***Gnutella on a Twisted Framework***

CS 114: Peer-to-Peer Networks

Professor: Giovanni Pau

Winter 2012

Emily Chen (403660109)

Pei-Ying Hsieh (603658544)

Alvin Huynh (503678533)

***Foreword***

Gnutella is a decentralized peer-to-peer network that is used to share files. It has five packet types: ping, pong, query, query hit, and push—all of which we have implemented excluding push for reasons later discussed. For our version of a Gnutella client, we decided to use python because it is simple and quick to write code in and has numerous socket handling libraries. Upon further research, we found that the Twisted framework was used to make an earlier version of BitTorrent. Twisted is an event-driven networking framework made in python that greatly simplifies writing network applications. Thus, the Twisted framework was used to create our Gnutella client application because we thought it would be perfect to handle the messages that came in at various times.

***Instructions***

General way to run client (will act as first node in network):

python gnutella.py [directory]

where [directory] is the directory that will act is the node's directory.

Files in the [directory]/files folder will be shared on the network.

A [directory]/output.log file will be created to log interactions the node makes.

If files and directories do not already exist, program will generate them.

To connect to another node in an existing network on start up:

python gnutella.py [directory] -i [ip] -p [port#]

\*\*Note: **please use the same IP for each node** (e.g.: do not connect to a node using "localhost" on some connections and the actual IP on others)

To watch the log file while the node is running, run in another terminal:

tail -c +0 -f [output.log path]

To request a file, after the node is running, enter:

GET [filename];

Downloaded files will be saved to the [directory]/files folder.

To stop the program:

Enter "QUIT" and then type a keyboard interrupt (Ctrl + C on mac).

***Implementation Description***

1. *Connection specifics and port allocation*

In Twisted, ports can be set to listen or to make a connection with another port. The connections made with our client are all TCP connections. At the start of the Gnutella client, a random, open port is select to listen for incoming connections. Other nodes can connect directly to this listening port—naturally, the listening port can handle multiple connections at a time while watching for new ones. However, when the client is making an outgoing connection, because of the way Twisted connections work, it will select a new, unused port to make the connection instead of using the listening port (but connects to the other client’s listening port). Thus, a single client may be in charge of multiple ports.

Each node is limited to a maximum of 10 connections (this value can be configured). Fortunately, connections are monitored in Twisted and an event handler exists for when any connection is lost. Thus, keep-alive checks with TTL of 1 are unnecessary in this version of Gnutella because when a node dies, its direct neighbors will be alerted through an event. Upon a loss of connection, the client will retry the connection once and if that fails, it may try to connect to a random node with a 50% or 10% depending on how many connections it currently holds. If it has more than 3 connections, it will use the 10% probability. These probabilities and the boundary for the change in probabilities can also be configured in the global values (see *Global configuration values* for more). We hope that these probabilities will produce a more interesting network map than connecting to the first few nodes that join the network or the nodes that have the most shared data.

1. *Protocol*

Unfortunately, bit manipulation is not easy in python because it inherently converts most values to strings—particularly with its various read functions. Thus, for simplicity and readability of code, we decided to transfer messages between nodes as strings instead of bit by bit. However, we kept most of the message formatting the same.

**Node ID assignment**

To ensure that each node’s ID was distinct, we simply used the MAC address of the computer and appended the node’s listening port.

**Gnutella Connect and Gnutella OK**

The Gnutella Connect and Gnutella OK will also include the message sender’s listening port. The receiver will save its peer’s listening port number for reference. The reason we decided to do this will be discussed in the *Challenges faced and solutions: Redundant TCP connections between two nodes* section.

**Message formats**

We retained the most of the general message formats mentioned in the specifications. However, we did not see a need to keep both the TTL and number of hops in the header, so the number of hops was removed. Furthermore, to ensure that each message is parsed correctly, we put a delimiter (‘&’) between each portion of the message. Thus, a Ping message would look as follows: [msgID]&[payload descriptor]&[ttl]&[payload]

**Message IDs**

The client keeps track of how many messages it has sent. The message ID is simply the message number appended to the node ID. However, for each message ID that a node comes across, it stores where the message came from in a dictionary with the message ID as the key. To ensure that the nodes will not run out of memory over a long time, after each 1000 messages, the message number is reset to 0. To differentiate between two different messages with the same message ID, a timestamp of when the message was last seen is added. If a message ID has not appeared for 3 seconds, it is deemed out of use (this timeout is configurable, see *Global configuration values* for more). For different network speeds, processor speeds, and large amounts of traffic, **the message lifetime value may need to be tweaked.**

**Ping**

By default, pings have a TTL of 7 to ensure that the network is not flooded with ping messages that would lead to a flood of pongs.

**Pong**

Pongs follow the message path back to the originator. Each node will save the payload information from the pong. Furthermore, whenever any node receives a pong it will attempt to connect to a peer with the probabilities stated in the *Connection specifics and port allocation* section. Normally, a pong will return the number of files the sender has and the total amount of data it has available. Since our implementation does not use a node’s amount of available data to determine which nodes it should make its neighbors, passing the shared data amounts is unnecessary. Thus, pongs only relay the IP and listening port of the pong-sending node.

