Napster and Gnutella: a Comparison of two Popular Peer-to-Peer Protocols

Anthony J. Howe
Supervisor: Dr. Mantis Cheng
University of Victoria

February 28, 2002

Abstract

This article presents the reverse engineered protocols of two popular peer-to-peer models, Napster and Gnutella. Napster presents a model where a central broker handles discovery and coordination of files among peers, but the exchange of files takes place between the peers. Gnutella removes the centralization and extends the model further by requiring the peers to contribute to the coordination and discovery efforts. Napster is a very scalable model but its resiliency is reduced since discovery and coordination are centralized. In contrast Gnutella is resilient since there are no centralized components, but it is not scalable since the structure of Gnutella produces a large number of messages. A system is scalable if the load on the system increases at a linear rate with an increase in users. A system is resilient if it is able to function correctly after one or more component failures. An alternative peer-to-peer solution using the scalability of Napster and the resiliency of Gnutella is presented at the end of this work.

Contents

List of Figures

1 Introduction

Traditional network (internet/intranet) applications are client-server based, where many clients communicate with a common shared server for application services. Examples of such application services include e-mail servers, web servers, and file servers. These centralized servers have two fundamental problems: scalability and resiliency. In the present state of the Internet, millions of users may be using the same server simultaneously. It is difficult to host a server for millions of users and remain online continuously.

An alternative to the client-server architecture is the peer-to-peer model. Every client in a peer-to-peer network is also a server. The coordination and discovery issues of these decentralized networks are central. To better understand these issues the protocols of two popular peer-to-peer applications Napster and Gnutella are closely examined. Each protocol presents a distinct approach to the coordination of information exchange between peers, and discovery of the information contained on those peers. The degree of resiliency and scalability vary between these peer-to-peer models. A system is scalable if the load on the system increases at a linear rate with an increase in users. A system is resilient if it is able to function correctly after one or more component failures.

Section 2 and section 3 present and examine the reverse engineered protocols of the Napster and Gnutella peer-to-peer models respectively. Section 4 discusses the advantages and disadvantages of each approach in peer-to-peer networking. After the discussion, section 5 proposes a better peer-to-peer network design. Section 6 concludes this work.

2 Napster

Napster has been described as the trigger application that made peer-to-peer web computing popular [1]. Pre peer-to-peer web applications such as ftp, shared drives, and Windows for workgroups did not have the ease of use, common protocols, standards, and scalability of Napster [1].

2.1 Overview

Napster hosts act as clients as well as servers for the exchange of music files. A host first joins the network by connecting to a central server known as a broker. Once connected, the host passes information on all the music files it serves to the broker. This information is known as metadata. The broker stores a database of the metadata; this metadata contains the information of all the hosts currently logged into the broker. In Figure 1 there are six computers logged into the broker.

Clients query the broker's database for particular music files. The broker replies back with a list of songs and matching peers that contain them. The client can then coordinate with the broker on the exchange of a file from one of the remote hosts. For example, in Figure 1 Computer F is downloading a file directly from Computer C. The broker participated in searching for the file and setting up the file exchange, but the download occurred directly between the two peers.

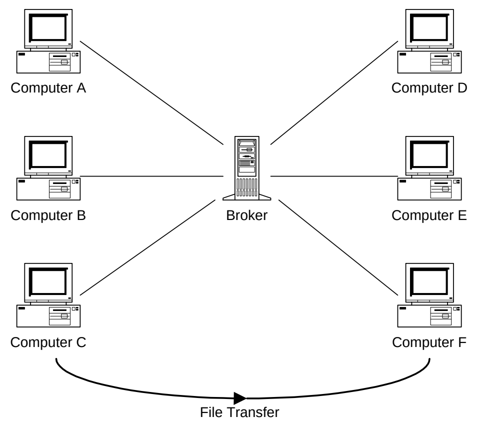


Figure 1: A Napster Network: the broker coordinates music file sharing among peers

In addition to searching and sharing music, Napster also provides peer-to-peer messaging, chat rooms, and user hot lists. Peer-to-peer messaging allows one peer to talk to another peer. Chat rooms allow groups of users to share information. A message posted to a chat room is seen by all users connected to the chat room. Hot lists contain a list of popular peers with whom a client has been in contact. Peers can add each other to their own hot lists. This hot list will provide information on a peer's metadata, as well as when the peer is online. The broker performs all the coordination of these extra features.

Due to the popularity of Napster, many Napster brokers are available to handle millions of users. Before August 2000, these brokers were incapable of sharing databases. This meant that each broker contained a separate Napster network [5]. In August Napster linked all brokers together so that they formed one large network allowing any user to query the metadata of all users connected to the Napster network [4]. Since that time Napster has kept its brokers unlinked most of the time [4]. It is suspected that while the brokers were linked together, the coordination of information between the brokers was too much of a load on the network. Figure 2 shows the number of users on the Napster Network for the week of November 18, 2000[1]. On average there were 871000 users on the Napster Network reaching a maximum of 1179433 users on November 20, 2000 at 4:00pm. Assuming there were 90 Napster brokers available all the time this led to an average load of 9680 users per broker with a maximum average load of 13104 users per broker when the network reached its maximum. These high numbers of users may have contributed to the cause of unlinking all the brokers.

[1]These results were obtained by periodically observering the ***server stats*** message of each of the 90 Napster brokers on IP range 64.124.41.150 to 64.124.41.239. Assuming that all brokers were unlinked, the number of users is the sum of all the users reported by each broker.

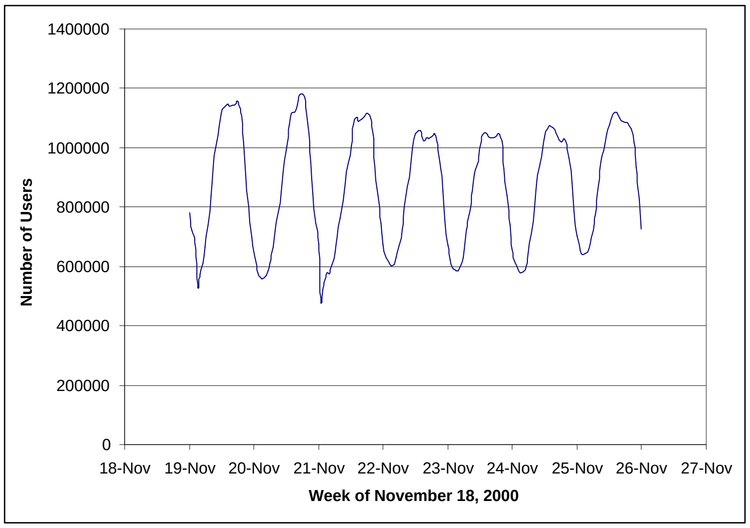


Figure 2: The Number of Napster Users for the Week of November 18, 2000

2.2 The Protocol

Napster has not released the specification for the Napster protocol. However, a project has been formed called the OpenNap project to reverse engineer the protocol. Most of the protocol messages have been successfully reversed engineered and can be found in a document on the OpenNap web site [3]. The protocol is large and contains many sections dealing with login, file sharing, chat rooms, and peer-to-peer chatting. This work is only concerned with the discovery issues related to the sharing of files between peers, so only the login and file sharing portions of the protocol are discussed.

The Napster messages used in communication between a Napster peer and a Napster broker are shown in ***bold and italic***. The description of each message and the information they carry have been reversed engineered and are explained on a web page named *Napster Messages* [3].

To evenly distribute peers among Napster brokers there are lookup servers to provide addresses of the least busy brokers. A Napster peer must first connect to a lookup server, get the address of a broker and then finally connect to that broker. In Figure 3 there are *m* Lookup servers and *n* Napster brokers. The lookup servers have connections to each of the brokers to determine the load on each broker. The brokers are interconnected in order to share the metadata. By sharing metadata the brokers as a whole act as one large Napster network instead of *n* different Napster networks.

The Napster lookup servers, brokers, and peers all communicate through TCP/IP. When discussing communication between lookup servers, brokers, and peers it can be assumed that the messages are delivered in order and without error. To join the Napster network Napster peers connect to a lookup server located at *server.napster.com* on TCP port 8875. The address of the broker returned from the lookup server includes the IP

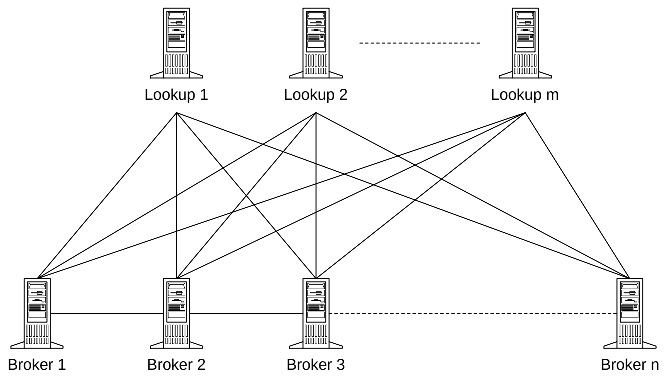


Figure 3: Napster Lookup Servers Provide address of least busy broker

address as well as the TCP port. The TCP port for a broker is usually port 8888. A Napster peer will normally listen for requests on TCP port 6699[2].

