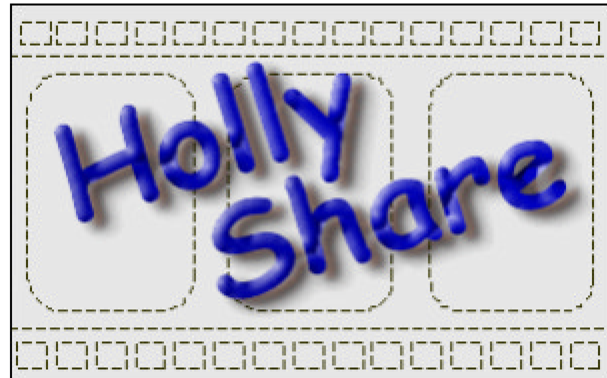

HollyShare: Peer-to-Peer File Sharing Application

ICS 243A Class Project

Songmei Han
Bijit Hore
Ilya Issenin
Sean McCarthy
Shannon Tauro

INTRODUCTION....3

SURVEY....5

Architecture of Existing Peer-to-Peer File Sharing Systems 5
The Centralized Model of P2P File-Sharing 5
The Decentralized Model of P2P File-Sharing 6
Comparison of Centralized and Decentralized P2P Systems 7

File Discovery in Decentralized P2P File Sharing Systems 12
File Discovery Mechanisms in Freenet: Chain Mode 12
File Discovery in Gnutella network: Broadcast 14
Search Performance Comparison of Gnutella and Freenet 16
Problems with Existing P2P File Sharing Systems 17

Lessons Learned from Surveying the Existing P2P File Sharing System....17

IMPLEMENTATION....19

System Architecture....19

System Overview & Graphical User Interface (GUI) 20

Node Database....21

File Database....22

Configuration Manager....24

File Transfer Module....24
Client component 25
Server component 25
File Transfer Module Summary & Future Work: 26

Network Connectivity Manager....26
Network topology 26
Connecting to the network 26
Reconnection 28
Information Discovery Mechanism 30
HollyShare Network Protocol 30

Project Development....33
Requirements Phase 33
Architecture Development 34
Distribution of Modules 34
Development & Integration of Modules 34
Division of Labor 34

Analysis of System....35

CONCLUSIONS 38

REFERENCES 39

Introduction

Peer-to-peer file sharing applications have been in existence for some time. Familiar applications or architectures such as Napster, Gnutella, and Freenet have a large user base. The main design idea of these systems is files are distributed throughout nodes. This is much different from traditional client/server systems, where files would lie on one central machine, and all transfers would occur only between the individual clients and that machine. In peer-to-peer, file transfers can occur between the individual nodes.

Hollyshare is another peer-to-peer file sharing application. Why did we decide to create one more file sharing system? The answer is that HollyShare have some unique properties, which distinguish it from other existing systems.

First, HollyShare is designed for use by a group of people that know each other as opposed to existing file-sharing systems, where anybody can enter the system.

Second, it is designed to be a catalogue system, as opposed to a query system like many of the more familiar peer-to-peer file sharing applications and architectures. The catalogue exists on each of the nodes in the system, so searching for files is not necessary. This is beneficial for relatively small number of shared files, when the users are not sure what do they want to download and prefer to select something from existing list, rather than try to search for something that is most probably is not in the system.

Third, our system is fully decentralized – here are no dedicated servers.

Fourth, HollyShare was designed with privacy considerations in mind – nobody outside the group is supposed to get any information about the files shared or obtain transferred file contents using a network sniffer.

Those considerations lead us to make the following assumptions, which affected our design decisions. The implications for each module will be discussed later. For this project we have assumed the following:

1. The number of users will be limited.
2. Users will not hop on and off frequently.
3. The files being shared are very large.
4. Since the files being shared are very large, the number of files will be limited.
5. One or more of the users in the system have the same IP address across sessions.

The rationale for assumption 1 is that the intended user base for the application is a set of persons inside the UCI community. Although use of the program is not limited to these users, the files will only be shared with a known group. Assumption 2 follows from assumption 1. In

order for the system to be of any use, the users must leave the application running to allow other users access to our shared files.

The files being shared in this system are large multimedia (digitized movie) files, with a typical size in excess of 600 MB. Obviously because of this expectation, the number of such files for any given user will be small, most likely limited to fewer than 10 per person (and likely to be less than that).

The algorithms we use for connecting to other nodes require that at least one of the nodes in the system has the same IP address in two consecutive reconnections. The reason for this will become obvious when connectivity is discussed.

This paper consists of several sections. We discuss peer-to-peer file sharing in general, and survey several existing architectures. The sections following our survey give an overview of the Hollyshare system architecture, and then expanded descriptions of each of those modules. The final sections provide an evaluation of our system, and finally conclusions and future work.

Survey

Architecture of Existing Peer-to-Peer File Sharing Systems

Peer-to-Peer (P2P) systems and applications are distributed systems without any centralized control or hierarchical organization [1]. In a pure P2P system, the software running at each node is equivalent in functionality. P2P is not a new concept. However, it has caught the public eye only recently with the familiar peer-to-peer file-sharing network applications such as Napster, Gnutella, Freenet, Morpheus, etc [2, 3]. Until recently, peer-to-peer file sharing applications have followed one of two main models: the hybrid peer-to-peer system with centralized servers such as that used by Napster, and the pure decentralized peer-to-peer system such as Gnutella and Freenet [4]. A third model that harnesses the benefits of the centralized model and the decentralized model has emerged recently. This model is a hybrid where a central server facilitates the peer discovery process and super-peers proxy the requests for local peers and unloads the searching burden on the central server [5]. Applications such as Morpheus and KazaA use the third model. The new versions of some Gnutella applications such as BearShare have also applied the super-peer concept in their file discovery algorithms.

In searching for an appropriate architecture for our HollyShare project, we surveyed the architecture of three models used in peer-to-peer file sharing systems. In the next part of the paper, we will first discuss different systems in terms of their architecture and then compare them in terms of performance, resource requirements, fault tolerance, and scalability. The last part of this survey will concentrate on search algorithms in two decentralized peer-to-peer file-sharing systems: Gnutella and Freenet.

The Centralized Model of P2P File-Sharing

In this model, a central server or a cluster of central servers directs the traffic between individual registered peers [4]. This model is also referred to as a hybrid file-sharing system because both pure P2P and client-server systems are present [6]. The file transfer is pure P2P while the file search is client-server. Both Napster and OpenNap use this model. The central servers in Napster and OpenNap maintain directories of the shared files stored at each registered peer of the current network. Every time a user logs on or off the Napster network, the directories at the central servers are updated to include or remove the files shared by the user. A user logs on the Napster network by connecting with one of the central servers. Each time a user wants a particular file, it sends a request to the central server to which it is connected. The central server will search its database of files shared by peers who are currently connected to the network, and creates a list of files matching the search criteria. The resulted list will be sent to the user. The user can then select the desired file from the list and open a direct HTTP link with the peer who possesses that file. The file is directly transferred from one peer to another peer. The actual MP3 file is never stored in the central server. The central server only holds the directory information of the shared file but not the file itself.

In the centralized model used by Napster, information about all files shared in the system is kept in the central server. There is no need to query individual users to discover a file. The central index in Napster can locate files in the system quickly and efficiently. Every user has to register with the central server to be on the network. Thus the central index will include all files shared in the system, and a search request at the central server will be matched with all files shared by all logged-on users. This guarantees that all searches are as comprehensive as possible. Fig. 1 shows the architecture of the Napster network.

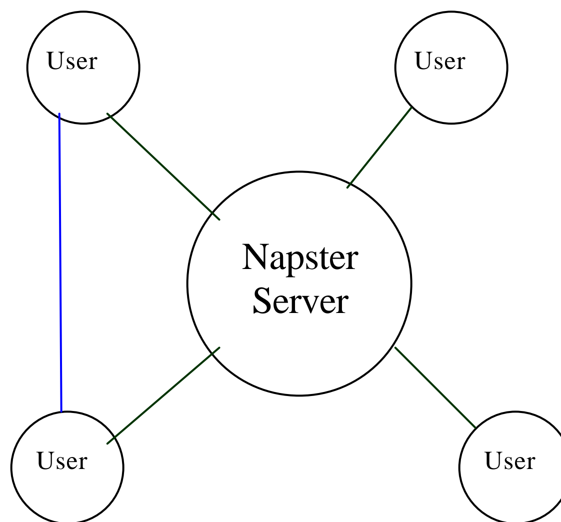

Fig 1 Napster's Architecture.

The Decentralized Model of P2P File-Sharing

In a decentralized P2P file-sharing model, peers have the same capability and responsibility. The communication between peers is symmetric in the sense that each peer acts both as a client and a server and there is no master-slave relationship among peers. At any given point in time, a node can act as a server to the nodes that are downloading files from it and as a client to the nodes that it is downloading files from. The software running at each node includes both the server and client functionality.

Unlike a centralized P2P file sharing system, the decentralized network does not use a central server to keep track of all shared files in the network. Index on the meta-data of shared files is stored locally among all peers. To find a shared file in a decentralized file sharing network, a user asks its friends (nodes to which it is connected), who, in turn, asks their friends for directory information. File discovery in a decentralized system is much more complicated than in a centralized system. Different systems have applied different file discovery mechanisms.

The success of a decentralized file sharing system largely depends on the success of the file discovery mechanisms used in the system. Applications implementing this model include Gnutella, Freenet, etc. We will discuss two file discovery mechanisms used in Gnutella network and Freenet later. Fig 2 shows the architecture of the decentralized model.

