1 Project Overview

Our goal was to create a distributed file-sharing system based on the Gnutella protocol, version 0.4. It is based on a prompt from a CS class at `http://www.cs.iit.edu/~iraicu/teaching/CS550-S11/pa2.pdf`

The goal is to have a P2P system that can handle multiple concurrent connections, propagate requests for files through a network even though each peer only knows its immediate neighbors, and successfully send the location of the requested files, if they exist in the network, to the original peer.

A full version of the Gnutella 0.4 protocol can be found at `http://web.stanford.edu/class/cs244b/gnutella_protocol_0.4.pdf`

We implemented a simplifed version of the Gnutella protocol that does not do neighbor discovery. Instead, the setup of peers and their neighbors is static and remains the same the entire time.

2 Design & Implementation

In our distributed file sharing system, each servent has a list of neighboring servents, and knows the IP addresses of these neighbors only. When a servent requests a file, it sends a `QUERY` for that file to all of its neighbors. If these neighbors have the file, they send a `HITQUERY` message to the requester. If they don't, they forward the `QUERY` to all of their neighbors. This way the request propagates through the network, asking all nodes if they have the requested file.

An overall overview of state transitions is as follows:

- Upon reception of a `QUERY`, the servent first checks if the messageID associated with the query has been seen before.
  - if it has, the servent abruptly returns (aka removes the query from the network), assuming that the request has been appropriately forwarded at a previous time. This is so the network is not clogged by spurious queries.
  - if it has not, the servent maps the `messageID` to the sender's InetAddress, so when it receives a `HITQUERY` for that messageID it can propagate the HITQUERY back to the original requester.
- Upon reception of a `HITQUERY`, the servent first checks if it has seen the same MessageID (corresponding to a previous `QUERY request`).
  - if it has not, the servent abruptly returns (aka removes the query from the network). This indicates a spurious `HITQUERY` request.
  - it it has, then there are two options. If the messageID is one that has servent sent, then it attempts to retrieve the file using information contained in the payload of the packet (that is, the IP address of the servent with the file).
  - If the target servent is under a firewall (as indicated by the appropriate flag in the config file), then the servent sends a `PUSH` message telling the target servent to initiate the file download connection.

  - Otherwise, the client opens a direct connection to the server to download the file.

- Upon reception of a `PUSH`, the servent checks if it has seen responded to the requested file with a `HITQUERY` message before. If so, it initiates a direct connection to the requesting servent (whose IP address is contained in the payload of the `PUSH` packet) and sends the file).

- Upon reception of a `BYE`, the servent removes the sender of the message from its list of neighbors.

In addition, each peer has multiple threads, one listening to user input in order to see which files to request, and several listening for incoming connections. We used the share-welcome-socket technique from the HTTP server assignment for accepting new connections.

Threads prepared to handle `QUERY/HITQUERY` messages listen on port `7777`, and threads prepared to handle file downloads listen on port `5760` (port 80, in the original Gnutella spec). When a packet is read from the socket, the handler determines which type of message it is (`QUERY, HITQUERY, OBTAIN, PUSH`) and handles it appropriately.

2.1 Input and Output

Once the servent starts, user input on the command line should be the names of files that the user wants to request, separated by spaces. An alternative way to request files (especially ones with spaces) is to put them one per line in a file which is fed to `main` using the `-f` flag.

If the `-d` flag is present in the command line args to Servent, the servent will print debugging output that specifies information about: incoming connections, packets being received and sent, which handlers are handling messages and where they are in the process, etc. This is intended to help the user / developer understand what is going on.

A servent requesting a file will also inform the user whether that file was downloaded successfully, and will output the contents of the file if it was successful.

If any files were not successfully downloaded, it will inform the user of those files. It will also write the contents of the downloaded file to a new file in the `ROOT` of the servent, with the file named <`original_name`>`-download`.

2.2 Architecture

The architecture is based on the fishnet TCP implementation from assignment 4, adapted for the Gnutella protocol. An overview of the architecture is as follows:

- **`GnutellaPacket` and headers**. Similar to the Transport class in the Fishnet TCP implementation, `GnutellaPacket` includes verification features, unpacking from bytes to object, and packing into bytes (multiplexing and de-multiplexing). Headers follow the protocol specification.

- **Pushes**. Because the Push message is accompanied by its own header, we decided to create a class to represent, encode, and decode this header. The `PushMessage` class is designed to be packed and set as the payload for a `GnutellaPacket`.

- **Threads** We created two types of threads, `FileThread`s and QueryThreads, to represent the ports these threads are listening to, and thus the types of requests these threads satisfy.
- **Thread handlers.** We created two types of handlers, `MessageQueryHandler` and `MessageFileRequestHandler`, to handle requests at each port.
- **Core data structures**
  - We kept track of `QUERY` and `HITQUERYs` using a `LinkedHashMap` mapping pairs of (Descriptor, MessageID) to InetAddresses. Hash maps are ideal in this situation because of their fast look-up time. In addition, we made it threadsafe using a synchronized block because multiple threads may be accessing it at once. Finally, we included the ability to prune old entries by popping the first entry, thus creating a bound on the size of the array.
  - We kept track of files the servent has seen using `HashMap` that maps the `hashCode()` of the file uri to a `File` object.
- **Configuration file** We determined the format and implemented the parsing for our own customized configuration files, as detailed in the How to Use section.