2.2.1 Napster Peer

There are up to five concurrent entities in a Napster peer shown in Figure 4: Main Coordination, Listener, Download Instance, Upload Instance, and Push Instance. The Main Coordination entity deals with the connection, and communication with a Napster lookup server and broker. The Listener handles all incoming connections from other Napster peers. The Upload, Download, and Push Instances are concerned with the exchange of files between other Napster peers. There may be zero or more of the Upload, Download, and Push Instances running at one time.

The state-chart for the Main Coordination entity is given in Figure 5. It has eight states: **offline**, **finding best broker**, **connecting to best broker**, **online**, **login error**, **logged in**, **metadata upload**, and **searching**.

Initially Main Coordination starts in the **offline** state. After connecting to a Napster lookup server it enters into the **finding best broker** state. If the lookup server is busy or there is no connection to it the state returns to **offline**. Otherwise it waits for the identification of a broker. Upon receiving identification, it closes the connection to the lookup server and enters into the **connecting to best broker** state.

In the **connecting to best broker** state the peer will open a connection to the broker address provided by the lookup server. If no connection is made to the broker, the Napster peer returns to the **offline** state. Otherwise it will create a new Listener entity and send a ***login*** message or ***new user login*** message to the broker and enter the **online** state. Either login message includes the username, password, and the listening port. The listening port is used to accept incoming connections from other Napster peers.

Username acceptance takes place in the **online** state. A failed ***login ack*** message from the broker or a connection error will cause a change in state to **login error**. Once

[2]The port numbers of the brokers, lookup servers, and peers were determined by using a packet sniffer.

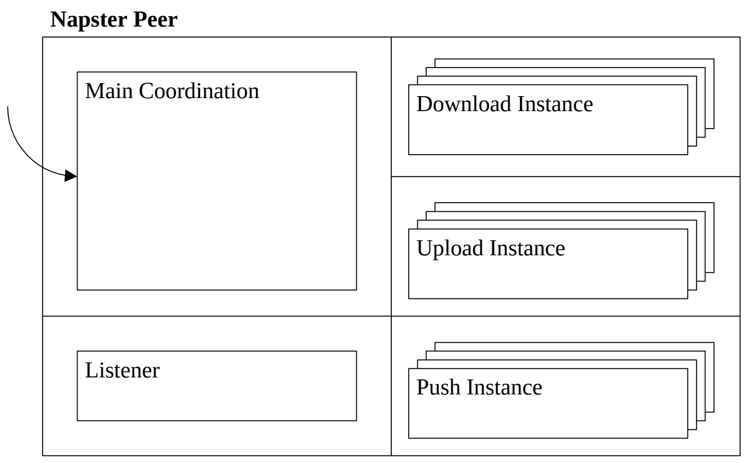


Figure 4: State-chart for the Napster Peer

in the **login error** state the connection to the broker will be closed, the Listener terminated, and the Napster peer will return to the **offline** state. A failed login will occur for a ***login*** message if the username or password is invalid. A failed login will occur for a ***new user login*** message if the username already exists, or if the username and password are of an invalid format. If the broker accepts the login, the Main Coordination entity will enter the **logged in** state.

Immediately following the successful login, the Napster peer will upload metadata to the broker describing the song instances. Upon sending the metadata for the first song by a ***client notification of shared file*** message, the Napster peer will enter the **metadata upload** state. It will stay in this state until every song notification has been uploaded to the broker. The peer will return to the **logged in** state upon the last metadata notification.

While in the **logged in** state the Napster peer may search for a song. Sending a ***client search request*** message to the Broker will cause it to enter the **searching** state. A ***client search request*** message contains a search string as well as additional information to help narrow a search. A timer will also be set when entering this state. This timer will allow the peer to get out of the **searching** state if the broker does not answer the search request. During the **searching** state the broker will send a ***search response*** message for every song, up to 100 songs, matching the search request. The ***search response*** message contains information relating to the song as well as the peer username that hosts the song. The broker will end the search with an ***end of search response*** message. At this point the state will change back to **logged in**.

In any one of the **logged in**, **metadata upload**, or **searching** states the peer may receive any one of two local events or six messages from the broker: local file deletion, user download request, ***server stats***, ***upload request***, ***alternate download ack***, ***download ack***, ***queue limit***, or ***get error***. When a song is deleted from the local repository, the broker is notified that the song is no longer available for download with a ***remove***

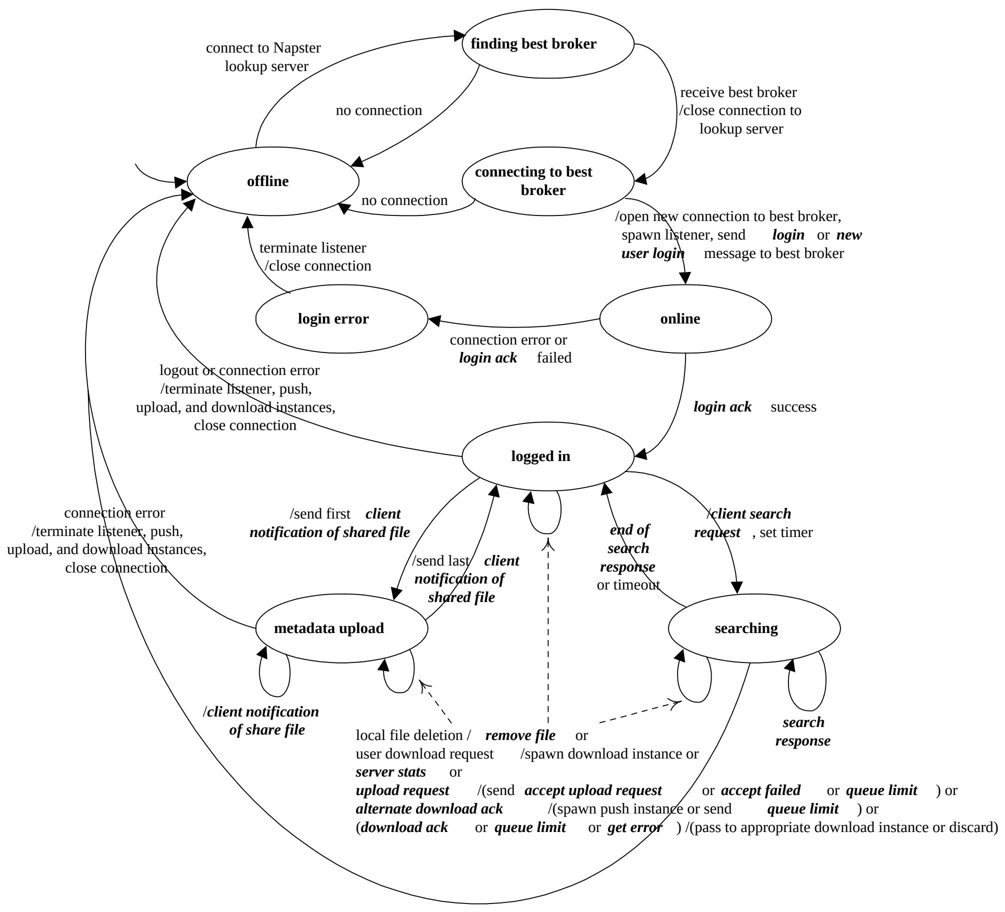

finding best broker
connect to Napster lookup server
no connection
receive best broker /close connection to lookup server
offline
connecting to best broker
no connection
/open new connection to best broker, spawn listener, send ***login*** or ***new user login*** message to best broker
terminate listener /close connection
login error
online
connection error or ***login ack*** failed
logout or connection error /terminate listener, push, upload, and download instances, close connection
login ack success
logged in
connection error /terminate listener, push, upload, and download instances, close connection
/send first ***client notification of shared file***
/send last ***client notification of shared file***
end of search response or timeout
/***client search request***, set timer
metadata upload
searching
/***client notification of share file***
search response
local file deletion / ***remove file*** or
user download request /spawn download instance or
server stats or
upload request /(send ***accept upload request*** or ***accept failed*** or ***queue limit***) or
alternate download ack /(spawn push instance or send ***queue limit***) or
(***download ack*** or ***queue limit*** or ***get error***) /(pass to appropriate download instance or discard)

Figure 5: State-chart for the Main Coordination

file message. After searching for a file the user of the Napster peer may decide to download a song. A user download request will result in the spawning of a Download Instance. The Download Instance will handle the download of a song from a remote peer. A ***server stats*** message is sent out periodically from the broker, and provides information on the total number of peers, the total number of shared files, and the size of information available in the Napster network. An ***upload request*** message is sent by the broker to notify the peer that a remote peer is requesting a download. An ***upload accept*** message will be sent to the broker if the peer is able to fulfill the request. If the file requested for download does not exist then an ***accept failed*** message is sent instead. If the maximum number of Upload Instances has been reached then a ***queue***

limit message will be sent. An ***alternate download ack*** message will be sent from the broker requesting that the peer push a file to a remote peer. This will occur if the peer is behind a firewall. The peer will either spawn a new Push Instance to handle the push or send a ***queue limit*** message to the broker if the maximum number of uploads has been reached. A download information message such as a ***download ack***, ***queue limit***, or ***get error*** message will be passed to the appropriate Download Instance, or if a Download Instance does not exist then the message will be discarded.