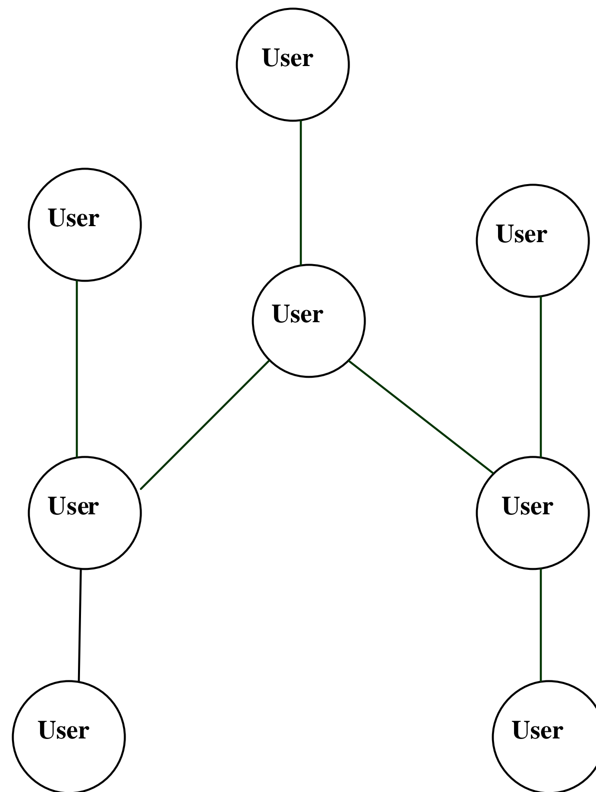

Fig 2 Architecture of a Decentralized P2P File Sharing System

Comparison of Centralized and Decentralized P2P Systems

In a centralized model, file discovery can be done efficiently and comprehensively with the file indexing information kept at central server. At the same time, the central server is the only point of entry for peers in the system. The central server itself can become a bottleneck to the system. A central server failure can lead to the collapse of the whole network. Information provided by the central server might also be out of date because the database at the central server is only updated periodically. To store information about all files shared in the system also requires a significant amount of storage space and a powerful processor at the central server to handle all the file search requests in the system. The scalability of the system depends on the processing power at the central server. When Napster central servers are overloaded, file search in the system can become very slow and some peers might not be able to connect to the network due to the central server's limited capability [7].

In a decentralized model, the responsibility for file discovery is shared among peers. If one or more peers go down, search requests can still be passed along other peers. There is no single point failure that can bring the network down [8]. Therefore, a decentralized P2P system is more robust compared with a centralized P2P system. At the same time, since indexing information is kept locally at individual user's computer, file discovery requires search through the network. File discovery in a decentralized P2P system can become inefficient and produce a large amount of network traffic.

Inspired by the efficient file discovery advantage in a centralized P2P system and the scalability and fault tolerance advantages in a decentralized P2P system, Morpheus and KazaA have implemented pseudo-centralized system architectures to take advantage of the strengths of both the centralized and decentralized model.

Partially Centralized System with Super Nodes

Both Morpheus and KazaA implemented a partially centralized architecture based on technology developed by FastTrack, an Amsterdam-based startup company. Like Gnutella but unlike Napster, Morpheus and KazaA do not maintain central file directories. Like Napster but unlike Gnutella, Morpheus and KazaA are formally closed systems, requiring centralized user registration and logon [5].

Both Morpheus and KazaA implement the FastTrack P2P Stack protocol, a C++-based protocol stack licensed from FastTrack. Although Morpheus claims to be a "distributed, self-organizing network", there is still a central server in the network that is responsible for maintaining user registrations, logging users into the system (in order to maintain active user statistics, etc.), and bootstrapping the peer discovery process.

The registered user can look for super nodes through http://supernodekazaa.com [9]. A user needs to provide user name and password information to get access to super node information. After a Morpheus peer is authenticated to the server, the server provides it with the IP address and port (always 1214) of one or more "SuperNodes" to which the peer then connects. Once the new node receives its list of super nodes, little communication between the central server and the new node is needed for file discovery and file transfer. Upon receiving the IP address and port of super nodes, the new node opens a direct connection with one super node. A SuperNode acts like a local search hub that maintains the index of the media files being shared by each peer connected to it and proxies search requests on behalf of these peers. A super node connects to other super nodes in the network to proxy search requests on behalf of local peers. Queries are only sent to super node not to other peers. A SuperNode will process the query received and send the search results back to the requester directly if the data found in the SuperNode's database. Otherwise, the query is sent to other super nodes for search through the network. Search results in Morpheus contain the IP addresses of peers sharing the files that match the search criteria, and file downloads are purely peer-to-peer. Like Gnutella, files are transferred with HTTP protocol in Morpheus.

Such a scheme greatly reduces search times in comparison to a broadcast query algorithm like that employed on the Gnutella network. Fig 3 shows the architecture of Morpheus network.

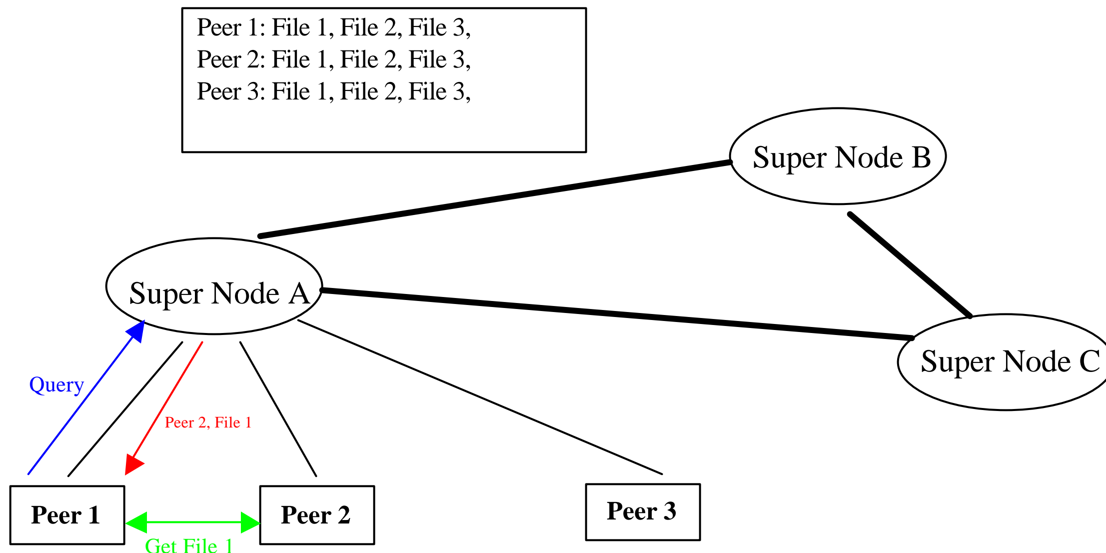

Fig 3 Architecture of Morpheus Network

A Morpheus peer will be automatically elected to become SuperNode if it has sufficient bandwidth and processing power. Peers can choose not to run their computer as SuperNode with a configuration parameter. Since the FastTrack P2P stack protocol used by Morpheus is proprietary, no documentation regarding the SuperNode election process is available. Clip2, a firm that provided network statistics and information about P2P networks before it ceased operations, independently developed the equivalent of a SuperNode for Gnutella, a product called the Clip2 Reflector and generically designated a "super peer." A prototype future version of BearShare (code-named Defender), a popular Gnutella application, implements the super-peer concept in the same integrated manner as Morpheus.

Since the application of the SuperNode concept in Morpheus has been done using proprietary algorithms and protocols, we were unable find the protocol, but we found the paper that proposed a scheme to implement the dynamic SuperNode selection mechanism in the Gnutella network [10]. The approach discussed in the paper may or may not be identical to the proprietary one used in Morpheus, with which we have no way to compare. The following discussion about SuperNode selection is solely based on the scheme proposed for Gnutella

network. It should be noted that a SuperNode is simply a traditional peer, but with super-powers. This is due to the machine properties on which the application is running.

There are two types of nodes in the proposed new version of Gnutella network protocol: super node and shielded node. A super node is just like any other node except that it has better networking capabilities and processing power. These capabilities are not necessary but are the desired ones. A super node also needs to implement additional functionality in order to proxy other nodes. A shielded node is a node that maintains only one connection and that connection is to a super node. A super node acts as a proxy for the shielded node, and shields it from all incoming Gnutella traffic.

When a shielded node A first joins the Gnutella network, it connects to Node B in the normal fashion, but includes the following header in the connection handshake:

GNUTELLA CONNECT/0.6 <cr><lf>
Supernode: False <cr><lf>

which states that Node A is not a super node. If node B is a shielded node, it will not be able to accept any connection. Node B will then reject the connection using suitable HTTP error code. However, Node B can provide IP addresses and ports of few super nodes that it knows, including the one it is connected to, to Node A. Node A can try to establish a connection with one of the super nodes it received from Node B. If Node B is a super node with available client connection slots, Node B will accept the connection from Node A. Node B may also optionally provide some other Supernode IP addresses in the HTTP header sent to Node A for it to cache. These can be used by Node A in the future in case the connection with this super node breaks. If Node B is a super node with no client connection slot available, it will reject the connection but provide Node A with IP addresses and port number of other super nodes it knows so that Node A can connect to the network through other super nodes.

A super node joins the network with connections to other super nodes. It can obtain the addresses and port of other super nodes either through well know host cache such as connect1.gnutellahosts.com:6346) or IRC channels, or through its own cached addresses from previous connections. When a new super node C tries to establish a super node connection to another super node D, node D will accept the connection if it has super node connection slot available. Otherwise, node D will provide it with the addresses and port numbers of other super nodes so that node C can establish connections with other super nodes.

Two schemes were proposed for super nodes to handle the query from shielded nodes. In the first scheme, a super node keeps index information about all files shared by all shielded nodes that connect to it. When a shielded node first connects to a super node, the super node sends an indexing query with TTL=0, HOPS=0 to the client. Upon receiving the indexing query, a shielded node will send in all the files that it is sharing wrapped in Query Replies. The super node indexes the information it received from all shielded nodes. When receiving a query, the super node search through the index in its database to find a match and creates query replies

using the IP address and the port of the corresponding file owner (itself, or a shielded node). In the second scheme, a super node does not keep index information for all files shared by shielded nodes, but instead of keep query routing information received from shielded nodes. When receiving a query, the super node routes it to shielded nodes selectively based on query routing information. A super node can occasionally update the shielded nodes it connects to with the IP addresses and port of other super nodes. The shielded nodes will cache this information. In case this super node goes down, the shielded nodes can use the cached super node information to establish connection with other super nodes.

The following protocol can be used in Gnutella network to assign a node as a shielded node or a super node. If a node has a slow CPU or a slow network connection, it should choose to be a shielded node itself and try to open a connection to a super node. If there is no super node available to accept its connection, it will act as a super node but accept no shielded-client node connection from other shielded nodes.

When a node that has enough CPU processing power and network capability joins the network, it acts as a super node and establishes the configured number of super node connections. At the configuration, a node also set the minimum number of shielded nodes needed for it to be a super node (MIN_CLIENTS) and the time period to reach the number (PROBATION_TIME). The new super node is on probation during the PROBATION_TIME. If it received at least MIN_CLIENTS number of shielded node connection requests during the PROBATION_TIME, it continues to behave as a super node. If the super node failed to receive MIN_CLIENTS number of shielded node connection requests during the PROBATION_TIME, it becomes a shielded client node and tries to connect to other super nodes as a shielded node. Whenever the number of shielded node connections drops below MIN_CLIENTS, a super node will go on probation until PROBATION_TIME. If the number of shielded node connection fails to reach MIN_CLIENTS when PROBATION_TIME is over, the super node becomes a shielded node and establishes a shielded node connection with other super nodes. When a super node goes down, the shielded nodes connecting to it can behave as a super node or a shielded node according to the node's processing power and network capability. If a shielded node chooses to behave as a super node in case of its connection to the super node breaks, it is on probation using the mechanisms described above. Consistent with the autonomous nature of Gnutella network, a node chooses to behave as a shielded node or a super node without the interference from a central server.