**Query**

Queries, like pings, have a TTL of 7 and flood outwards. If the file is not found within 7 hops, the requester request the file again after the network topology has changed or one of the nodes within 7 hops has also requested and successfully retrieved the file. As with pings, this design decision was to minimize network traffic.

**QueryHit**

In addition to its listening port, each node has a port to serve files as an HTTP server would. When a node receives a query and has the requested file, it will send a query hit to the requester. The query hit includes the node’s IP, file serving port, and the requested file name. Upon the first query hit the requester sees, it will send a HTTP GET request to the file serving port of the query hit sender to retrieve the file and save it in the [directory]/files folder.

**Push**

We decided that implementing push to get around firewalls was unnecessary because our implementation should be used within a single network (on the same router or even on the same computer). When our group tried testing through a normal network, we could not find the IP of our actual computers; our router’s public IP was always returned instead. Thus, connections between computers using different routers could not be made because the given IP could not identify the computer behind the router. Thus, firewalls were not an issue under our test cases—which made push unnecessary to implement.

1. *Global configuration values*

As mentioned in the previous sections, there are a few global values that can be changed to configure the node. These values can be found at the top of the gnutella.py file under the “GLOBAL DATA” section. Some of these values may need to be changed based off of network topology, network speeds, amount requests and traffic, etc…

The values are as follows:

* *msgTimeout* – the lifetime of a message ID in seconds. If a message ID has not been seen within the msgTimeout number of seconds, it can be recycled, meaning a node should not ignore a message with that message ID if it has already seen it before.
* *MIN\_CONNS* – when the node has less than this many connections, it will use the UNDER\_PROB (under the min connection barrier) to decide if it should connect to a new node.
* *MAX\_CONNS* – maximum number of connections a node can maintain at any time.
* *UNDER\_PROB* – the probability, in percent, that the node should attempt to connect to another node if the number of connections is under MIN\_CONNS. (e.g.: 50)
* *OVER\_PROB* – the probability, in percent, that the node should attempt to connect to another node if the number of connections is over MIN\_CONNS but under MAX\_CONNS. (e.g.: 10)

***Test Results***

The first testing on the completed client was done with 6 nodes on a single machine to ensure that the general functionalities were correct. General tests for pings, pongs, queries, query hits, and file transfers were tested. Afterwards, we tested thirty nodes on three computers—10 nodes per computer. All three computers were connected to the same router. As we were adding nodes to the network, we simply chose random nodes to connect to. For the most part, the nodes that were alive longer had more connections after all the connections had been completed. This occurrence makes sense because the nodes that lived longer would have seen more pongs than the newer nodes. After disconnecting some nodes, most nodes still held at least 2-3 connections because the client considers reconnecting to another node when one of its connections is lost.

***Challenges Faced and Solutions***

**Handling output and user input**

Initially, we were not sure how to implement user interaction so the user could see what the client was doing and request files at the same time. Our first thought was to write the output to the terminal and take requests from a file that the client would periodically check for changes. However, this solution seemed like it could be quite buggy or possibly exploitable. We also disliked the thought of writing output to a file because then we could not see what the node was doing in real time. Fortunately, we found the UNIX tail command and its –f option that constantly reads the changes to the end of a file and prints it to the terminal.

**Serving files**

At first, we were unsure how we would transfer files between nodes. After some research into Twisted, we realized we could set up a simple HTTP server for each node with one of Twisted’s libraries and have it serve files as a webserver would.

**Send buffer issues**

After tests with more nodes, the number of pings and pongs at any given time increased. Since many of the nodes were on the same computer, messages were sent between them very quickly. After a few tests, we noticed that despite Twisted’s event-driven nature, the listening port would sometimes read multiple messages in as a single line; this problem messed up the message parsing and handling. To fix this, we simply added a delimiter (‘;’) after each message and split the incoming messages on the ‘;’ character and handled each separately.

**Redundant TCP connections between two nodes**

Because each node is comprised of more than one port, checking whether or not a certain node was already connected to did not always work. Naturally, we take the IP and port numbers from the pong messages and save them. Afterwards, we check if the receiver was already connected to the node that sent the pong message and remove the pong-sender’s info from the list of possible new connections. However, node A would often mistakenly believe that its connection to node B, which used a different port number from its listening port to connect to node A, was from a different node because its port number did not match its listening port. Thus, node A would make a new connection to node B through B’s listening port. If B saw a pong from one of its other port’s it could even mistakenly connect to itself.

To remedy this situation, when a node sends a GNUTELLA CONNECT or GNUTELLA OK, it also includes its listening port number. The instance of the gnutella protocol run on each port would save this port number and correctly remove the information of the nodes it is already connected to from the list of viable new connections.

**IP issues**

We could not figure out how to connect to clients on different networks. Each IP address that we tried to use did not match up to the actual machine’s IP—instead, it was the IP of a router that did not know which computer to forward the information and connection to. As mentioned earlier, however, our program runs fine if all the computers are on the same router and uses the IP addresses of each computer within the router’s network. Unfortunately, we could not find a solution to this problem.