If the connection is lost while in the **logged in**, **metadata upload**, or **searching** states or the Napster peer decides to log out in the **logged in** state, all the download, push, upload, and Listener entities are terminated and the connection to the broker is closed.

The state-chart for the Listener entity is given in Figure 6. It has four states: **waiting for peer connection**, **determine connect request**, **handle a receive request**, and **handle a download request**.

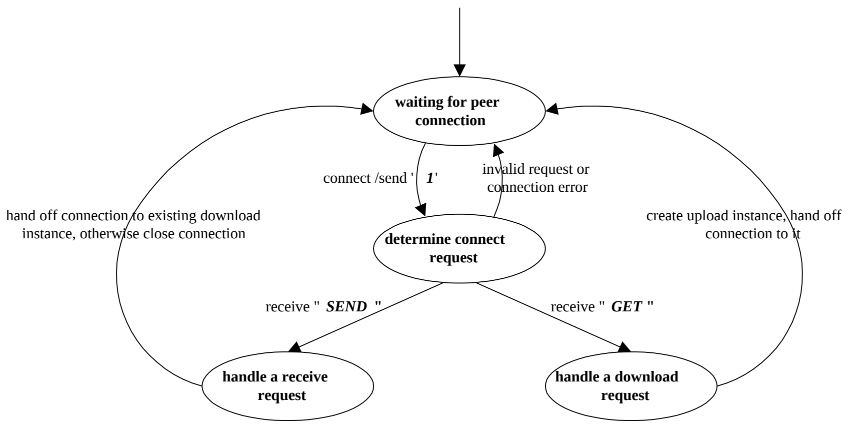

waiting for peer connection
connect /send ' 1'
invalid request or connection error
hand off connection to existing download instance, otherwise close connection
determine connect request
create upload instance, hand off connection to it
receive " SEND "
receive " GET "
handle a receive request
handle a download request

Figure 6: State-chart for the Listener

Initially the Listener begins in the **waiting for peer connection** state. When a remote peer connects to the listening port, the character '***1***' is returned to acknowledge the connection and the Listener transfers to the **determine connect request** state.

In the **determine connect request** state the Listener waits for the message '***SEND***' or '***GET***' from the remote peer. If the message is neither of these or there is a connection error the Listener returns to the **waiting for peer connection** state. Otherwise if the request is a '***SEND***' it enters the **handle a receive request** state. If the request is a '***GET***' request then it enters the **handle a download request** state.

In the **handle a receive request** state, the Listener hands off the connection to an existing Download Instance and returns to the **waiting for peer connection** state. If there is no matching Download Instance for the connection then the connection is closed. There may be no Download Instance available to accept the incoming file if the Download Instance has been terminated or the remote peer is malicious and trying to upload a non-requested file. In the **handle a download request** state a new Upload Instance will be created and the request will be handed off to it.

The state-chart for the Download Instance is given in Figure 7. It has twelve states: **download start**, **download request**, **remote client upload**, **waiting for connection**, **waiting for send**, **remote client download**, **connect to peer**, **response from peer**, **waiting for file**, **file transfer**, **download finish**, and **terminate**.

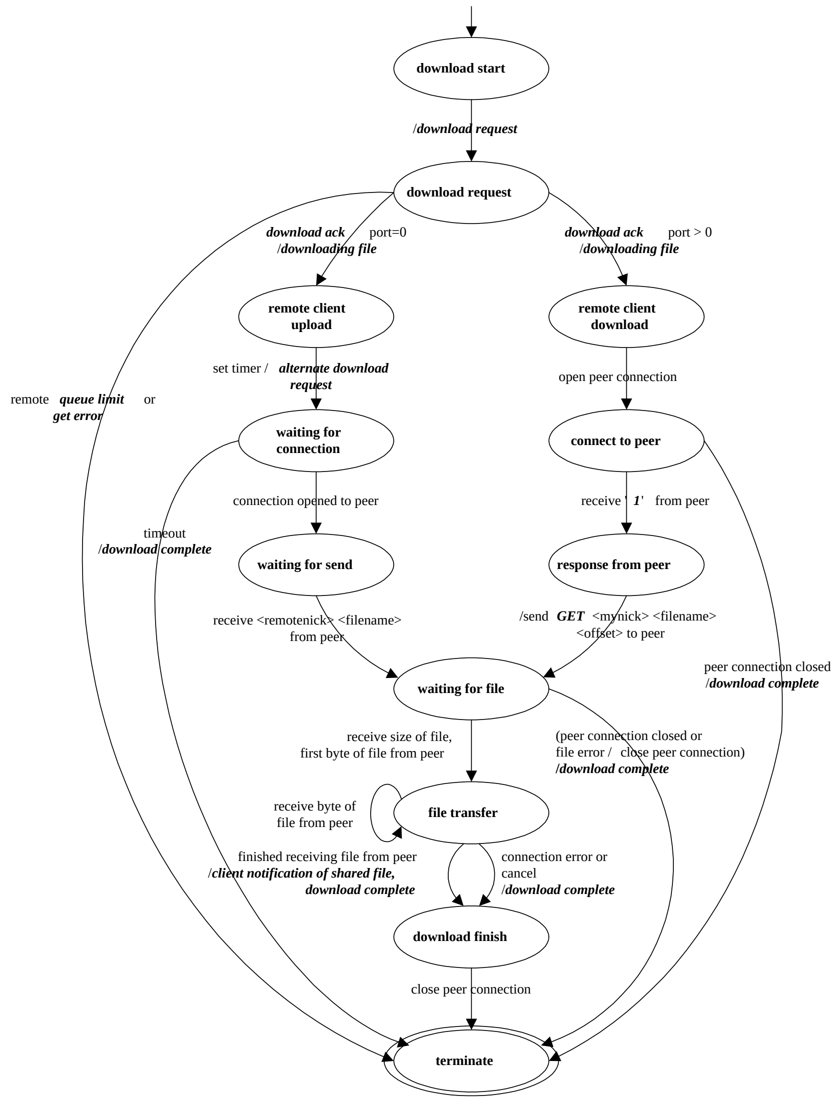

download start
/*download request*
download request
download ack port=0 /*downloading file*
download ack port > 0 /*downloading file*
remote client upload
remote client download
set timer / *alternate download request*
open peer connection
remote *queue limit* or *get error*
waiting for connection
connect to peer
connection opened to peer
receive '*1*' from peer
timeout /*download complete*
waiting for send
response from peer
receive <remotenick> <filename> from peer
/send *GET* <mynick> <filename> <offset> to peer
peer connection closed /*download complete*
waiting for file
receive size of file, first byte of file from peer
(peer connection closed or file error / close peer connection) /*download complete*
receive byte of file from peer
file transfer
finished receiving file from peer /*client notification of shared file, download complete*
connection error or cancel /*download complete*
download finish
close peer connection
terminate

Figure 7: State-chart for Download

Initially the Download Instance starts in the **download start** state. The Download Instance is created when the Napster peer requests a song from a search result to download. The Download Instance first sends the ***download request*** to the broker and then enters the **download request** state. The ***download request*** message contains the name of a song, and the nickname of the remote peer that is hosting the song.

In the **download request** state the Download Instance waits for a response from the broker. If the remote host has reached a maximum number of uploads the broker will respond with a ***queue limit*** message and the state will change to **terminate**. If there is an error with the download request a ***get error*** message will be received and the state will change to **terminate**. The broker will send a ***download ack*** message for a successful download request. The ***download ack*** message contains TCP/IP information on where the song is located. If the TCP port specified in the information is equal to zero then the Download Instance sends a ***downloading file*** message to the broker and enters the **remote client upload** state. Otherwise if the TCP port is greater than zero then the Download Instance sends a ***downloading file*** message to the broker and enters the **remote client download** state. The ***downloading file*** message in the above two state changes notify the Napster broker that a download is taking place.