Besides the rules described above, a new node can get guidance from the existing super node about whether it should be a super node or a shielded client when it establishes connection with a super node.

With the proposed protocol described above, the new version of the Gnutella network self-organizes into an interconnection of super nodes and shielded nodes automatically. The super node scheme will reduce the traffic in Gnutella network since only super nodes participate in message routing. At the same time, more nodes will be searched because each super node may proxy many shielded nodes. Moreover, many queries can be satisfied locally within the

shielded nodes connecting to the same super node without routing the query to other super nodes. This will also reduce the network traffic related to query.

Since the protocol does not require a central server to assign super node, there is no single point failure that can bring the network down. This partially centralized P2P network is more robust and scaleable than the centralized P2P systems such as Napster and OpenNap. Even in Morpheus and KazaA where a central server is used, the central server only keeps information about super nodes in the system but not indexing information about all files shared in the system. This reduces the workload on central servers in comparison with fully centralized indexing system such as Napster and OpenNap. In the case where the central server of Morpheus or KazaA breaks down, the nodes that have been in the network before can use super node information cached from previous connections to establish a connection to super nodes they knew. However, new users cannot join the network without getting super node information from the central server.

The super nodes in Morpheus and KazaA function differently from the central server in Napster. The central server in Napster just keeps the index of the files shared in the system. The central server itself does not share any file with peers in the system or download files from other peers. In Morpheus and KazaA, a super node itself is a peer. It shares file with other peers in the system. Napster will collapse if the central server goes down. If one or several super nodes goes down, the peers connected to these super nodes can open connection with other super nodes in the system, the network will still function. If all super nodes go down, the existing peers can become super nodes themselves.

Since super nodes keep indexing or routing information about files shared in the local area, searches in these systems is more efficient than that in completely decentralized systems such as original Gnutella network and Freenet. The new version of the Gnutella protocol that is proposed in paper [10] and FastTrack P2P stack used by Morpheus and KazaA reduce discovery time in comparison with purely decentralized indexing system such as original Gnutella network and Freenet. While Morpheus is largely a decentralized system, the speed of its query engine rivals that of centralized systems like Napster because of its SuperNode.

File Discovery in Decentralized P2P File Sharing Systems

Since there is no central directory service available in a decentralized P2P file sharing system. A peer who wants a file is required to search the network to locate the file provider. The success of a decentralized P2P file sharing system largely depends on the success of its file discovery mechanisms. Both Freenet and Gnutella networks are decentralized P2P file-sharing systems, but their file search algorithms are not the same. The next part of this survey will concentrate on the query mechanisms in Freenet and Gnutella network.

File Discovery Mechanisms in Freenet: Chain Mode

Freenet is an adaptive peer-to-peer network of nodes that query one another for file sharing [11,12]. Files in Freenet are identified by binary file keys [11]. There are three types of file keys in Freenet: keyword-signed key, signed-subspace key, and content-hash key. The key for a file is obtained by applying a hash function. Each node in Freenet maintains its own local files that it makes available to the network as well as a dynamic routing table containing the addresses of other nodes associated with the keys that they are thought to hold. When a user in Freenet wants a file, it initiates a request specifying the key and a hops-to-live value. A request for keys is passed along from node to node through a chain where each node makes a local decision about where to forward the request next depending on the key requested. To keep the requester and the data provider anonymous, each node in Freenet only knows their immediate upstream and downstream neighbors in the chain.

The hops-to-live value of the request, analogous to IP's time-to-live, is decremented at each node to prevent an infinite chain. Each request in Freenet also has a unique ID for node to keep track of the requests. A node will reject a request with the ID it saw previously to prevent loops in the network. Messages in Freenet contain a randomly generated 64-bit ID, a hops-to-live value, and a depth counter [11]. The depth counter is incremented at each hop and is used to set hops-to-live value when a reply message is created so that the reply will reach the original requester. If a downstream neighbor rejects a request, the sender will choose a different node to forward to. The chain continues until either the requested data is found or the hops-to-live value of the request is exceeded. If found, the requested data is passed back through the chain to the original requester. If the request times out, a failure result will be passed back to the requester through the same chain that routes the request. The following routing algorithm is used to search and transfer files in Freenet.

After receiving a request, a node first searches its own files and returns the data if found with a note saying that it was the source of the data. If a node cannot satisfy the request with its own files, it looks up its routing table for the key closest to the key requested in terms of lexicographic distance and forwards the request to the node that holds the closest key. If a node cannot forward the request to the best node in the routing table because the preferred node is down or a loop would form, it will forward the request to its second-best, then third-best node in the routing table, and so on. When running out of candidates to try, a node will send a backtracking failure message to its upstream requester, which, in turn, will try its second, then third candidate. If all nodes in the network have been explored in this way or the request T.T.L. reaches 0, a failure message will be sent back through the chain to the node that sends the original request. Nodes store the ID and other information of the Data Request message it has seen for routing Data Reply message and Request Failed message. When receiving a Data Request with ID that has been seen before, the node will send a backtracking Request Failed message to its upstream requester, which may try other candidates in its routing table. Data Reply message in Freenet will only be passed back through the nodes that route the Data Request message previously. Upon receiving a Data Reply message, a node will forward it to the node that the corresponding Data Request message was received from so that the Data Reply will eventually be sent to the node that initiated the Data Request. If a node receives a Data Reply message without seeing the corresponding Data Request message, the Data Reply message will be ignored.

If a request is successful, the requested data will be returned in a Data Reply message that inherits the ID of the Data Request message [12]. The TTL of the Data Reply should be set equal to the depth counter of the Data Request message. The Data Reply will be passed back through the chain that forwarded the Data Request Message. Each node along the way will cache the file in its own database for future requests, and create a new entry in its routing table associating the actual data source with the requested key for future routing. A subsequent request to the same key will be served immediately with the cached data. A request to a "similar" key will be forwarded to the node that provided the data previously. This scheme allows the node to learn about other nodes in the network over time so that the routing decision can be improved over time and an adaptive network will evolve. There are two consequences with this scheme. First, nodes in Freenet will specialize in locating sets of similar keys. This occurs due to the fact that if a node is associated with a particular key in the routing table, it is more likely that the node will receive requests for keys similar to that key. Hence this node gains more experience in answering those queries and become more knowledgeable about other nodes carrying the similar keys in its routing table to make better routing decision in the future. This in turn will make it a better candidate in the routing table of other nodes for those keys. Second, nodes will specialize in storing files with similar keys in the same manner because successfully forwarding a request will gain a copy of the requested file for the node. Since most requests forwarded to a node will be for similar keys, the node will obtain files with similar keys in this process. In addition, this scheme allows popular data to be duplicated by the system automatically and closer to requesters.

To keep the actual data source anonymous, any node along the way can decide to change the reply message to claim itself or another arbitrarily chosen node as the data source. Since the data are cached along the way, the node that claimed to be the data source will actually be able to serve future request to the same data.

File Discovery in Gnutella network: Broadcast

Gnutella is a protocol for distributed file search in peer-to-peer file sharing systems [13]. Applications that implemented the Gnutella protocol form a completely decentralized network. Gnutella was a protocol originally designed by Nullsoft, a subsidiary of America Online. The current Gnutella protocol is the version 0.4 and can be found at [13]. Many applications have implemented the Gnutella protocol: BearShare LimeWire, ToadNode, NapShare are just a few.

Unlike Freenet, a node in the Gnutella network broadcasts to all its neighbors when it requests a file search. There are five types of messages in the Gnutella network: Ping, Pong, Query, QueryHit and Push. Each message in Gnutella contains a Descriptor Header with a Descriptor ID uniquely identifying the message. A TTL field in the Descriptor Header specifies how many times the message should be forwarded. A "Hops" field in the Descriptor Header indicates how many times the message has been forwarded. At any given node z, the "TTL" and "Hops" fields must satisfy the following condition:

$$TTL(0) = TTL(z) + Hops(z)$$

where TTL (0) is the TTL at the node that initiates the message. A node decrements a descriptor header's TTL field and increments its Hops field before forwarding it to any node [13].

When a node joins the Gnutella network, it connects to an existing node and announces it is alive by sending a ping message to the existing node. After receiving a Ping message, a node will send a pong message backwards to the originator of the ping message with the ID inherited from the ping message and also forward the ping message to all its neighbor except the one where it received the ping message. Nodes in Gnutella network keep information about the ping messages that they see in their routing table. When receiving a pong message, a node looks up its routing table to find the connection that the corresponding ping message came from and routes the pong message backwards through the same connection. A pong message takes the corresponding ping message's route backwards. A pong may only be sent along the same path that carried the incoming ping. Only those nodes that routed the ping will see the pong in response. If a node receives a pong with descriptor ID = n, but has not seen a ping descriptor with the same ID, it should remove the pong from the network. The TTL field of the ping message ensures that the ping message will not propagate infinitely.

Queries are routed in the same way with pings. A node posts a Query to its neighbors. When a node sees a Query message, it forwards it to its neighbors and also searches its own files and sends back a QueryHit message to the node that originates the Query if a match is found. A QueryHit message takes the Query's route backwards.

A node forward incoming ping and Query message to all of its directly connected neighbors except the one that sent the incoming ping or Query. If a node receives the same type of message with the ID that it saw previously, it will not forward the message to any of its directly connected neighbors to avoid loops.

The QueryHit message sent back to the Query originator contains the following fields: Number of Hits, Port, IP Address, Speed, Result Set, Servent ID. The QueryHit message does not contain the actual files. After receiving the QueryHit message, a node could download the selected files from the nodes that provide the files directly using HTTP protocol. The actual files are transferred off the Gnutella network.

