chat (with identity verification)	Yes	No	No
Credit system and usage metering	Yes	Yes[3]	No
File searching	Yes	No	Yes
Automatic account creation	Yes	N/A	No

7.3.2 Download Speed Comparison

The following two tables describe how the different protocols fare in comparison to *DirectShare*.

Protocol (10 Mbit LAN)	Average Speed (KBps)
BitTorrent	951
DirectShare	998
DC++	711

Table 7.1: 10 Mbit LAN

Protocol (1 Gbit LAN)	Average Speed(MBps)	Continuous Speed (MBps)
BitTorrent	28.56	28.56
DirectShare	25.40	31.18
DC++	31.05	31.05

Table 7.2: 1 Gbit LAN

[1] .torrent files / DHT is needed
[2] Using protocol extension
[3] Private trackers can perform logging

7.3.2.1 Test Setup

The transfer speeds were analyzed by transferring a fixed set of files, totaling 4.85 GiB and then calculating the total transfer time.

Two different setups were used for the comparison. The test is aimed at measuring the bandwidth of different protocols for standard implementations and use-cases, for point to point large file transfers. All the tests were performed twice in both directions for consistency.

- **10 Mbit LAN:** A 10 Mbps USB-LAN hub was used to connect the devices.
- **1 GBit LAN:** A 1 Gbps LAN switch was used to connect the devices.

For the protocols the following application setup was used.

- **DirectShare:** DirectShare server application was run on one of the PC's and both the PC's had respective clients.
- **NMDC:** Ptokax 0.4.2.0 was used for NMDC Hub and DC++ version 0.825 was used as NMDC client.
- **BitTorrent:** uTorrent 3.2.3 was used with *bt.enable_tracker* option to enable the tracker. Torrents were created and seeded on one PC while it was downloaded by the other PC. uTP bandwidth management was disabled.

7.3.2.2 Discussion

The results of the file transfer speed test are as expected. As all the protocols use TCP for transport, there is minimal difference between the multiple protocols. Most of them can be attributed to factors like default buffer size settings and number of simultaneous connections.

The value of *Average Speed* for *DirectShare* in table 7.2 is a bit slower than the others because of the implementation. New thread/connection allocation is currently costly, because re-use of connections, which is done by the other clients is not yet implemented. As a result, the small delays (waiting for allocation) in download process leads to a slightly lower speed. This effect will be much lower if more connections are used, and will be non-existent if the connections are re-used.

The results should be similar for downloads with a large number of peers, because, the speed depends on the transport protocol, which is TCP almost universally.

Chapter 8

Conclusion

After a lot of testing and usage of *DirectShare* by around 30-40 users over several months, and development backed by feedback from users and usage trend, it is clear that the protocol is suitable for real world use. The protocol and the implemented application allows for easy sharing of files quickly over the network; it is specially suited for effective file sharing in large campuses. The download speeds and network utilization is high, and at par with similar protocols like DC and ADC, while providing extra security at multiple levels and authenticity of users.

Many additional features like secure chat, distributed search, incentives, authentication, automatic account creation, sharing of lists etc., are provided at the base level of the protocol.

Even though a large number of standard features are already described, the protocol can be easily extended by simply adding more *commands* and *responses*.

An updating system was also developed after some small scale use, it allows the server to initiate updates to the clients whenever there's a change in the protocol version or there's an improvement in the client application. This allows for all the users to have the latest version of the application, or whichever version the server admins choose to run. The update system is also a part of the base protocol.

There are still many more interesting directions for future work and scope for further improvement.

Chapter 9

Future Work

9.1 Connection Reuse

Even though the download speed is high enough over standard network interfaces, but, by re-using connections between connected peers its possible it improve network efficiency, this is an application feature.

9.2 Automatic Server Address Resolution

The protocol may be extended to allow the clients to find the possible server addresses by communicating with other clients in their neighborhood.

9.3 Pure P2P operation in the absence of a server

In case of a server failure, it should be possible for the clients to communicate effectively among themselves and make a listing of all their neighbors using a Gnutella like protocol, this will make the network very resilient.

9.4 Direct Streaming of Video and Audio

Directly streaming digital content will allow the users to either preview or even fully view the content, without even downloading. This can help users with smaller hard drives to access content, but, this feature will promote free-riding, so proper incentives should be given to content providers.

9.5 Implementing Dynamic Chunk sizes

The CHUNK_SIZE in the current implementation is 50 MB, which is good for medium and low speed access, but, for much higher speed connections like Gigabit Ethernet / Fiber optic, larger chunk sizes are required for optimal network usage.

9.6 NAT Support

Proper NAT support will simplify the setup and use of the client to work behind routers and firewalls. This is very important feature for users behind a router.

9.7 Bandwidth Management

The current bandwidth management features in the client application is based on passive throttling of the streams. A bandwidth management system should be devised which would allow the user to have good internet browsing experience by throttling the downloads actively.

9.8 Supported Platforms

Currently the implementation only supports windows, but, the code is written in fully managed language and by using technologies like MONO, a platform independent application can be developed.

Bibliography

[1] Bellare, M., and Rogaway, P. Optimal asymmetric encryption. In *Advances in Cryptology-EUROCRYPT'94* (1995), Springer, pp. 92–111.

[2] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. Extensible markup language (xml). *World Wide Web Journal 2*, 4 (1997), 27–66.

[3] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies* (2001), Springer, pp. 46–66.

[4] Cohen, B. Bittorrent protocol. *http://www.bittorrent.org/beps/bep_0003.html* (2001).

[5] Eastlake, D., and Jones, P. Us secure hash algorithm 1 (sha1), 2001.

[6] FIPS, N. 180-2: Secure hash standard (shs). Tech. rep., Technical report, National Institute of Standards and Technology (NIST), 2001. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf, 2001.

[7] Hess, J. Direct connect protocol. *http://www.plop.nl/ptokaxbots/Mutor/DC-Protocol.htm* (1999).

[8] Josefsson, S. The base16, base32, and base64 data encodings.

[9] Justin Frankel, Gianluca Rubinacci, T. P. The gnutella protocol specification. *http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html* (2000).

[10] Kulbak, Y., Bickson, D., et al. The emule protocol specification. *eMule project, http://sourceforge. net* (2005).

[11] Levenshtein, V. I. Binary codes capable of correcting deletions, insertions. Tech. rep., and reversals. Technical Report 8, 1966.

[12] Reinhold, L. Quicklz, 2009.

[13] Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.

[14] Sieka, J. Advanced direct connect protocol. *http://adc.sourceforge.net/ADC.html* (2007).

[15] Stevens, W. R. Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms.

[16] XXXXXX. edonkey protocol. *http://www.jmule.org/files/eDonkey-protocol-0.6.2.html* (2000).