Motivation

Napster vs. Gnutella

There are essentially two extremes to the current location-independent non-life critical file-sharing tools available over the internet: The “Central Server” Napster approach and the “Ultra-Distributed State” Gnutella/FreeNet approach…

**Napster:** Napster makes use of a central server which returns query results to users by accessing a private database. The central server allows their model to be fast and for the most part reliable. However, its central server costs money to operate and they are constantly at the risk of being blocked out by some institution or shut down by the Government.

**Gnutella:** Gnutella is on the other extreme. Unlike Napster, their model doesn’t have a central server and as a result it is much harder to filter or shut down. However, Gnutella is very slow. Users complain about frequent disconnects and slow searches

**A balance between the two extremes:**

Both Napster and Gnutella have their advantages and disadvantages. We hope to find a solution that is more structured than Gnutella yet does not depend on dedicated resources like Napster. By doing so, we expect to achieve the benefit of both spectrums.
Proposed Solution

Project Quaero is a location-independent file-sharing tool targeting non-life critical data such as mp3s and other forms of media. The Quaero network maintains a dynamically changing hierarchical structure where computers are qualified to perform particular tasks based on certain heuristics such as bandwidth, hard drive, average login time, the files that they store, etc. Quaero does not require dedicated resources, so regardless of what tasks a user may have, they will be able to leave the network without causing any harm.

Features:

From the end-user’s point of view, our Quaero will have the following features:

- **Search** – User’s should be able to search for files, and view the results of their search. This does not guarantee that all files that match the search will be returned or even a majority of them. However, since we are assuming non-life critical data the user shouldn’t care too much.

- **File Transfer** – Once the user is given the search results, they can request file transfer from other users who have files they want.

- **Ease of use** – Our program will be extremely easy to use, much like napster is

- **User’s aren’t overburdened** – Regardless of what role a person may play in the topology of our network, a user should never feel a significant performance in their computers or their bandwidth

- **Platform independence** – User’s will be able to run the application under environments that support Java and Swing UI.

Assumptions:

In order to produce a relatively simple and efficient implementation of this distributed system, we are building the application under the following assumptions:

1) **Non-life critical data and lots of duplicated data:**
   
   Our search engine is based on delivering results for non-life critical data (mp3s). As a result, we don’t have to return all possible results or scan the entire network since duplicated data will more than likely be located in many subsets of the network. Even if our searching algorithm is flawed, as long as it works most of the time it will be acceptable. As a result of this, searches will be faster and each search will contribute less traffic to the network than otherwise
2) **There is a set of computers that are connected to the network for long durations of time**

Obviously not every computer in the network will be a super-computer, but we are basing our implementation on the fact that at there will be at least some computers that are on for several hours at a time (dedicated connections, etc). This will help us to establish a network topology which undergoes very frequent reconfigurations.

**Architecture:**

Quero is a distributed, hierarchical network with 3 basic roles.

**Different roles in the network hierarchy:**

**Leaf Nodes:** This is the simplest node in the network. It is not involved in receiving search queries from anybody. Their tasks are limited to being able to send out search queries, and send/receive files with other nodes.

**Master Browsers:** Master browsers have all the requirements of a Leaf Node, with their master browser being their parent. In addition to the responsibilities of a leaf the master browser is responsible with doing three things, holding a search directory of their children files, splitting / holding elections, and caching queries.

**Top-Level Master Browsers:** Top-level master browsers are master browsers with no parents. These master browsers are required to implement all the features of the regular master browser along with the features of the top-level master browser. A node acting as a top-level master browser is required to do a few extra things. It must keep a list of all the other top-level master browsers in the network, and it must send all these top-level master browsers messages with some frequency. Further if asked for a list of the entire top-level master browsers in the network the master browser must provided one. Finally the search propagation for this master browser is a little different. If the top-level master browser doesn’t know enough answers for a query it will do one of two things. If the general network traffic is low the master browser will send the query on to all other top-level master browsers in the network. If the network traffic is high then the top-level master browser will send the query on to a random sample of the other top-level master browsers. Remember when top-level master browsers receive queries from other top-level master browsers they do not search there caches.