Gnutella protocol allows a node behind a firewall to share files using Push requests when a direct connection cannot be established between the source and the target node. If a direct connection to the source node cannot be established, the node that requests the file can send a Push request to the node that shares the file. A Push request contains the following fields: Servant ID, File Index, IP Address, and Port. After receiving a Push request, the node that share the file, identified by the Servant ID field of the Push request, will attempt to establish a TCP/IP connection to the requesting node identified by the IP address and Port fields of the Push request. If a direct connection cannot be established from the source node to the requesting node, it is most likely that the requesting node itself is also behind a firewall. The file transfer cannot be accomplished in this case. If a direct connection to the requesting node is established, the source node behind the firewall will send the following GIV request header to the requesting node:

GIV <File Index> : <Servant Identifier>/<File Name> \n\n

Where <File Index> and <Servant Identifier> are the corresponding values from the Push Request. After receiving the above GIV request header, the requesting node constructs an HTTP GET request with <File Index> and <File Name> extracted from the header and sends it to the source node behind the firewall. Upon receiving the GET request, the source node sends the file data preceded by the following HTTP compliant header:

HTTP 200 OK\r\n
Server: Gnutella\r\n
Content-type: application/binary\r\n
Content-length: xxxx\r\n
\r\n

where "xxxx" is the actual file size.

The routing algorithm described above is based on Gnutella Protocol Specification Version 0.4, which is currently used by most Gnutella applications. Several schemes were proposed to extend the Gnutella Protocol Version 0.4 in order to reduce the Query traffic in the network and improve the network's scalability (see references [14, 15, 16, 17] for detailed information about the extension to the current Gnutella Protocol).

Search Performance Comparison of Gnutella and Freenet

The search performance in Freenet follows the Small-World Effect found by Stanley Milgram [1]. In 1967, Stanley Miglram, a Harvard professor, mailed 60 letters to a group of randomly chosen people in Omaha, Nebraska and asked them to pass these letters to a target person in Boston as a part of a social experiment. Those people did not know the target person and they were asked to pass the letters only using intermediaries known to one another on a first-name basis. Each person would pass the letter to one of his or her friends who were assumed to bring the letter close to the target person in Boston. The friend would pass the letter to his or her friend, and so on until the letter reached the target person. Surprisingly, 42 out of 60 letters reached the target person through just 5.5 intermediaries on average. This famous phenomenon was called the Small-World Effect.

The file search process in a decentralized P2P network such as Freenet resembles the social experiment described above. The question is finding the node that holds the file. Each node passes the search request to its neighbor that is thought to most likely hold the file. The search request will eventually reach the node that holds the file through a small number of intermediate nodes because of the Small-World Effect.

This can be seen as a graph problem where people or nodes are vertices and the relationship between people or the connections between nodes are edges. The question is to find a shortest path between two people or two nodes. In a random graph where each of N vertices in the graph connects to random K vertices in the graph, the path length is

approximately logN/logK, which is much better than N/2K, the path length in a regular graph where each node connects to the nearest K vertices. The random connection among people or nodes is what yielded the Small-World Effect.

The search performance in Freenet also has the Small-World Effect, which renders a good average path length. However, in the worst case, the search can result in unnecessary failure due to poor local routing decision. This is true especially at the beginning when nodes do not have enough knowledge about other nodes. As the nodes in the network gain more knowledge about other nodes, the routing decision improves.

With broadcasting, Gnutella queries are guaranteed to find the optimal path in any case. The price Gnutella paid for the optimal path length is a large amount of query traffic over the network.

Problems with Existing P2P File Sharing Systems

Peer-to-peer file sharing has become popular recently, but there is a flip side to the story. The decentralization and user autonomy that makes P2P appealing in the first place also poses some potential problems.

The content of the files shared by individual users could be spurious. However, there is no authority in the P2P system that can remove objectionable or invalid content.
The quality of the download service can vary due to heterogeneous connection qualities. According to one study, 35% Gnutella users have upstream bottleneck bandwidth of at least 100Kbps, but only 8% of Gnutella users have at least 10Mbps bandwidth while other 22% have bandwidth 100Kbps or less [18].

Another problem with current P2P file sharing is Free Riding. According to one study, approximately 66% of Gnutella peers share no files and 73% share ten or less files and nearly 50% of all responses are returned by the top 1% of sharing hosts [19]. The designer of P2P system needs to think of ways to regulate the growth of P2P network to compensate the information providers and discourage the Free Riders.

Most files shared in popular P2P file sharing system are audio or video files. Most of them involve some kind of copyright infringement and intellectual piracy. The lawsuit of Napster has caught public’s attention to this issue. In a decentralized P2P system, there is nobody to sue even though there is copyright infringement. The popularity of the P2P file sharing system has posed a potential threat to the music industry.

Lessons Learned from Surveying the Existing P2P File Sharing System

The first lesson we learned from the existing P2P file sharing system is that there are always tradeoffs in designing a P2P network protocol. Freenet trades off the optimal path length for less query traffic in the network. On the other hand, Gnutella trades off the query traffic for the optimal path length. A centralized P2P system trades off fault tolerance and scalability for quick file discovery. A decentralized P2P system trades off quick file discovery for fault tolerance and scalability. In a partially centralized indexing P2P system, the protocol

designer tries to get both scalability and search efficiency. In designing the HollyShare system, we made a trade-off of storing a complete catalogue in return for no search time.

The second lesson we learned from surveying the existing P2P network is that the users of the P2P file sharing network are heterogeneous in terms of many characteristics: network connection speed, online time, the amount of data shared, etc [18, 20]. The designer of future P2P system must take peers' heterogeneity into consideration when implementing routing algorithms. The HollyShare application we implemented is curtailed to a particular of user population. Peers in the HollyShare network are much more homogeneous than peers in otherP2P file-sharing systems. Several design decisions were made based the assumption that the system is used only by a small group of peers who are geographically close to each other and share a small number of large files among them. These decisions were made to best serve the need of this particular group. If the system were to be used by a large group of people who share an enormous amount of files with various sizes, some design choices would be different. As we saw earlier, P2P system design involves many tradeoffs. We have to make decisions based on our knowledge about our user population.

Implementation

System Architecture

The Hollyshare system architecture consists of 6 modules. The modules include:

- Network Connectivity Manager – responsible for connections between the nodes not relating to direct file transfers (i.e. node database information, shared file information)
- Node Database – responsible for storing and retrieving information about nodes in the system (IP Addresses, port numbers, user IDs, etc.)
- File Database Manager – responsible for storing and retrieving information about shared files in the system (filename, host sharing file, local file information for file transfers, etc.)
- Configuration Manager – responsible for information about the local node (user ID, port information, shared directories, etc.)
- File Transfer Module – responsible for both serving requested files stored at a local node and initiating requests to remote nodes for files
- GUI – the user interface which provides the front end for the users to view and request files for download, and to control local configuration information

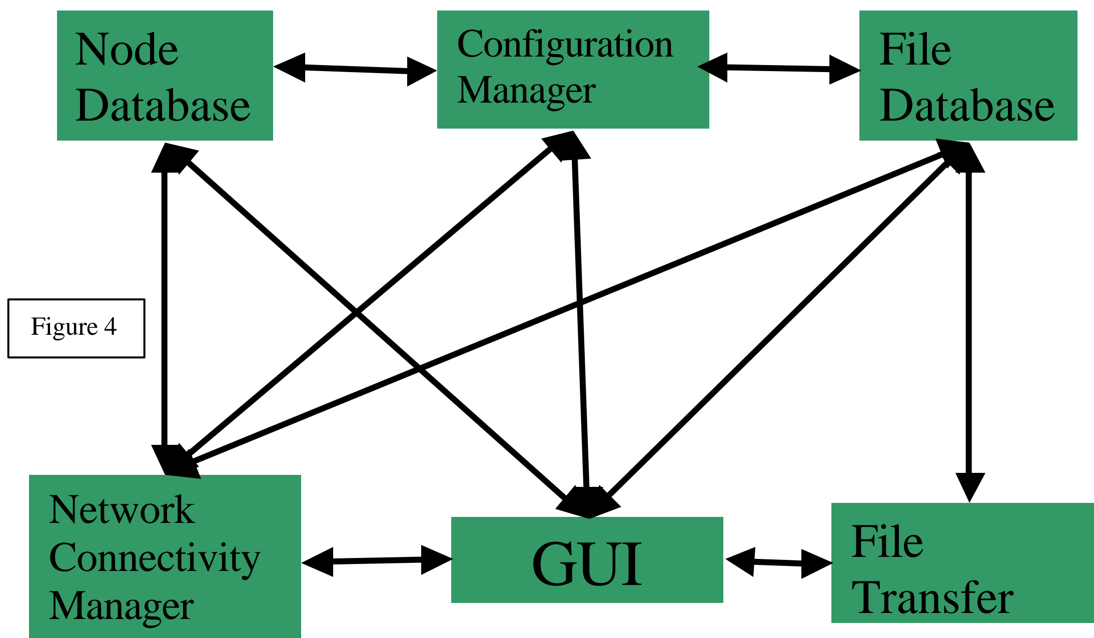

Figure 4

Figure 4 shows a picture of the interactions between the modules. The following sections will each describe one module in detail.

System Overview & Graphical User Interface (GUI)

It is appropriate to discuss the GUI and the system overview simultaneously because the user's familiarity of the system comes from the GUI. The GUI presents the user with enough information to effectively use the system, but hides the complexity of the underlying architecture. The GUI in HollyShare mainly interacts with the following modules: Configuration Manager, File Database, Node Connectivity Module, and the File Transfer Module. To show the interactions, we shall walk through a typical registration and use of the HollyShare application.

Upon the first run of the HollyShare application, the GUI contacts the Configuration Manager to determine whether the application has been registered with the machine the application is running. The Configuration Manager checks for the existence of a file called "config.ini". This file contains information about previous runs of the program. For example, it contains the user ID, the IP addresses for initial connection, and the cached IP addresses of previous connections (Node Database). If this file is not present, the user is presented with a Registration Dialog. This dialog requires the user to register at least one IP-address (provided by whom this user has been invited), and that the user provide an ID. Once the minimum required fields have been registered, the user is presented with the main window.

The main window has many sub-parts. It is primarily a tab-dialog, which allows easy access to information within one window. There are 5 tabs. Each tab presents a different interface.

The first tab listed is the File List tab. This tab presents all files that are available to the user for download from other HollyShare users. This list is provided to the GUI from the File Database. The File Database receives the listings of available files from the Network Connectivity Manager (NCM). The NCM maintains the connectivity of the node. The NCM is aware when a connection to a node has been lost, as well as when a new connection to the system has been made. The NCM defines the set of messages that are used to establish and tear down connections, as well as transfer or propagate file-listing information (the NCM will be described in deeper detail in a subsequent section of this paper). The File Database is notified of the changes, and updates it's listings. The GUI is notified by the File Database, and receives the additional files. Thus, the propagation of new files happens automatically. In contrast, in a decision to reduce traffic on the network, the deletion of files from the list (due to a node disconnection) is handled on a case-by-case or file-by-file basis. The "File List" tab also provides commands to control the download of a file. These commands are download, cancel, and pause/resume. Once the file has been selected, the File Transfer Module requires one more piece of information: the download directory. To reduce the amount of free-riding (this occurs when a user employs the system to obtain files, and does not make files available to other users), we present the user with a dialog to select the download directory from the list of available shared directories. Thus, we are ensuring that the file will be placed in a directory that other users can see. If no download directory is selected, the download does not proceed. If a directory was selected, a download request is sent to the File Transfer Module. The GUI places

the movie entry into the Transfer tab (see below for explanation), and the FTM updates the status of the download.