If the Download Instance entered the **remote client upload** state then the Download Instance must have the file pushed to it. This push is required since the remote peer is behind a firewall. Once the Download Instance sets a timer and sends an ***alternate download request*** message to the Napster broker it enters the **waiting for connection** state. If the timer expires before a remote client connects the Download Instance sends a ***download complete*** message to the broker and enters the **terminate** state. Otherwise a connection is opened from the remote peer, and it enters the **waiting for send** state. The Download Instance will then receive remote information such as the username (remotenick) of the remote peer and song file information causing a change to the **waiting for file** state.

If the Download Instance entered the **remote client download** state from the **download request** state then it can directly connect to the remote peer and download the file. Opening a direct connection to the peer causes the Download Instance to change to the **connect to peer** state. If the remote peer closes the connection, the Download Instance will send a ***download complete*** message to the broker and will change to the **terminate** state. Otherwise, if the Download Instance receives an acknowledgement in the form of the character '***1***' from the remote peer it will change to the **response from peer** state. Next it will send a '***GET***' request to the remote peer and enter the **waiting for file** state. The '***GET***' request contains the peer's username, desired filename, and the offset in the file. The offset in the file will allow for resuming file transfers.

The Download Instance will change from the **waiting for file** state to the **file transfer** state until it receives the size of the file and the first byte from the remote peer. It will stay in this state as it continues to receive file bytes from the remote peer. Once the last byte of the file has been received the Download Instance will send a ***client notification of shared file*** message and a ***download complete*** message to the broker. If there is a connection error or cancel during the file transfer only a ***download complete*** message will be sent to the broker and the Download Instance will enter the **download finish** state.

Once in the **download finish** state the Download Instance will close the remote peer connection. The Download Instance will make a state change to **terminate** and the Download Instance will terminate.

The state-chart for the Upload Instance is given in Figure 8. It has five states: **receiving id**, **setting up transfer**, **file transfer**, **upload finished**, and **terminate**. The Listener will create an Upload Instance when there is a '***GET***' request from a peer.

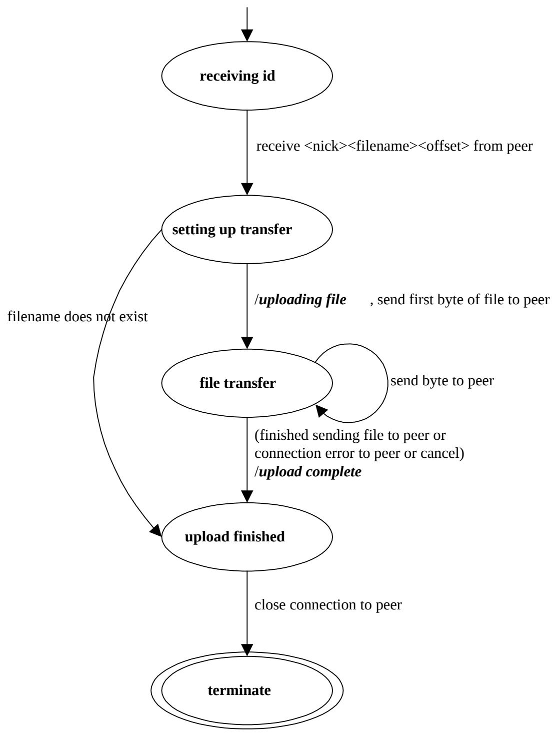

receiving id
receive <nick><filename><offset> from peer
setting up transfer
filename does not exist
/uploading file , send first byte of file to peer
file transfer
send byte to peer
(finished sending file to peer or connection error to peer or cancel) /upload complete
upload finished
close connection to peer
terminate

Figure 8: State-chart for Upload

Initially the Upload Instance starts in the **receiving id** state. After the '***GET***' request the remote peer will send its username (nick), filename, and the offset. This will cause a state change to the **setting up transfer** state. If the requested filename does not exist the state changes to **upload finished**, and the connection as well as the Upload Instance is terminated. If the file does exist an ***uploading file*** message will be sent to the Napster broker, and the first byte of the file will be sent to the peer. The state will change to the **file transfer** state.

The Upload Instance will stay in the **file transfer** state until the file is finished uploading, there is a connection error, or the request has been cancelled. Any of these actions will cause a state change to the **upload finished** state and an ***upload complete*** message will be sent to the Napster broker. The connection is closed and the Upload Instance is terminated when in the **upload finished** state.

The state-chart for the Push Instance is given in Figure 9. It has five states: **connect**

to peer, **connection open**, **uploading file**, **upload finished**, and **terminate**. The Main Coordination entity creates the Push Instance when an ***alternate download ack*** message is received by the peer.

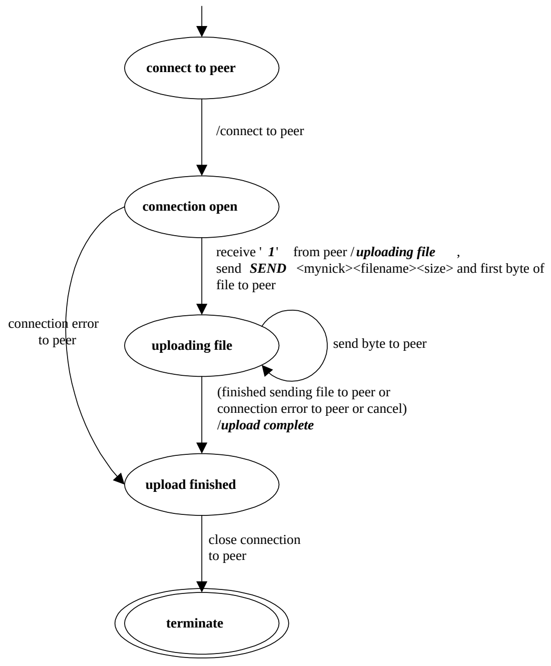

connect to peer
/connect to peer
connection open
receive '1' from peer /uploading file, send SEND <mynick><filename><size> and first byte of file to peer
connection error to peer
uploading file
send byte to peer
(finished sending file to peer or connection error to peer or cancel) /upload complete
upload finished
close connection to peer
terminate

Figure 9: State-chart for Push

Initially the Push Instance starts in the **connect to peer** state. A connection will be made to the remote peer according to the connection information provided in the ***alternate download ack*** message. Next the Push Instance will change state to the **connection open** state.

In the **connection open** state the Push Instance waits for an acknowledgement from the remote peer. If there is a connection error the state changes to **upload finished** state. Otherwise the Push Instance will receive the character '***1***' in acknowledgement for the connection to the remote peer. At this point an uploading file message will be sent to the Napster broker. The request '***SEND***' followed by the local peer's nickname, filename, size, and first byte of the file will be sent to the remote peer. At this point the state will change to the **uploading file** state.

The Push Instance will stay in the **uploading file** state until the file has been sent, a connection error has happened, or the file request has been cancelled. Any of these situations will cause the state to change to the **upload finished** state. In the **upload finished** state the connection to the remote peer is closed and the instance terminated.

To clarify the above state-charts Figure 10 shows the information flow when a Napster peer downloads from a remote peer directly. The left side of the message sequence chart shows the state changes of the Download Instance on the peer. The right side of the chart shows the state changes of the Upload Instance of the remote peer.

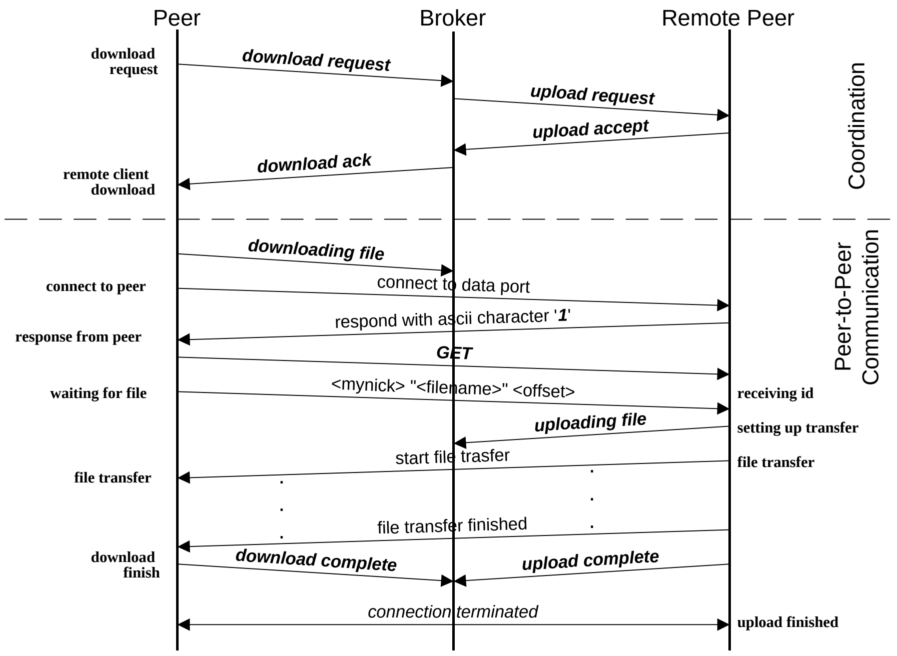


Figure 10: Information flow of a Napster Client Download

The Message sequence chart is divided into two sections. The section above the dotted line shows coordination. The broker coordinates the peer-to-peer download. Instead of the peer directly requesting a desired song from the remote peer, the broker handles the request. This provides added security since the remote peer will be aware that a connection will be made from another peer. For example, if a connection is made to a peer without previous notification from the broker, the connection would be closed.