**Searching the Network:**

Network searches will be conducted in clear text, and will initially just be queries of file names and potentially MP3 Id3 tags. To search the network, a client will first search its local database to determine if it knows of any machines satisfying the query. If the client knows enough machines at this point the query stops, most likely the client won’t. In this case it will randomly select one of its master browsers. It will pass its query on to this machine. The master browser receiving the query will then search using the following
system. The master browser will search its local directory like the node; if the master browser finds enough results it will return the results to the node that queried it. If the master browser doesn’t know enough results it will ask its master browser. On receiving the results of a propagated query the master browser will cache any results it doesn’t already know and pass the results back down to the node that called it.

**Problems and Possible Solutions:**

**Flooding the master browsers:**
Master browsers can receive a large number of search requests at any moment. To avoid excessive load on the master browsers and enhance network performance:

1. **Limit on number of children**
   Each master browser will be a browser for no more than a certain number of machines. If more machines connect to a master browser the master browser will pick out extra browsers and split its children with the new browsers.

2. **Limit on number files**
   To keep the master browsers from being required to index too much data the master browsers will impose a strict limit on the number of nodes which they let connect to them, in that they will let the total number of files shared by any one master browser contain no more than a certain number of files. If the total number of files shared under a master browser grows too big, the master browser can eject one of its nodes, turning that node into a top-level master browser. <see top level master browsers>

3. **Bandwidth splitting**
   If a machine, which currently is the master browser, is getting hammered but has fewer than maximum number of nodes connected to it. Then it can perform a bandwidth split. To perform a bandwidth split the master browser performs an election among its nodes. If there is a single node that is significantly better than the master browser, the master browser and that node will swap position, if there isn’t a
significantly better node, the next best node will be upgraded to a co-master browser (see diagram above). The nodes in the network will then randomly select a master browser to talk to, and the parent of the master browser will randomly choose between the two master browsers. For the parent of the master browser the old master browser and the co master browser only count as one node, when counting the number of connections to the parent’s master browser. The new node or the upgraded node will must learn about its children’s search directories before it can become operational, it will obviously start out with empty cache.

![Diagram of bandwidth splitting before and after split]

4. Caching
The master browser will cache results of queries. The idea is that if a master browser is being queried by one of its children and the master browser doesn’t know the answer to the query the master browser will propagate the query <see searching> on receiving the answer of the query the master browser will cache the answer. Then future queries can be tested against this cached result, and hopefully reduce the need for propagation of the query. To keep the cache from getting too large, cached data will expire at some undetermined rate and have a fixed maximum size. The rate of expiration will most likely start as a constant, but might be expanded to be an advertised rate much in the way cached DNS names expire in DNS servers.
**Unkowns:**

There are still a few unknown aspects of the project that won’t be defined until we are able to run simulations (see Plan for Evaluation). Most of these unknowns deal with fine-tuning heuristics, such as:

- Will bandwidth splitting and Caching be enough, even with the right heuristics, to effectively reduce burden on a node when the network gets very large?
- What is an acceptable branch factor so that master browsers aren’t too overburdened?
- How much data can a user manage at one time? Obviously a user’s capabilities vary, but defining appropriate heuristics to make such a decision will be very important.
- When should a Master Browser add to its query by asking its own Master Browser?
- When should a Master Browser cache results?
- What are the heuristics for determining which computers are best for which roles?
- What level should Master Browsers stop propagating the file list (i.e. become Top-Level Master Browsers)?

We hope to address these unknowns as we find out more information from running simulations.
Plan for Evaluation

Defining the Environment:

Our evaluation is going to be based on the fact that we are running in an environment that is similar to Napster and Gnutella:

- Lots of duplicated data
- Similar file requests (lots of people searching for Metallica, for example)
- Some users log on for a while
- User’s have varying resources
- Large user base

Key Features to Establish:

- **A Distributed state where no node ever gets too burdened:**
  We want to establish that even when the # of nodes becomes relatively large, no node is overburdened with too many file requests, too many requests, etc…
- **Relatively fast:**
  The # of hops should be less than Gnutella like searches and in some way it should be comparable to Napster
- **Search success rate is within a moderate range:**
  Our network does not guarantee finding all search results but it should be sufficient for use.

Defining a Napster/Gnutella Sample Userbase:

We will have a simulator which can run a sample userbase similar to that of Napster or Gnutella. Defining exactly what is a good sample userbase to run will involve defining in detail our underlining assumptions, such as:

- How much is data duplicated?
- How many users are using T1s vs. 14.4 modems, etc?
- How many search requests do users make per day and when?