Another sub-part to the main window is the "Local Files" tab. The information presented in this view, are the shared files from this user. The information is presented by directory, with the files for sharing in this directory, listed beneath it. An additional feature provided is the space remaining for the disk on which the shared directory resides. The command available to the user in this tab is the delete function. This allows a user to delete the file from the directory if they so wish. It should be noted that the review system has not yet been implemented.

The third tab ("Node List") shows the contents of the Node Database. The listings displayed show the IP addresses of currently active (indicated by a 'A' in the active column) nodes, as well as nodes that participated in the group in the past. The date when the node was on last time is provided in the 'Last on' column. The Configuration Manager places the information for the Node Database in permanent storage, as well as provides the list of potential candidates for connection. The current user is always listed, and thus will always be represented as active when connected. A status bar at the bottom of the main window (outside of the tabbed portion of the dialog) indicates how many nodes are connected. "Active Hosts: 1" indicates that this peer is the only node connected to the network.

The fourth tab provided to the user is the "Transfers" tab. This tab shows the current downloads, and uploads taking place. The download status (percentage of completion) value is updated through a slot provided to the File Transfer Module. From this window, it is only possible to cancel a download. The bottom window shows the current uploads. Again, the File Transfer Module updates the status of the transfer.

The fifth and final tab is the "Notifications" tab. This tab provides the user with the system messages. These messages include network connection status, and warning messages. It provides notice that new files have been made available, and is available to any module (including the GUI) that needs to make something made known to the user.

It should be noted, that the user can initiate downloads via the menu system, as well as perform most functionality provided in the "File List" tab. In the User Preferences menu, the user is able to update their registration information and add/remove the share directories.

Node Database

The node database stores all information necessary to create connections between nodes. This includes the IP address for each host, as well as the port number that a given host uses to accept new connection requests. Each node also has a user ID, a string representing a user name or some other identifier to allow Hollyshare users to tell who is on the system (without having to know IP addresses). The database also stores activity data, including an "active" flag and a "last on" field (which is used to delete old nodes that have had no activity for a long time).

The node database provides this functionality with several methods:

- add_node – add a node to the database
- delete_node – delete a node from the system
- deactivate_node – change state of nodes in the database when a user logs out
- get_node_list – returns a list of all nodes in the system
- get_next_node – returns the next node in a random sequence of all nodes stored in the database (active or inactive) – used in connecting to other nodes in the system
- reset_next_node – resets the random node sequence generator
- getUpdatedNodeList – creates a node list for presentation in the GUI

Because the node database is expected to be small, the information is stored in an array of records that can be kept in main memory at all times. Any searches performed in the database are done in a linear manner. The data is unsorted, so no more efficient search is possible. This perhaps limits the scalability of the system, but in all tests we have run, the system performs very well. A possible improvement would be to sort the data while inserting.

The only method that does something unusual is the get_next_node method. There is a private method, generate_random_sequence, which functions behind the scenes. This method takes the current contents of the node database and creates a random sequence of nodes for use by the Network Connectivity Manager to attempt connection.

The node database communicates directly with the GUI, the Configuration Manager, and the Network Connectivity Manager. Communication with the GUI is handled through a messaging system supplied by the QT library known as slots and signals. When a new node is added to the database, or the state of the node changes, a signal is emitted by the database. Any slots in other modules that are connected to the signal execute the methods tied in by the main program (driver).

The node database does not send any updates to the Network Connectivity Manager (NCM). Since any changes to the system are propagated from neighbors, the NCM calls the appropriate functions (add, deactivate, delete, get_next) directly on the node database. Any updates for the GUI are signaled by the node database.

The Configuration Manager (CM) also interacts directly with the node database. When the CM reads the persistent information from the configuration file (the contents of the node database when the user last logged off), nodes are added one at a time to the database. When the CM writes out to the configuration file, it requests the list of nodes directly from the node database, and writes them out to the file.

File Database

The file database stores all the information necessary to allow individual nodes to transfer files between them. This includes such information as the file name, the IP address of the host that

has the file, the port the requester must connect with, the size of the file, and special information if the file is locally stored (path, local flag, etc.).

The file database provides this functionality with several methods:

- find_local_file – used by the file transfer module for serving requests for files stored at the current node
- add_file – add a file to the database
- remove_file – removes a file from the database
- remove_share – remove all files from the database that are stored in the specified share directory (only for local files, but the files removed are propagated to all other nodes in the system via the Network Connection Manager)
- add_share – add all files in the specified directory to the file database (local files), information is propagated to all other nodes in the system via the Network Connection Manager
- non_local_files – returns the list of non-local files for the "File List" tab in the GUI
- all_files – returns a list of all files in the file database
- all_locals – returns a list of all local files in the database for the "Local Files" tab in the GUI
- give_list_of_hosts – returns a list of all hosts serving a specified file
- free_space – a non-portable method that returns the free space on a disk (for allowing the user to choose which directory to download files to)

Like the node database, the internal representation of the file database is an array, but many different representations are supported as return types depending on what is needed. For example, there are two separate lists of files for the GUI: a list of non-local files (files available for downloading), and a list of local files. Each list is returned to the caller in a different format. Once again, all searches performed in the database are linear searches. In many cases, collections of files are returned, or collections of hosts in the case of looking for all users who have a specific file.

The challenge for this particular module was determining exactly what each module required from the database. This involved planning out exactly what information each module would require, and tailoring each representation to facilitate easy access by the calling module.

The Configuration Manager (CM) passes the share directories read in from the configuration file to the file database. The file database in turn goes out to the share directory and adds each file in the directory to the database. Each time a file is added to the database, a signal that a new file is available is emitted, and the Network Connectivity Manager (NCM) requests the list of files and propagates these new files to neighbor nodes, which in turn pass them to the rest of the system.

The NCM adds files to the database one at a time as the information for them is received by directly calling the add_file method. When a new file is added, the file database emits a signal for the GUI to update the "File List" window.

When the user selects a file for download via the GUI, the File Transfer Module (FTM) requests (via the file name and file size) the list for all hosts who are serving that particular file. How the FTM uses that information will be discussed in the FTM section.

Configuration Manager

The configuration manager stores all information regarding the local node including the node ID, connection port, the share directories, and is responsible for saving and restoring the node database to a configuration file.

The configuration manager provides this functionality with the following methods:

- get_shares – returns the list of current share directories
- add_share – adds a directory to the allowed shares, also notifies the file database that a new share directory has been added to update the file database
- remove_share – removes a directory from the allowed shares, and notifies the file database that a share directory has been removed
- write_config_file – writes the configuration information and the node database to the configuration file "config.ini"

The shared directories and user ID are simply strings, and the port information is a 16-bit integer. The shared directories are stored in a linked list. The only time the list changes is either when the program starts up, or the user selects to add or delete share directories via the GUI. When the shared directories changes, the CM calls either the add_share or remove_share method in the File Database (FD), which in turn sends updates to the GUI and Network Connectivity Manager (NCM).

When the program starts, the CM reads in the user ID, the port used for connections in the NCM, the share directories, and the contents of the node database from the previous session. This information is passed on to the appropriate modules as the node initializes. Once the node has been configured, the NCM takes over to attempt connection with nodes in the Node Database. When the user closes the program, the CM writes out the user ID, the connection port, the share directories, and the contents of the node database to the configuration file. The user ID can be changed in the GUI, which passes the update to the CM.

File Transfer Module

The file transfer module is implemented in the class FtManager. The FtManager implements the logic for efficient download of files from peer nodes. The File transfer module has a Server and a Client component. The implementation of the components are described below.

Client component

Client component implements the algorithm to efficiently download a large file in parallel from several servers simultaneously:

User chooses a file from the catalogue that she wishes to download.

- GUI notifies the FtManager about the event.
- FtManager requests list of all active-hosts for that file from file database.
- Client divides the download task of the large file into smaller sub-tasks (i.e. downloading chunks at a time) and puts these into a sub-task stack. For the present we assume that all the hosts of the file are equivalent in all respects so that choice of nodes to connect to can me made serially in order from the list of hosts (or in a random manner). Also currently we have a fixed size (100KB) for each chunk of sub-task being assigned, but sub-tasks are small enough so that most of them could be assigned to the faster connections and thus the bulk of the work is done faster.
- Client then attempts to establish simultaneous connections to a predetermined fixed number of hosts (this number is fixed arbitrarily) from the list of hosts passed to it. Their number is determined by the max_connections_for_a_task constant and is fixed in the current implementation.
- Once a number of connections are established, the client pops a task from the sub-task stack and allocates it to each of the stable connections set up for that file. The sub-task specifies the name and total length of the file (for identification of the file), starting position and the length of the chunk to be downloaded.
- It sets a time-out timer for each one of these allocated sub-tasks.
  - In case the time-out timer signals, the sub-task is considered failed and put back on stack to be reassigned. (There is a data structure corresponding to each File-download which stores the task assignments, timer information, unsuccessful download attempts, socket numbers etc. corresponding to each connection opened for that download).
  - In case of success, the sub-task is complete and a new sub-task is popped from the stack and assigned to that connection.
- If one connection fails multiple times to complete an assigned task then it is closed and a new connection is opened with a new host of the file to replace that connection.
- The number of times that a sub-task can be reallocated, number of new connections that can be opened simultaneously or number of attempts to open a new connection each have limits.
- The dynamic allocation of sub-tasks to different connections allows the file transfer process to distribute the load well amongst the set of connections by utilizing faster connections to the maximum and hence optimizing the performance.

Server component

This component is much simpler. The server accepts connections from the clients attempting to connect. It also receives their requests (for file chunks) and provides the specified file chunks.

File Transfer Module Summary & Future Work:

The File transfer module has a comparatively complex client. The client's basic algorithm has one main goal: load distribution. The rationale for incorporating load distribution and is to ensure successful downloads even in the case of a few failed connections.

The profiling of nodes based on speed and reliability of connection and using these to set up newer connections could be an interesting future work. Also, dynamically adjusting the size of chunks and the number of simultaneously open connections presents interesting possibilities.

Network Connectivity Manager

The whole HollyShare system consists of peer nodes, which are PCs, running the HollyShare application. In this chapter we will discuss the way nodes communicate with each other.