The section of the message sequence chart below the dotted line shows the peer communication. Once the coordination is complete the peer connects directly to the remote peer and downloads the file. Upload and download notification is provided to the broker by both peers in the form of ***downloading file***, ***uploading file***, ***download complete***, and ***upload complete*** messages.

Figure 11 shows the message sequence chart for a remote client upload. A remote client push will occur when the remote peer is behind a firewall. The left side of the message sequence chart shows the state changes of the Download Instance on the peer. The right side of the chart shows the state changes of the Push Instance of the remote peer.

Similar to Figure 10, this message sequence chart has a coordination section and a peer-to-peer communication section. However, the coordination in the remote upload

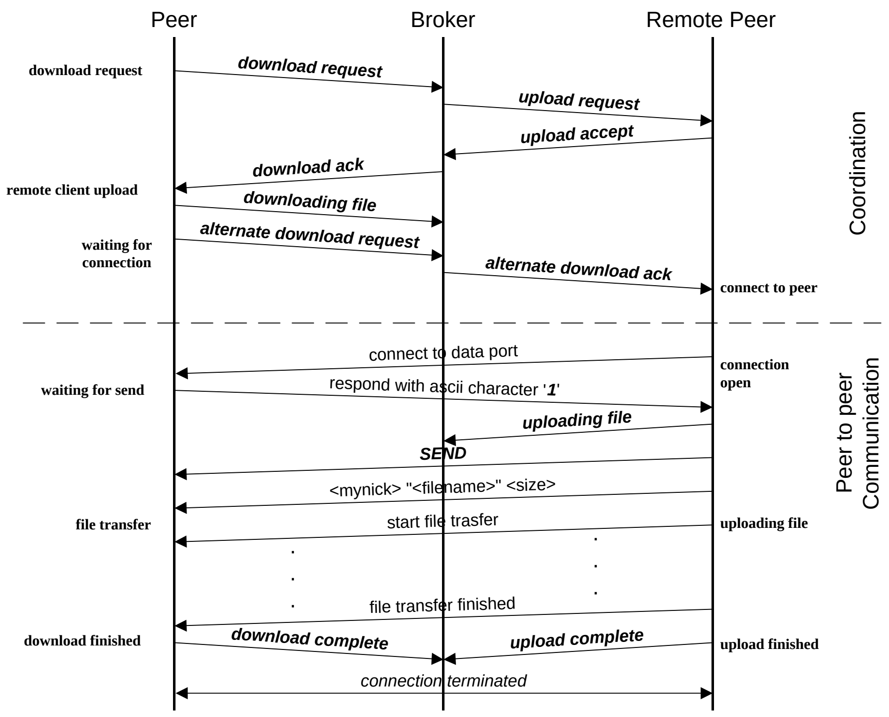


Figure 11: Information flow of a remote client upload

case requires that the peer send an ***alternate download request*** message to the broker. The broker will forward this as an ***alternate download ack*** message to the remote peer to notify it to push the file to the peer.

For the peer-to-peer communication section, the remote peer opens a connection to the peer. Once the peer acknowledges the connection, the file is pushed to the peer. Upload and download notification is passed to the broker from both the peers in the form of ***downloading file***, ***uploading file***, ***download complete***, and ***upload complete*** messages.

2.2.2 Napster Lookup Server and Broker

To use the Napster network a Napster peer connects to both a Napster lookup server and then a Napster broker. A peer uses the Napster lookup server to find the address of an available Napster broker.

The state-chart for the Napster lookup server is given in Figure 12. It has two states: **waiting for client connection**, and **best broker sent**. It initially starts in the **waiting for client connection** state. When a connection is made to the lookup server, it immediately sends the address of a broker to the remote peer and changes to the **best broker sent** state. The broker sent to the peer is the least busy broker on the Napster network. Immediately after sending the broker address, the connection to the remote peer is closed and the lookup server returns to the **waiting for client connection** state.

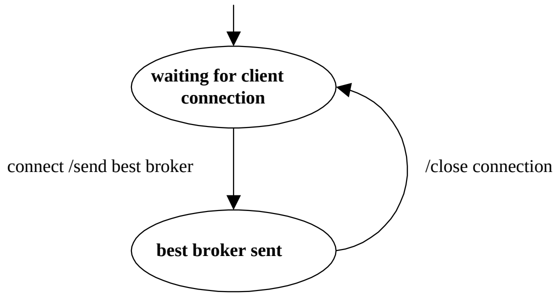


Figure 12: State-chart for Napster Lookup Server

The state-chart for the Napster broker is given in Figure 13. It is composed of three concurrent entities: Client Connection, Broadcaster, and Client Instance. The broker initially starts in the Client Connection entity. On startup the Client Connection entity creates the Broadcaster entity and then enters the **waiting for client connection** state. When a Napster peer connects, the connection is handed off to a new Client Instance and returns to **waiting for client connection** state. The Broadcaster entity periodically broadcasts the ***server stats*** message to all Client Instances. The ***server stats*** message includes the number of users, number of files available, and the total size in gigabytes of the shared files.

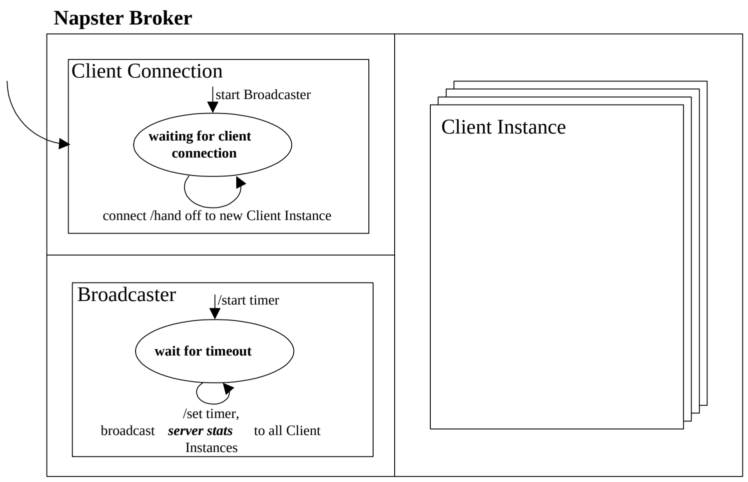

Napster Broker
Client Connection
start Broadcaster
waiting for client connection
connect /hand off to new Client Instance
Broadcaster
/start timer
wait for timeout
/set timer, broadcast server stats to all Client Instances
Client Instance

Figure 13: State-chart for the Napster Broker

The state-chart for the Client Instance is given in Figure 14. It has eight states:

waiting for login, **verify user**, **logged in**, **sending search results**, **download request**, **wait for remote peer status**, **transaction finished**, and **terminate**.