We hope to obtain answers to these questions from Napster and Gnutella studies, and as a last resort some educated guesses.

Running the Sample Userbase:

Once we have came up with a userbase, we will be using it to evaluate whether our network has struck a balance between the Napster/Gnutella extreme. We’ll also be using this userbase for fine tuning heuristics, testing, etc…
Plan for Implementation

Components:

- **Core Controller**
- **UI Controller**
- **Transfer Controller**
- **Search Index Controller**
- **Login Controller**
- **Master Browser Controller**
- **Client Controller**
- **Heuristics Controller**
- **Employment Agent**

**Transfer Controller** --
Responds to file transfer requests (send/recv)

**Search Index Controller** --
Stores file search lists and handles caching (add/remove/update/find/cache)

**UI Controller** --
Deals with displaying information (data and heuristics), accepting user queries, file requests, etc…

**Login Controller** --
Deals with login logic. As it stands now, the login controller maintains a frequently logged on user list of which the login controller can contact

**Master Browser Controller** --
Maintains a list of one or more master browsers (in the case of bandwidth splitting), and maintains the list as necessary. Responsible for handling and dispatching messages to the appropriate master browser

**Client Controller** --
Responsible for maintaining a list of clients and for handling and dispatching messages.

**Heuristics Controller** --
Maintains Heuristics information such as average login time, bandwidth, file request rate and any other information that would be relevant during elections

**Core Controller** --
Ties all the components together, dispatches messages to appropriate components, etc.

**Employment Agent** --
A set of computers will run an employment agent that maintains a list of available workers and finds an appropriate computer for a particular task upon request. This may include finding a master browser for a computer logging in, or finding a replacement for a master browser, etc. This agent can then be queried when positions need to be filled
Project Milestone:

**Monday, Jan 22\textsuperscript{nd}**
- Basic Napster-like UI with a login interface, and a search, file transfer, and heuristics tab as well as a dialog box for preferences
- Filename parsing into keywords
- Write FileScanner to browse directories and find files of appropriate type

**Monday, Jan 23\textsuperscript{th}**
- Complete design report

**Monday, Jan 29\textsuperscript{th}**
- Basic login to a single static master browser, and hardcoded file transfer
- Basic searching with just the static master browser and maybe a few clients

**Monday, Feb 5\textsuperscript{th}**
- The Heuristics controller should be computing the most relevant data needed for employment
- More sophisticated login where you are assigned a non-static master browser from the employment agent

**Monday, Feb 13\textsuperscript{th}**
- Bandwidth splitting implementation
- Setting up dedicated login list pages
- Primitive Network Simulator and some sample simulation data

**Monday, Feb 13\textsuperscript{th} to Monday, Feb 27\textsuperscript{th}**
- Improving employment agent algorithms such as bandwidth splitting, dealing with logouts (ungraceful and graceful), employment assignments, etc…
- Caching
- Expanding on Network Simulator and sample simulation data for testing phase

**Monday, Feb 27\textsuperscript{th} to Monday, Mar 13\textsuperscript{th}**
- Intensive testing, time shift, and minor improvements
Summary

Project Quaero is a location-independent file-sharing tool. The targeted-data is non-life critical data such as media files (mp3, movie, etc.). This project is an attempt to balance the two current technologies (Napster and Gnutella), taking advantages of both technologies with minimal trade-off. The goal is to develop a hierarchy that is more structured than Gnutella yet does not require any dedicated machines like Napster does.

From end-point user’s view, Quaero will have key features such as searching, file-transferring, ease of use, reasonable burden among users, and platform independence. Each computer running Quaero is a node in the Quaero’s tree structure. A node can be a leaf, a master browser, or a top-level master browser. As discussed in previous sections, each node has different roles (depending on the type of the node) such as splitting/holding elections, caching queries, searching for files, etc.

Quaero has some mechanisms to enhance the network performance. Those mechanisms include limiting the number of children, limiting the number of files, bandwidth splitting, and caching.

To evaluate this project, we are going to compare the scalability (measured by the load on each node) and the performance (measured by the number of hops to do a search) of Quaero with Napster and Gnutella. Also, we will implement a simulator to collect the metrics (scalability and performance) from each node.