Network topology

All nodes participating in the system have permanently open TCP connections with one or more neighbors. The shared file lists, connectivity messages, and node information are passed through them. The algorithms used in HollyShare were designed to maintain a tree configuration of nodes. The tree configuration ensures that there are no loops in the network. You can see an example of such a configuration in Figure 5.

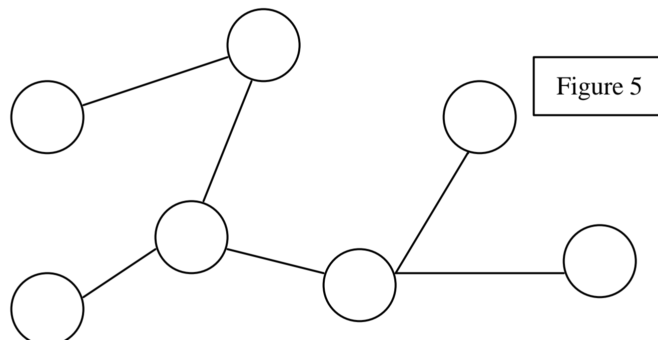

Figure 5

The Network Communication Module (NCM) handles establishment, maintenance, and use of connections within a node. The main aspects of its operation are connecting to the network, reconnecting when one of the nodes exits the application, and propagating the information about shared files in the system. This is discussed in more detail below.

Connecting to the network

Each node in the HollyShare system has its IP address, port number and name of the user (user ID).

The name of the user serves as additional information for display purposes only. No algorithm makes any decision based upon it. The user ID together with an encryption key (given to this participant by the administrator of the group) is supposed to be stored in the *.key file.

However, since encryption is not currently implemented, it is kept in configuration file (config.ini) and can be changed by editing this file.

In order to join the network, the user of the application must know an IP address and port of at least one currently active node. Thus this user must have received an invitation from another user. If the config.ini file, where the persistent node database information is kept, is not present at the start of the program, the user is asked to provide at least one IP address and port of a node in order to try to connect to it. After successful connection to a node, the node database from the connected node (which contains all the currently active nodes and past active nodes) is downloaded to the new node. This information is in turn placed in persistent storage, thus at the next start of the program the user does not have to provide an address for connection.

The following algorithm is used during the connection phase to ensure connections form a tree.

Notes:
1. All nodes that are attempting to connect to the network, but are not connected yet are considered to be in the Init State. Nodes in the Init State do not accept connections from other nodes that are trying to connect to the network (those also in the Init State). Init State nodes that have been asked for a connection reply with a 'Wait' message. When the node has succeeded in obtaining a connection (got reply 'OK' instead of 'Wait' from some node), it goes to state 'On' and starts accepting connections to it.
2. Every node tries to establish only one connection. This ensures the tree structure of connections. In this case the number of nodes is N and the number of connections (edges) is N-1 because the very first node does not connect to another node and any connected graph of N nodes and N-1 vertices is a tree.
3. The 'ID' of the node shall be referred to as the IP address concatenated with port number of a node.
4. A Boolean variable K of a node is used in the algorithm to show if the TCP connection has been established with a node that has a higher ID than the current node.

Connection Algorithm:

1. Create a random permutation of the IP addresses in the node database.
2. Take first address A
3. Assign K = false.
4. Try to connect to A.
5. If it was possible to establish TCP connection to A and (ID of A) > (my ID):
    6. Assign K = true
7. If connection succeeded (my node got the 'OK' answer) – go to state 'On' and exit.
8. If you get a 'Wait' message or didn't get anything – wait till timeout (5 sec currently). If connection succeeded before timeout– go to state 'On' and exit.
9. If there are more nodes in the current permutation:
    10. Take next node A.

11. Go to step 4.
12. If K == true:
13. Go to step 1.
14. Assume that you are the first node in the network and go to state 'On'.

In all cases this algorithm will terminate:

- If there are some nodes in the system, which are in the 'On' state, the algorithm finishes within the first round (loop 1 – 13) as soon as the node tries to connect to that 'On' node.
- If all the nodes are in 'Init' state, the node with the highest ID will exit in step 14 after the first round, and all the rest of the nodes will connect to it (or to other nodes which change their state to 'On') after that.

Reconnection

If any node disconnects, the network gets disconnected into several connected components (the property of a tree). So it is necessary to restore the tree structure of the network connections. Thus a reconnection algorithm is required.

Three algorithms were proposed for this purpose, which are described below. Algorithm 1 seemed to be more optimal at first for two reasons. First, it tries to preserve the connectivity of the connected components, and second, it would add only one connection to join these connected components. It is obvious that it will work in case when only one node goes off. However, it is not clear what will happen if several nodes go off after the start of the reconnection process (during the process of selecting the highest ID of the connected component). Most probably it is possible to create a robust implementation of this algorithm relying on time outs, but in this case it may take the same time or more to reconnect in comparison with Algorithm 2, which is much simpler. We implemented the second algorithm in the HollyShare application.

Reconnection Algorithm 1:

The neighbors of the disconnected node initiate the process of reconnection:
1. Each connected component (group) determines the highest ID (IP || port) present in this group:

a) Propagate message 'Need highest ID' to the leaf nodes
b) Each leaf node replies back with the message 'Current highest ID' with their IDs
c) Each non-leaf node waits for the answers from the neighbors to which he propagated 'Need highest ID' message, selects the highest between that IDs and his ID, and replies back with 'Current highest ID' message using flooding.
d) Finally the nodes that initiated the selection receive the highest ID of their group of connected nodes.

2. The nodes that initiated the selection propagate the highest ID to all the nodes in their group.

3. The nodes that initiated the selection try to reconnect to other nodes in randomly selected order. If the 'group ID' of the node they are trying to connect to is greater than ID of their group, they establish the connection. If not, they break the connection and try to connect to the next node, until they try all nodes in the node database.

The group with the highest ID will not be able to connect to anybody, and group will not be able to connect to itself. Again, we will have N-1 connections per N groups, which ensures tree topology. Figure 6 shows an example of this.

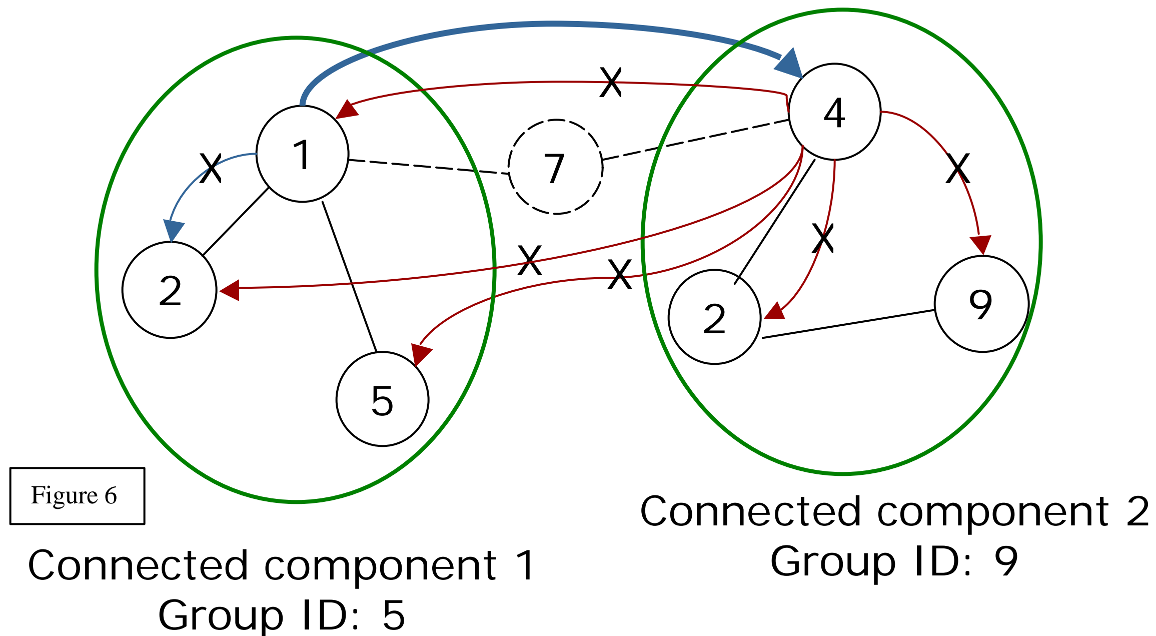

Figure 6

Reconnection Algorithm 2:

If a node detects a disconnection, it closes all connections with the neighbors and after a small delay tries to connect to the network again using Connection Algorithm.

Both Reconnection Algorithms 1 and 2 work even if the computer of the node which goes off has crashed or was turned off without properly closing the program. However, both of them are relatively slow if the number of nodes participating in the network is big. It is possible to significantly reduce this time by Reconnection Algorithms 3, but with one condition: the application program of node, which goes off, should be quitted properly. If the program (or computer) crashes, we still need to use Reconnection Algorithms 1 or 2 for reconnection.

Reconnection Algorithms 3:

1. The node which is going to go off sends 'MemOff' messages to all of its neighbors, letting them know that they should delete all its shared files from their databases and mark him as inactive.
2. The node which is going to go off, which has N neighbors, sends messages to neighbors 1..N-1, asking them to reconnect to the neighbor N. The node also sends a message to neighbor N telling it that (N-1) nodes are going to switch to it. As soon as neighbor N gets these (N-1) new connections, it disconnects the node. After that the node can exit from the program.

Currently only Reconnection Algorithm 2 is implemented.

Information Discovery Mechanism

One of the properties of our system, that distinguishes HollyShare from other existing file sharing applications, is the information discovery mechanism. Instead of using searches that are based on string matching (which is effective technique in the systems with the large number of files) we are using catalogue-based approach. That means that each node displays a synchronized list of all shared files in the system. When new node connects to the systems or adds a new share directory, it sends the information about new files available to all the nodes using a flooding algorithm. Along with file name it sends the file size, the IP address and the port number of the computer that has this file, and the name of the user. To delete a shared file from the list, the node sends the file name, file size, computer IP and the port. All files are identified by the file name plus file size. Thus, if the two nodes have two files with the same file name and size, the system assumes that the contents of the two files are identical.

The flooding algorithm is relatively simple. When a node receives a message (say, to add a shared file), it adds this file to local database and sends the same message to all other neighbors. Since nodes are connected in a tree, no unnecessary messages are sent and no messages arrive at the same node twice. This eliminates the need of tracking of previously received messages or having a TTL-similar mechanism.

HollyShare Network Protocol

All communication in HollyShare is performed by the messages in ***Table 1***. The message consists of the message header, which specifies the type of the message and an optional message body where information is stored. Even though TCP connection provides reliable communication channel, checksums are added in order to ensure integrity of the messages when they are encrypted.