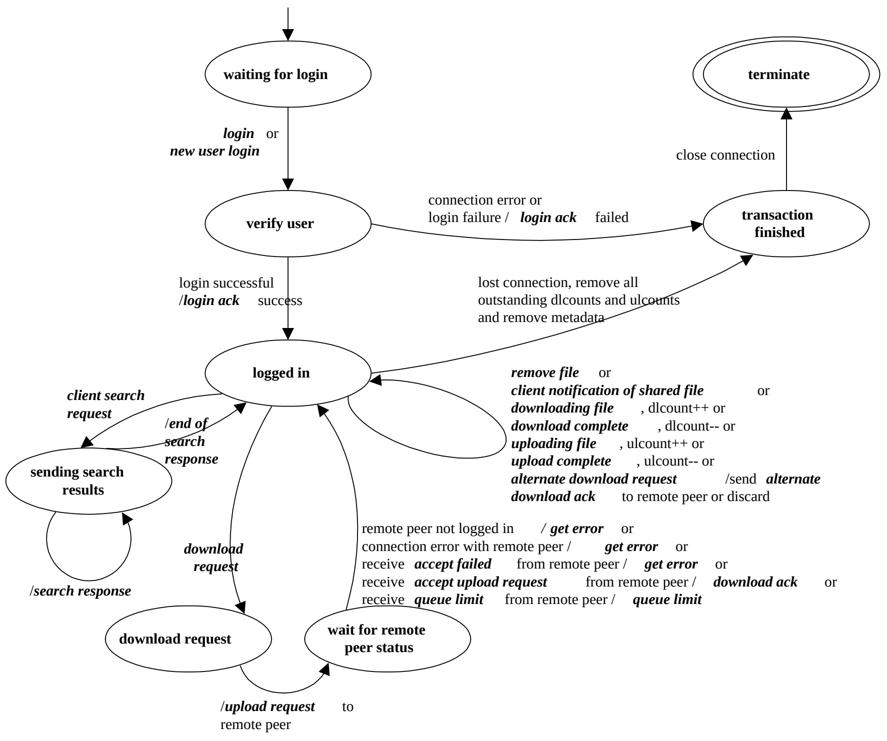

waiting for login
terminate
login or *new user login*
close connection
verify user
connection error or login failure / *login ack* failed
transaction finished
login successful /*login ack* success
lost connection, remove all outstanding dlcounts and ulcounts and remove metadata
logged in
remove file or *client notification of shared file* or *downloading file*, dlcount++ or *download complete*, dlcount-- or *uploading file*, ulcount++ or *upload complete*, ulcount-- or *alternate download request* /send *alternate download ack* to remote peer or discard
client search request
/*end of search response*
sending search results
/*search response*
download request
remote peer not logged in / *get error* or connection error with remote peer / *get error* or receive *accept failed* from remote peer / *get error* or receive *accept upload request* from remote peer / *download ack* or receive *queue limit* from remote peer / *queue limit*
download request
wait for remote peer status
/*upload request* to remote peer

Figure 14: State-chart for the Client Instance

The Client Instance initially starts in the **waiting for login** state. When the Client Instance receives a ***login*** message or ***new user login*** message from the Napster peer it changes to the **verify user** state. If there is a connection error or a login failure a failed ***login ack*** message will be sent to the peer, and the state will change to **transaction finished**. A failed ***login ack*** message will contain an invalid e-mail address. A failed login will occur for a ***login*** message if the username or password is invalid. A failed login will occur for a ***new user login*** message if the username already exists, or if the username and password are of an invalid format. If the login was successful then a successful ***login ack*** message is sent to the peer and the state changes to the **logged in** state. A successful ***login ack*** message will contain the e-mail address of the peer.

The Client Instance may receive a variety of messages from the connected peer while in the **logged in** state. Search messages or download messages will cause the Client Instance to change state. All other messages do not cause a state change in the Client Instance. The ***client notification of shared file*** or ***remove file*** messages will allow the broker to update the metadata database. Receiving a ***downloading file***, ***download complete***, ***uploading file***, or ***upload complete*** message will allow the broker to keep track of the number of concurrent uploads and downloads between peers. The variables *dlcount* and *ulcount* are local to the Client Instance. An ***alternate download request*** message from the peer will simply be forwarded to the remote peer as an ***alternate download ack*** message or discarded if the remote peer is not connected to the broker.

When the client receives a ***client search request*** message it will enter the **sending search results** state. While in this state the broker will send the available song titles matching the search string along with the nicknames of the peers sharing the files. A ***search response*** message will be sent for each song. If the peer is behind a firewall, that peer will only receive matching songs by peers not behind firewalls. A limit by the broker is placed on the number of song titles that it will send to the peer. When the last ***search response*** message is sent, the Client Instance will send an ***end of search response*** message to the peer and return to the **logged in** state.

A ***download request*** message from the peer will cause the Client Instance to change state to the **download request** state. While in this state an ***upload request*** message will be sent to the remote peer hosting the desired song file to get permission for the peer to download the file. The state will then change to the **wait for remote status**. If the remote peer is not logged in anymore, there is a connection error to the remote peer, or an ***accept failed*** message is received from the remote peer then a ***get error*** message will be sent to the peer. If the remote peer sends an ***accept upload request*** message then a ***download ack*** message will be sent to the peer. The ***download ack*** message will contain TCP/IP connection information. If the remote peer is behind a firewall the TCP port specified in the ***download ack*** message will be set to 0, otherwise the TCP port will be a valid number. If the remote peer responds with a ***queue limit*** message then a ***queue limit*** message will be sent to the peer. Sending any information regarding the download request will cause the state to change back to the original **logged in** state.

When the peer disconnects or the connection is lost to the peer then all outstanding download counts, upload counts, and song file metadata will be removed from the broker. The Client Instance will then enter the **transaction finished** state. From this state the peer connection will be closed and the Client Instance will be terminated.

3 Gnutella

Gnutella is a decentralized network and has no central server used for coordination. Once a Gnutella host is connected to a Gnutella network they act as a coordination node that is free to exchange files with other hosts.

3.1 Overview

A Gnutella host first joins a Gnutella network by connecting to at least one other host on the network. To obtain files a client must query the network to find out where files are located. In Figure 15, Computer A sends a ***search*** message to all of its connected computers. This message is propagated to other connected computers. A time to live (TTL) field in a query message will ensure that a message does not last forever. The furthest node possible where the query message dies is termed as the search horizon.

When a computer contains files that match the ***search*** message it sends a ***search results*** message back along the path the request came. Once the search results are returned, the client can directly connect and download from a peer carrying a desired file. For example in Figure 15, Computer A downloads a file directly from Computer E.

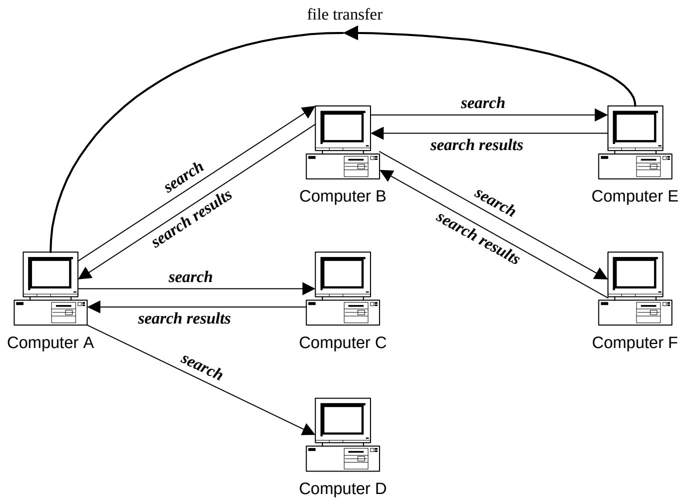


Figure 15: A Gnutella Network

There are five messages defined for coordination and discovery of files between Gnutella Peers: ***ping***, ***ping response***, ***client ping request***, ***search***, and ***search results***. All connections between Gnutella peers use the TCP/IP protocol. A small subset of HTTP constructs will be used for the initialization of peer connections. These include the messages ***CONNECT***, ***OK***, ***GET***, and ***GIV***. In discussion of the protocol all messages are shown in ***bold and italic***. The description of each message and the information they carry are explained on a web page named *Gnutella Protocol Specification* [2].

3.2 The Protocol

There are up to four concurrent entities in a Gnutella peer shown in Figure 16: Connection Handler, Coordination Instance, Download Instance, and Upload Instance. The Connection Handler entity manages all the incoming and outgoing connections made to other Gnutella peers. Each Coordination Instance handles a coordination connection to another Gnutella peer. A coordination connection is any connection to another Gnutella peer that is not an upload or a download connection. The Download and Upload Instances handle a download and upload connection respectively. There may be zero or more of the Coordination, Upload, and Download Instances running at one time.