2.3 Achievements & Impact

We succeeded at creating a multi-threaded file-sharing system that can share files between both adjacent and non-adjacent peers using only neighbor-to-neighbor queries. This protocol is relatively simple, but useful for applications - Gnutella applications in deployment have had millions of users.

In addition, we followed the assumptions detailed by the protocol in deciding when to discard packets. This is important for real-world deployment because if used in an actual application, our implementation does not exist in isolation; it has to be respectful of not congesting overall network traffic. Therefore, both correctness and politeness (as defined by not clogging the network with unnecessary packets and creating potential security holes) are both important.

We also implemented it slightly differently, using a `OBTAIN` request rather than an HTTP request to download a file from a server-servent to a client-servent. This indicates the flexibility of gnutella and a possible alternative to the current design, although it mitigates the usefulness of the `PUSH` request.

3 Possible Improvements & Extensions

A few extensions to the current implementation we did not have time to work on, but considered, are:

1. **Neighbor Discovery.** Since we implemented a simplified version of Gnutella that has a static configuration, our peers do not use PING and PONG to discover neighbors. A future improvement for the project would be to have a dynamic configuration and peers that use PING and PONG.

2. **HTTP file requesting.** Gnutella uses HTTP to directly request files when a servent receives its `HITQUERY` back. We did it differently using a packet sent directly to the servent with the file containing an OBTAIN message. This works just as well in our testing, but to more accurately emulate the Gnutella protocol we could improve the project by using HTTP requests.
3. **Ultrapeers.** Gnutella 0.6 treats some users of "ultrapeers" which contain many connections, are faster, and have most messages routed through them. This shields other, slower nodes from dealing with excessive traffic.
4. **Caching.** Similar to the caching in HW3, frequently accessed files could be cached in a servent so it can upload the file more quickly when that file is requested.

4 Problems Encountered

We had some problems reading to and sending from the correct sockets. Since the sockets remain open, reading a message from the socket's stream would hang because the connection never closed and thus read never returned -1. To solve this, we made the first 4 bytes of every message indicate the int that was the length of the message, and thus the number of bytes to read from the socket. Then the thread could wait until that many bytes were read and move on to handling the message.

Another problem we encountered was sending return messages (e.g. a HITQUERY) between messages correctly. We encountered this problem because we were saving the port and InetAddress to send the HITQUERY to, but were creating a new socket to send the message back rather than writing to the original socket that received the message. This meant that the local computer assigned a different local port to the message and thus the other machine did not recognize the message as part of the original connection. We fixed this by passing in the socket as a parameter to the handlers, rather than just the InetAddress of the message destination, so the handler could write directly into that socket.

5 How to Use

Each individual peer (or "servent") is started on a separate machine in the command line. A servent can be started through the command:

```
java Servent -id < id_number > -config <config_file> [-d]
```

where <`id_number`> is a integer ID number unique to the servent, and <`config_file`> contains the static network configuration for all servents in the system. All servents must be started with the same configuration file. -d is optional, and if present it will turn on debugging output.

After all servents have been started, the user can type into the command line the names of one or more files they want to request, separated by spaces. Press enter and the servent will attempt to request and download these files.

We will provide some testing instructions and example configuration files for testing, listed at the end of this document.

5.1 Configuration File

The setup for a servent of a configuration file is as follows:

```
<Servent ID#>
ROOT: root/directory/to/search/for/files/in
<Neighbors>
 ip.address.or.name.of.neighbor yes/no
 ip.address.of.other.neighbor yes/no
</Neighbors>
</Servent>
```

Where:

- `ID#` is the positive ID number of the servent, provided on the command line with -id. The servent also uses a cryptographically secure UUID identifier; however, the usage of the simplified ID system in the configuration files increases the ease of deployment and rollback.
- `ROOT` is followed by the local directory that the peer will consider to be the root from which it searches for requested files.
- `<Neighbors>` contains a list of IP addresses or names of the machines that are neighbors to this servent. This is followed by either "yes" or "no" which indicates whether the servent acting as the server (the one with the desired file) needs to initiate the file download instead of the client servent, i.e. if there is a firewall.

  A configuration file contains a list of these setups, one for each servent in the system, separated by newlines. We took inspiration from the Apache configuration files referenced in homework assignment 2.

6 Testing

Please see the `README` at the source code repository.

7 Source Code

All source code is available at `https://github.com/linii/gnutella-cs433`

8 References & Resources

1. Original assignment: `http://www.cs.iit.edu/~iraicu/teaching/CS550-S11/pa2.pdf`
2. Gnutella protocol, 0.4: `http://rfc-gnutella.sourceforge.net/developer/stable/`
3. Gnutella protocol, 0.6: `http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html`