	Bytes	Field	Description
Message Header	0-1	Type	Type of the message, see next table
	2-5	Size	Message body length in bytes
	6-13	Reserved	Should be filled with 0s
	14-15	H_CRC	Message header checksum
Message Body	16 –(15+Size)	Body	Message body
	(16+Size) – (17+Size)	B_CRC	Message body checksum, 0 if message body is empty

All possible message types and body formats are listed in Table 2. *Int32* and *int16* are written MSB – first, LSB – last, and implementation was made in a platform-independent manner, regardless of the byte-orientation of the system memory.

Type #	Type	Description	Message body format
0	Res	Reset – first message after establishing TCP connection	Server port, ID (user name) *int16, null-terminated string*
1	NShare	New share – have some new files to share	List of all shared files that should be added to the file list. *int16 – total number of files in the list* *List of files:* *string – name of the file* *int32 – size of the file* *int8 – has review or not (1 or 0)* *int32 – IP* *int16 – port* *string – computer ID (name of the user)*
2	DShare	Delete file from the shared list	List of all files that should be deleted from the file list *int16 – total number of files in the list* *List of files:* *string – name of the file* *int32 – size of the file* *int32 – IP* *int16 – port*
6	MemOn	Member connected	IP, port#, ID *int32, int16, string*
11	OK	Reply for Res message – connection accepted	server ID and list of IP addresses currently in the system (node database) *string - server ID* *int16 – number of nodes in the list* *List of nodes:* *int32 – IP address*

Type #	Type	Description	Message body format
			int16 – port *int8 – active (1) or not (0)* *string – node ID (user name)* *int16 – year when it was last time on* *int8 – month when it was last time on* *int8 – day when it was last time on*
12	Wait	Reply for Res message – wait for OK message (reconnection is in progress)	server ID and list of IP addresses currently in the system (node database) *string - server ID* *int16 – number of nodes in the list* *List of nodes:* *int32 – IP address* *int16 – port* *int8 – active (1) or not (0)* *string – node ID (user name)* *int16 – year when it was last time on* *int8 – month when it was last time on* *int8 – day when it was last time on*
14	Ping	Request to send a Pong message	Empty
15	Pong	"I'm alive" message	Empty

After establishing a connection (receiving an 'OK' message), both nodes start a timeout timer. They reset a timer after getting any message via this connection. If the timer times out, the node sends a 'Ping' message (the opposite node is supposed to answer with 'Pong' message). If a timeout occurs four times in a row and no messages has arrived, the connection is considered broken and is closed by the node where the time outs have happened (after that a reconnection phase occurs).

A typical scenario of exchanging messages right after establishing TCP connection is shown in Figure 7.

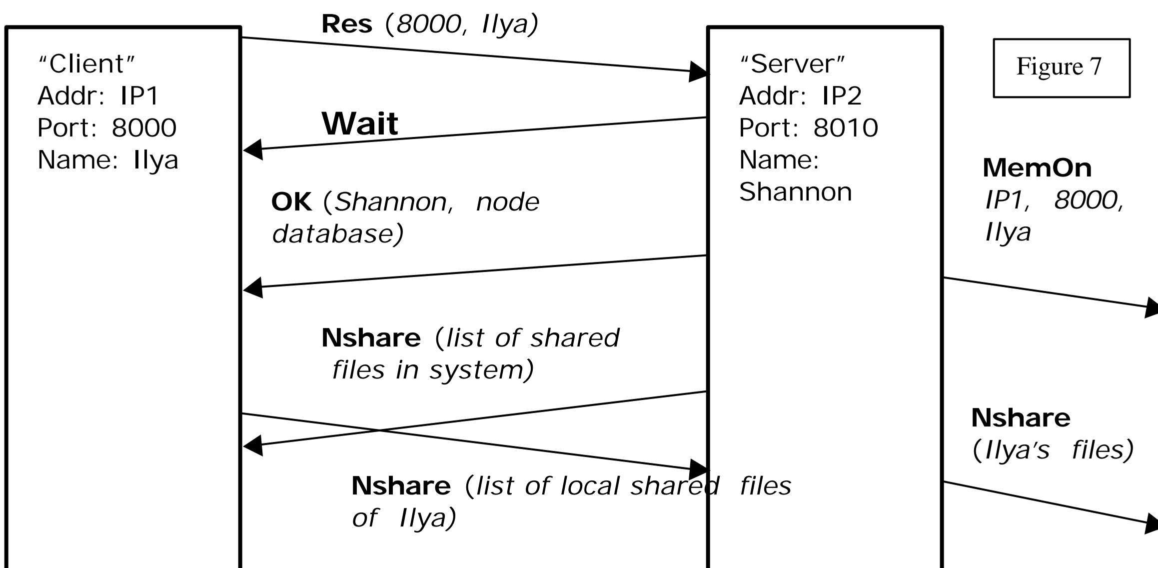

Figure 7

A real trace of starting a HollyShare application and joining the network is shown below.

Trace	Comments
-----GUI: Welcome To HollyShare-----	Initial greeting
Debug: NetworkManager::connectToNetwork	
Debug: Connecting to host 128.195.10.141, port 8000, index 0	Took first host from the node database,
Debug: NetworkManager::addConnection: socket 472, index 1	it happened to be the address of the computer
Debug: Connection from: 128.195.10.141 Port: 3631	on which application is running
Debug: NetworkManager::socConnected	TCP connection succeeded
Debug: NetworkManager::sendMessage, index 0, type Res	
Debug: NetworkManager::processNewMessage, index 1, type Res	
Debug: Message from: 128.195.10.141 Port: 3631	
Debug: (socNetwConnect: 128.195.10.141 Port: 3631, soc 184)	
Debug: Can't connect to myself	Application detected that it is trying
Debug: NetworkManager::deleteConnection, index 1, IP 0.0.0.0	to connect to itself
Debug: NetworkManager::connectionDropped, index 0	
Debug: NetworkManager::deleteConnection, index 0, IP 128.195.10.141	
Debug: NetworkManager::connectToNetwork	
Debug: Connecting to host 128.195.11.218, port 8000, index 0	Took next address from the node database
Debug: NetworkManager::socConnected	TCP connection succeeded
Debug: NetworkManager::sendMessage, index 0, type Res	Sent 'Res' message
Debug: NetworkManager::processNewMessage, index 0, type OK	Got 'OK' answer – connection is successful
Debug: Message from: 128.195.11.218 Port: 8000	
Debug: (socNetwConnect: 128.195.11.218 Port: 3632, soc 456)	
Debug: NetworkManager::sendMessage, index 0, type NShare	Sent its shared file list
Debug: NetworkManager::processNewMessage, index 0, type NShare	Got the list of the shared files in the system
Debug: Message from: 128.195.11.218 Port: 8000	
Debug: (socNetwConnect: 128.195.11.218 Port: 3632, soc 456)	
Debug: NetworkManager::processNewMessage, index 0, type Ping	Got 'Ping' request
Debug: Message from: 128.195.11.218 Port: 8000	
Debug: (socNetwConnect: 128.195.11.218 Port: 3632, soc 456)	
Debug: NetworkManager::sendMessage, index 0, type Pong	Replied by 'Pong' message
Debug: NetworkManager::processNewMessage, index 0, type Ping	Got 'Ping' again
Debug: Message from: 128.195.11.218 Port: 8000	…..
Debug: (socNetwConnect: 128.195.11.218 Port: 3632, soc 456)	
Debug: NetworkManager::sendMessage, index 0, type Pong	

Project Development

The following section describes how the development of the HollyShare project took place. The final project resulted in 16 header files, 17 implementation files, and over 8000 lines of code developed by 4 members.

Requirements Phase

There were approximately 5 initial meetings to establish the project idea. We had several. We were thinking of a chat application, an on-line game system, and a movie-sharing application. The winning idea was the peer-to-peer movie-sharing application.

Next, the group determined the basic requirements and assumptions of our application. They are/were as follows:

- develop a peer-to-peer application
- share large multimedia files
- small group of users (users are known to each other)
- list all shared files

We also specified the initial peer-to-peer systems that we wanted to survey for our final paper.

Architecture Development

Again, there were approximately 5 meetings to establish the architecture required to meet the requirements of our system.

From Requirements ⟹ what Modules are needed

- **Transfer Files**: File Transfer Module
- **Connect/Disconnect/Remain**: Network Connectivity Module
- **Support/Databases**: Node Database(who is connected), Configuration Manager(user/system settings), File Database(manages list of users files)
- **User/System Interaction**: GUI

Distribution of Modules

The team held approximately 2 meetings to establish which member would develop which module(s). The following were the assignments:

- User/System Interaction: Shannon Tauro
- Support/Databases: Sean McCarthy
- Transfer Files: Bijit Hore
- Connect/Disconnect/Remain: Ilya Issenin

Development & Integration of Modules

The development of HollyShare took place on Windows NT workstations using C++ & Qt Library. Each member developed his or her modules independently. We met after classes to discuss progress, concerns, and provide assistance. We sent countless E-mails to notify of/send updated project files, request for services to be added in a module, and integrate. We had originally planned an integration week for week 9, however one member (Ilya Issenin) took it upon himself to integrate and resolve conflicts between modules. Thus, throughout development, we were able to develop & test using a functioning application.

Division of Labor

Given below are the team members and their contributions to the project.

Shannon Tauro:

Graphical User Interface (GUI), final presentation, System Overview & GUI + assembly/correction of final report Team Coordinator (i.e. deadline manager)

Sean McCarthy:

Node Database, File Database, Configuration Manager, third presentation, Introduction + Node Database + Configuration Manager + File Database + Analysis + Conclusion chapters in final report + assembly of final report
Ilya Issenin:
HollyShare Network protocol and reconnection algorithms, + File Transfer Module
Network Connectivity Manager module, integration of all the modules, debugging, initial presentation, network topology + Node Connectivity Module in final report
Bijit Hore:
File Transfer Module, second presentation, File Transfer Module in final report
Songmei Han:
Survey presentation, survey part in final report

Analysis of System

In order to analyze our system, we chose to track both the numbers of messages passed between nodes and the size of those messages. To set up the tests, static counters were placed in the Network Connection Manager to keep track of messages and the bytes of information being sent out. Quite simply, every time a message was sent from a node, the message counter was incremented, and the running byte total was updated with the size of the message.

Two portions of the system were measured: the connection traffic, and file database change propagation traffic. The connection traffic includes the connection establishment between nodes, as well as the propagation of the node database information and file database information for the new node.