The state chart for the Connection Handler entity is given in Figure 17. It has

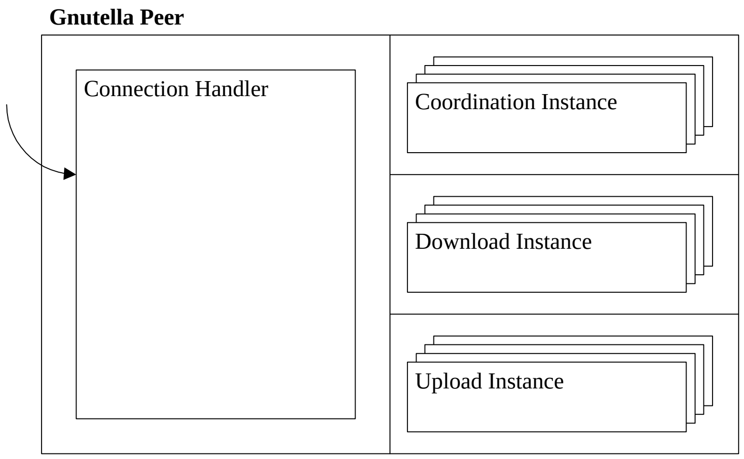


Figure 16: State-chart for Gnutella Peer

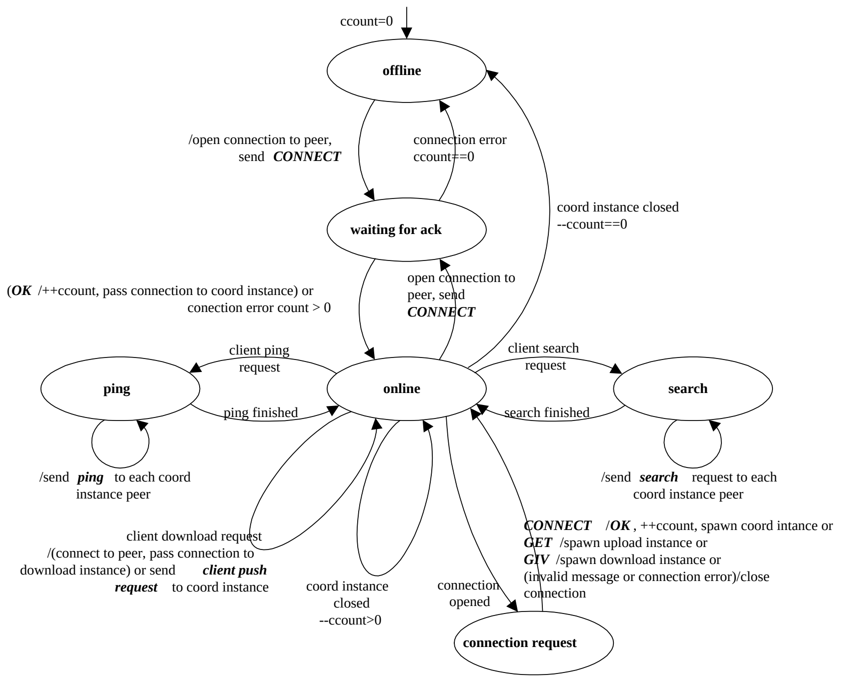
six states: **offline**, **waiting for ack**, **ping**, **online**, **search**, and **connection request**. The Gnutella peer is online when it has one or more coordination connections to other Gnutella peers. The *ccount* variable keeps the current count of coordination connections.

Initially the Connection Handler Entity begins in an **offline** state with the *ccount* variable set to zero. It changes to the **waiting for ack** state by opening a coordination connection to a remote peer and sending a ***CONNECT*** message. Since the *ccount* variable is equal to zero a connection error to the new remote peer will cause the state to return to **offline**. If the new remote peer acknowledges the connection with an ***OK*** message, the connection to the remote peer is passed to a new Coordination Instance, the *ccount* variable is incremented, and the state will change to the **online** state. Further connection coordination requests to new remote peers will cause a state change back to the **waiting for ack** state. However, when a connection error to the new remote peer occurs, the *ccount* variable will be greater than zero and the state will return to the **online** state.

While in the **online** state the client may ping the Gnutella network to determine the size of the network. A client ping request will cause a state change to the **ping** state. A ***ping*** message will be sent to each remote peer defined in the Coordination Instances. Once this is finished the state will return to the **online** state.

A client search request will cause a state change to the **search** state. A ***search request*** message will be send to each remote peer defined in the Coordination Instances. Once this is finished the state will return to the **online** state.

The Gnutella peer may choose to download one of the files listed in the ***search result*** messages from all Coordination Instances. A ***search result*** message will contain information relating to the file and the TCP/IP connection information of the remote peer that is hosting the file. If the remote peer hosting the file is not behind a firewall a connection will be made to that peer and the connection will be passed to a new



ccount=0
offline
/open connection to peer, send CONNECT
connection error ccount==0
waiting for ack
coord instance closed --ccount==0
(OK /++ccount, pass connection to coord instance) or conection error count > 0
open connection to peer, send CONNECT
client ping request
client search request
ping
online
search
ping finished
search finished
/send ping to each coord instance peer
/send search request to each coord instance peer
CONNECT /OK, ++ccount, spawn coord intance or
GET /spawn upload instance or
GIV /spawn download instance or
(invalid message or connection error)/close connection
client download request /(connect to peer, pass connection to download instance) or send client push request to coord instance
coord instance closed --ccount>0
connection opened
connection request

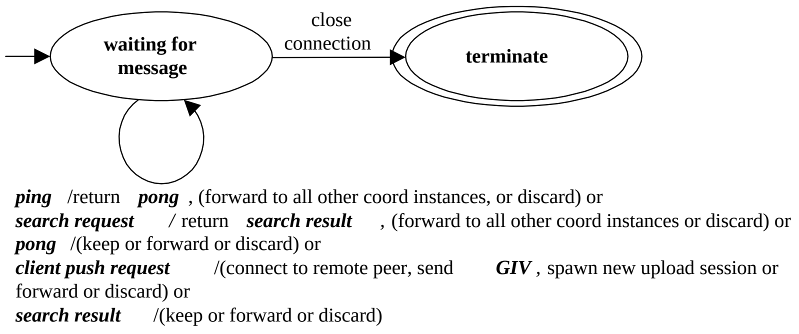
Figure 17: State-chart for Connection Handler

Download Instance. Otherwise if the remote peer is behind a firewall then the Gnutella peer will send a ***client push request*** message through the coordination connection of the remote peer. If the Gnutella peer is behind a firewall then it may only download files from remote peers that are not behind firewalls.

The termination of a Coordination Instance will cause the *ccount* variable to be decremented. If the new value of *ccount* variable is greater than zero then the Connection Handler entity stays in the **online** state, otherwise it changes to an **offline** state.

When a remote peer opens a connection the state changes to a **connection request** state. If the new remote peer sends a ***CONNECT*** message, an ***OK*** message is returned to acknowledge the connection, the ccount variable is incremented, and the connection is passed to a Coordination Instance. If the new remote peer sends a ***GET*** message then the connection is passed to a new Upload Instance. If the new remote peer sends a ***GIV*** message then the connection is passed to a new Download Instance. If the new remote peer sends an invalid message or there is a connection error the connection to the remote peer is closed. Any of the above actions following a new connection from a remote peer will cause the state to return to the **online** state.

The state chart for a Coordination Instance is given in Figure 18. It has two states: **waiting for message** and **terminate**. A Coordination Instance handles a coordination connection to a remote peer.



waiting for message
close connection
terminate
ping /return *pong* , (forward to all other coord instances, or discard) or
search request / return *search result* , (forward to all other coord instances or discard) or
pong /(keep or forward or discard) or
client push request /(connect to remote peer, send *GIV* , spawn new upload session or forward or discard) or
search result /(keep or forward or discard)

Figure 18: State-chart for Coordination Instance

Initially the Coordination starts in the **waiting for message** state. The Coordination Instance may receive five messages: ***ping***, ***search***, ***pong***, ***client push request***, and ***search result***. After receiving a ***ping*** message, the client will respond to the forwarding peer with a ***pong*** message. If the TTL field of the ***ping*** message is greater than zero, the TTL field will be decremented and the ***ping*** message will be passed to each of the other remote peer connections defined by the Coordination Instances. The global id, and the forwarding peer of the ***ping*** message will be recorded in order to aid in the routing of returned ***pong*** messages. If the message received is a ***search*** message the Gnutella peer may respond to the forwarding peer with a ***search result*** message. If the Gnutella peer contains any files matching the search string defined in the ***search*** message then a ***search result*** message will be sent. If the TTL field of the ***search*** message is greater than zero, the TTL field will be decremented and the ***search*** message will be passed to each of the other remote peer connections defined by the Coordination Instances. The global id, and the forwarding peer of the ***search*** message will be recorded in order to aid in the routing of returned ***search result*** messages. If the message received is a ***pong***, ***client push request***, or a ***search result*** message and it is destined to the Gnutella peer the message will be kept. A ***pong*** message will be used to update the Gnutella peer's statistics on the Gnutella network. For the ***client push request*** message a new connection will be made to the remote peer, a ***GIV*** message will be sent, and the connection will be handed to a new Upload Instance. A ***search result*** message will be used to update the Gnutella peer's overall search results. If the ***pong***, ***client push request***, or ***search result*** message is destined for the Gnutella peer and the global ID in the message is not recognized, or the TTL field on the message is zero it will be discarded. Otherwise the TTL field will be decremented and the message wi ll be forwarded along the original path that it came.

When either the Gnutella peer or the remote peer closes the coordinating connection the Coordination Instance is terminated.