We tested the system in several different configurations, from 2 nodes, up to 6 nodes. Each configuration was tested separately, restarting from scratch each time. This was done to insure that all tests were conducted in a uniform manner.

In order to test the connectivity traffic, nodes were added to the system one at a time until the number of nodes in that configuration (2, 3, 4, 5, or 6) was reached. Once the system had settled (once all non-ping/pong traffic ceased), the measurements for both the number of messages and the bytes of messages were taken. You can see from figures 8 and 9 that the increase in traffic is linear in both cases. This is far from unexpected, as there are no loops in the network topology so no duplicated messages are sent. The variation from straight linearity observed in the graph is caused by each node having a different amount of information to add to the system when connection is established (each node shares a different number of files). The minimum amount of information required to maintain both the node and file databases is transferred during connection.

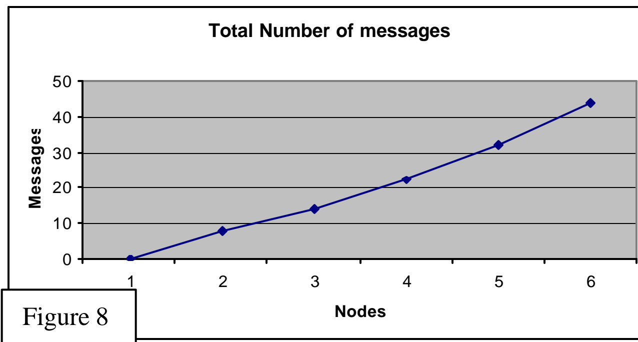

Figure 8

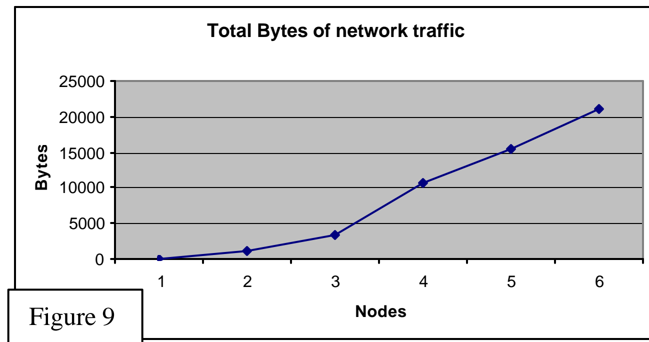

Figure 9

In order to test the propagation traffic, nodes were added to the system one at a time until the number of nodes in the configuration was reached. Once the system had settled, a base measurement for each node was taken. Then a share directory (containing 427 files) was added at one node. Once the new file information had propagated to all other nodes in the system, another measurement of the number of messages and bytes of traffic was taken at each node, compared with the base measurement, and the difference between the two was determined as the propagation traffic. As you can see from figures 10 and 11, the propagation traffic is strictly linear. This is exactly what was expected, as the same amount of information (both number and size of messages) was propagated to all other nodes in the system.

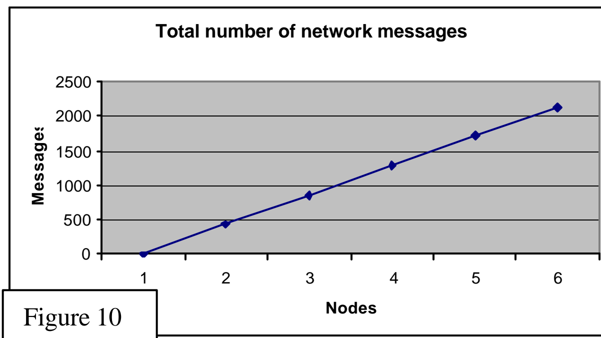

Figure 10

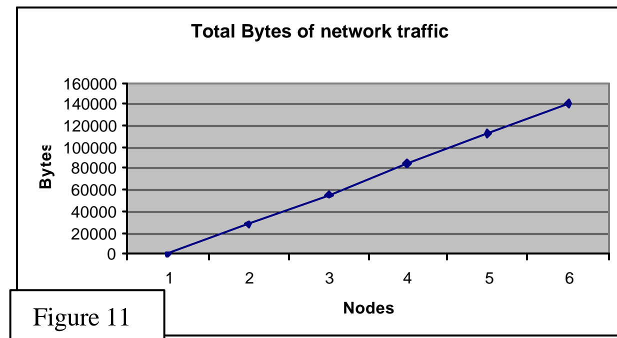

Figure 11

In order to test the system further, we want to set up specific controlled conditions. Our test bed for the measurements we took was Windows workstations running on the campus network. The campus network has traffic from many users constantly. In this sort of situation, making accurate measurements of time for file transfers and information propagation is at the very least inaccurate.

To actually measure these things, we would want an isolated network where the only traffic is the Hollyshare traffic between active nodes on the system. All nodes on the network should have their clocks synchronized in order to allow measurement of connection and propagation times. Since we can measure exactly how many bytes are sent in each message (the header size is known, and we can count the number of bytes sent out).

Actual speed of propagation would be highly dependent on the topology of the network.

Obviously, it will take more time to propagate to the rest of the nodes in the system if we choose node 3 than node 1.

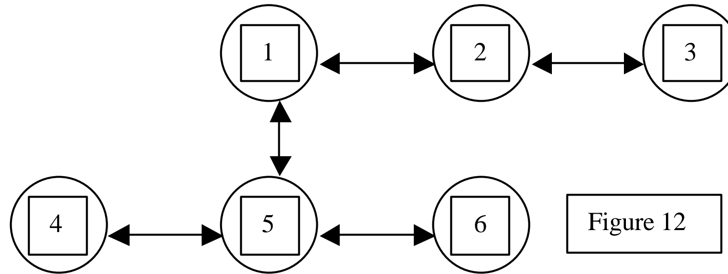

Figure 12

Our test should be a worst-case topology, which would simply be such that the nodes would be connected in a straight line, like the topology below.

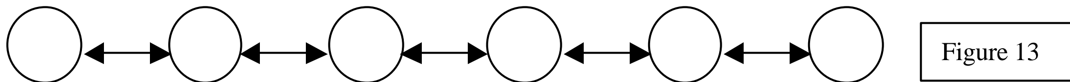

Figure 13

In this situation, we would select one of the end nodes, and measure the time to propagate everything to the other end of the network. Since we know the bandwidth of the connection media, we could measure the total time and determine the processing time required at each node. This would give us a good indication of the sort of scalability of our system by increasing the size of our network, and pushing more and more data through.

We know that the number of messages and amount of data increases linearly in the number of nodes in the system, and it is reasonable to expect that the time to propagate from one end of the network to the other would also increase linearly. Since each node is running the identical program, processing time at each node should be the same at each point in the network.

Conclusions

Our goal for this project was to produce an application that would allow us to easily share large multimedia files among a small group of known users. In pursuit of this goal, we learned much about peer-to-peer file sharing applications. There are several different architectures, each with its own strengths and weaknesses. Despite these differences, there are some common threads among all such applications.

Firstly, files are distributed among users of the system (peers). The notion of decentralized storage allows for sharing a very large number of files, where storage space on centralized servers is typically limited. In a peer-to-peer system, if you wish more storage space, you need only add another node. Another consequence of this is that such a system quite often has a built-in redundancy. If a particular file is popular, multiple users likely share it.

Having files on multiple nodes also distributes the download traffic. Unlike a centralized system where all download requests would go to the central server, in peer-to-peer each download request is passed among peers. This may increase overall network traffic, but also has a good chance of relieving system bottlenecks that often occur in client/server systems.

Another ability provided by peer-to-peer systems that can't be done in a centralized system is parallel downloads. It is possible to open up multiple connections with more than one node to download files. This allows for files to be obtained more quickly than might normally be possible in a centralized system, especially if the system is busy. Downloading from peers distributes the system load so that system bottlenecks can hopefully be avoided.

HollyShare was designed with these ideas in mind. Each node in the system is identical, providing a completely decentralized architecture. Files shared by multiple nodes can be downloaded from multiple sources in parallel, providing reliable, high-speed transfers. Users can share many files easily, and have easy access to any files shared by other users. HollyShare is a flexible peer-to-peer application with many features (mentioned in the report) that can be added in the future.

REFERENCES

1. Yima, The survey of the technologies of peer-to-peer.
2 Fox, G. Peer to Peer Networks.
3. Parameswarn, M.; Susarla, A. & Whinston, A. P2P Networking: an Information-Sharing Alternative.
4 Modern peer-to-peer file-sharing over the internet. http://www.limewire.com/index.jsp/p2p
5. http://www.openp2p.com/pub/a/p2p/2001/07/02/morpheus.html
6. Yang, B. & Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems.7. http://www.napster.com/help/win/faq/#x-2
8. What is Gnutella? http://www.gnutellanews.com/information/what_is_gnutella.shtml.
9. Sander, S. Investigating one incidence of anomalous network traffic.
10. Supernode specification.

    http://groups.yahoo.com/group/the_gdf/files/Supernodes.html
11. Clarke, L.; Sandberg, O.; Wiley, B.; Hong, T.W. (Edited by: Federrath, H.) *Freenet: a distributed anonymous information storage and retrieval system.* Designing Privacy Enhancing Technologies. International Workshop on DesignIssues in Anonymity and Unobservability. Proceedings (Lecture Notes in Computer Science Vol.2009), (Designing Privacy Enhancing Technologies. International Workshop on Design Issues in Anonymity and Unobservability. Proceedings (Lecture Notes in Computer Science Vol.2009), Designing Privacy Enhancing Technologies. International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, 25-26 July 2000.) Berlin, Germany: Springer-Verlag, 2001. p.46-66
12. Clarke, I. *A distributed decentralized information storage and retrieval system.* unpublished dissertation, University of Edinburgh, 1999.
13. The Gnutella protocol specification v0.4. Clip2 distributed search services, http://www.limewire.com/index.jsp/developer.
14. Gnutella 0.6 Protocol Extension**:** Handshaking Protocol (also called the LimeWire Connection Proposal): http://groups.yahoo.com/group/the_gdf/message/2010
15. Proposed Gnutella Protocol Extensions: Ping/Pong Scheme: http://www.limewire.com/index.jsp/pingpong.
16 MetaData Proposal: http://www.limewire.com/index.jsp/metainfo_searches, http://www.limewire.com/developer/MetaProposal2.htm.
17 Query Routing: http://www.limewire.com/developer/query_routing/keyword routing.htm.
18. Sarious, S., Gummadi, P.K., Gribble, S.D. A measurement study of peer-to-peer file sharing systems.

19. http://www.firstmonday.dk/issues/issue5_10/adar/index.html.

20. Gedik, B. Determining Characteristics of the Gnutella Network.