The state-chart for the Download Instance is given in Figure 19. It has five states: **connect to peer**, **wait for file**, **download file**, **download finished**, and **terminate**. The Download Instance is created when the Gnutella peer requests a download from a remote peer or a file is pushed from a remote peer. The connection to the remote peer

Figure 19: State-chart for Download Instance

Initially the Download Instance starts in the **connect to peer** state. To start the download the peer sends a ***GET*** message containing the desired filename to download. It then changes state to the **wait for file** state.

A connection error to the remote peer will cause a state change from the **wait for file** state to the **download finished** state. Otherwise the peer will receive an HTTP file header and the first byte of the file. This will cause a state change to the **download file** state.

The Download Instance will remain in the **download file** state as long as it continues to receive bytes of the file. Once the file has been successfully received or there is

Figure 20: State-chart for Upload Instance

Initially the Upload Instance starts in the **send HTTP header** state. In response to the ***GET*** request of the remote peer an HTTP content header, and the first byte of the file will be sent to cause a state change to **uploading file**. Bytes will be sent to the remote peer until the file has successfully uploaded or there is a connection error. At this point the state will change to the **upload finished** state. The connection will be closed and the Upload Instance terminated.

To clarify the above state-charts Figure 21 shows the information flow of a Gnutella connection, search, and download. The connections between computers A, B, and E

Figure 21: Information flow of a Gnutella Search and Download

The message sequence chart is divided into four sections. The top section shows the Connection Handler entity of Computer A creating a connection to Computer B in the first section and starting a search. The next section of the chart shows how a search propagates through the Gnutella Network. The Coordination Instance on Computer A and the Coordination Instance on Computer E handle the search request. The Connection Handler on both Computer A and Computer E handle the new file exchange connection. The fourth section shows the exchange of a file. The Download Instance

Figure 22: Information flow of a Gnutella Push

4 Discussion

Peer-to-peer networks such as Napster, and Gnutella have been compared to presence management of instant messaging [6]. Instead of querying if a certain user is online, peer-to-peer clients query if a file is online. The peer-to-peer networks provide a method to discover the location of a file. Both Napster and Gnutella provided up-to-date search results that describe the location of files at any one time. Once a peer leaves either a Napster network or a Gnutella network all the files they are serving become unavailable and will not show up in any search results of other peers.

Napster and Gnutella differ on resiliency. Some resiliency was added to Napster by having multiple lookup servers and brokers to handle coordination and discovery. However all these lookup servers and brokers are situated in the same geographical location. This presents a single point of failure. If the network link to these Napster servers fails, the whole network stops functioning. Gnutella has no centralized characteristics. The coordination and discovery functions are shared equally among all the peers in the network. The network will still continue to function even if a number of Gnutella peers fail.

Napster and Gnutella distribute the load of file serving to the peers. Distributing this workload over many computers should help produce a more scalable network. However, scalability is a concern for Gnutella. The propagation of a ***search*** or a ***ping*** message throughout a Gnutella network will create a large number of messages. For example, if a client sends a ***search*** request message with a TTL of seven to four hosts and each of those four hosts sends a message to four other hosts and this continues until the TTL reaches zero then 21845 messages[3] are created for one ***search*** message. A similar amount of traffic will be produced by a ***ping*** message. In contrast a coordinating central broker of the Napster Network will produce very few messages. For example, a search from a peer in a Napster Network will produce one ***search request*** message to the broker.

Even though Napster requires few messages for coordination and discovery it is important to remember that a peer login is expensive. When a peer first connects to the broker the peer must register each of the songs it is hosting. Figure 23 shows a chart[4] comparing the total number of messages produced in a Napster network as opposed to a Gnutella network for the first 500 search messages. The total number of messages produced on the Napster network is greater than the Gnutella network until

[3]We obtain 21845 by using the geometric series $\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{4-1} = \frac{4^{7+1}+1}{4-1} = 21845.$

[4]This chart assumes that a single broker Napster network and a Gnutella network each contain 21845 users. The Gnutella network is structured such that there is a root node with 4 connected peers and each connected peer has 4 hosts and this continues to a depth of 7. The average number of songs hosted by each client is 150; this average number of hosted songs was taken from the data gathered during the week of November 18th (Section 2.1). A Napster login will produce 152 messages: 2 messages for ***login*** and ***login ack***, and 150 messages for each song notifi cation. Assuming a maximum of 100 songs is returned from a broker a total search on Napster will produce 101 messages: 1 message for the ***client search request*** message, and 100 messages for each of the ***search response*** messages and 1 message for the ***end of search response*** message. A connection to another peer in Gnutella requires 2 messages: 1 message for ***CONNECT*** and the other for ***OK***. A search message in Gnutella will produce 22195 messages: 21845 ***search request*** messages produced, and 350 messages for the ***search result*** messages. This chart assumes that 100 search results are returned to a peer in a Gnutella network at an average depth in the network at TTL/2. In this example the TTL is 7.

Figure 23: The message load on a Napster network and Gnutella network for the first 500 messages.

Napster has increased the scale of their network to handle more peers by increasing the available number of lookup servers and brokers. With the efficiency of the coordination and discovery in the Napster network as well as the ability to add more lookup servers and brokers it appears that a Napster network can scale very well. However, as discussed earlier in Section 2.1, Napster has been unable to keep the brokers linked. This is most likely due to the fact that communication between the brokers in a Napster network causes a considerable amount of traffic. For example, if there are one hundred Napster brokers, a ***search request*** message from a peer will cause the peer's broker to create a ***search request*** message for each of the other brokers. This will create ninety-nine more messages.

Another limitation to Gnutella is that both ***search*** and ***ping*** messages in the Gnutella network will only reach as far as the TTL field of the message will allow. In contrast a broker will return search results based on all the metadata from all the peers in the Napster network.

Napster and Gnutella provide domain name and geographical independence for the peers. Domain name independence presents an advantage such that the confusion created by hosts that are contained behind firewalls and hosts' dynamic Internet addresses is easily bypassed. Geographical independence provides a disadvantage such that peers do not necessarily exchange files with other peers that are geographically close. This will produce poorer performance for the exchange of files between peers in a Napster

Figure 24: Coordination and Discovery Clusters located throughout the World.

[5]The location of the Napster servers was discovered using Visual Route 5.0b.

Each cluster would contain multiple lookup servers, brokers, and connector servers. This redundancy at the cluster level would add resiliency to the cluster since any one broker, lookup server, or connector could fail without any harm to the functioning of the cluster. Since there are multiple clusters there is resiliency in the overall network. If a cluster fails then other clusters would be available to coordinate peers.

Each of the clusters would be linked in a manner similar to Gnutella via the connector servers. A cluster would communicate with other clusters if it would be unable to satisfy a peer's coordination or discovery request. Upon the receipt of a search request a connector server would determine if its brokers contain metadata matching the search request. If it could only partially fulfill the request then it would return the matching search results it does have and forward on the search request to another cluster. This would continue to happen until the request is fulfilled or when no more clusters are available.

To provide the best performance peers would connect to the closest cluster. The closest cluster would be the one with the smallest network latency. By localizing peers to clusters they would have a higher probability of exchanging files with the closest possible peers.

6 Conclusion

Napster and Gnutella each present a unique peer-to-peer model. Napster presents a model where discovery and coordination of files among peers are centralized, but the exchange of files takes place between peers. Gnutella removes the centralization and extends the model further by requiring the peers to contribute to the coordination and discovery efforts. Napster is very scalable but its resiliency is reduced since discovery and coordination are centralized. In contrast Gnutella is resilient since there are no centralized components, but it is not scalable since the structure of Gnutella produces an exponential number of messages.

References

[1] Knighten, Bob. *Peer to Peer Networking.* “http://www.peer-to-peerwg.org/specs_docs/collateral/PtP_IDF_Rev1.11.pd”. Aug 24, 2000.

[2] Mayland, B. *Gnutella Protocol.* “http://capnbry.dyndns.org/gnutella/protocol.php”. Date Viewed: November 21, 2000

[3] *Napster Messages.*“http://opennap.sourceforge.net/napster.txt”. Last Updated: August 6, 2000. Date Viewed: October 6, 2000.

[4] *Napster Service Status.* “http://www.napster.com/status/”. Last Updated: October 6, 2000. Date Viewed: October 6, 2000.

[5] Nilsson, Erik. *Napster: Popular Program Raises Devilish Issues.* “http://www.oreillynet.com/pub/a/network/2000/05/12/magazine/napster.html”. May 12, 2000.

[6] Sims, Dave, Tim O’Reilly, and Jon Udell. *O’Reilly’s Peer-to-Peer Summit Synopsis: Transcript.* “http://www.oreillynet.com/pub/a/linux/2000/09/22/rt-transcript.html”. September 22, 2000.