DOCUMENTATION

I - Introduction

This first version of the gossip server is described to be an 'iterative server'. For that reason, some design choices (most notably for TCP) were made which may be suboptimal under certain conditions. This document will address these issues, and provide appropriate explanation where necessary.

II - Technologies Used

The project is implemented in Java, and relies on the Apache Commons CLI library for parsing command line options. We opted to use the filesystem rather than a database library for saving information.

III - Architecture

i) Structure

The project centers around interactions between three central classes, TCPServer, UDPServer, and GossipServer. The TCP and UPD servers inherit from GossipServer.java. GossipServer.java handles the bulk of the work, processing messages and interacting with the data files. The UDP and TCP server classes handle the networking side of the project, receiving and sending messages from and to peers via UDP and TCP connections respectively. The project begins with a different class called Main.java that takes in command line input using the Apache Commons CLI library. Main.java then creates a thread for the UDPServer and one for the TCPServer and then runs those threads. Two other classes also exist for support: Gossip.java and Peer.java. Gossip.java stores information about a Gossip message and provides functions for formatting a Gossip message to be stored in the gossip data file and forwarding the gossip message to other peers. Peer.java does the same thing as Gossip.java but for information about peers instead of information about gossip messages.

ii)Design Choices

There were a few design choices made in this project with the aim of staying as close to the project description as possible. Most of those design choices concerned TCP connections and TCPServer.java. Since this version of the project is meant to be an iterative server, and simply to accept messages via tcp and udp connections, it accepts one message at a time and then closes the connection after that one message. The TCP server accepts a single message over TCP and then closes the socket connection to be able to handle the next socket connection made with a new message. For this reason, any peer connecting to the server via a TCP connection will have to open a connection, send a message, have their connection closed by the server or themselves, and open a new connection to be able to send another message. The TCP server only accepts one message per TCP connection in order to remain strictly iterative in function. It was our design choice to implement the TCP server in this manner because it stayed closest to the project description, even though it is not a very realistic implementation of a TCP server. This implementation stays closest to the project description because it is guaranteed to be fully iterative, handling one connection at a time, and still fulfills all of the requirements given for the project. It can

process all three types of messages correctly and send both types of messages properly as well. With this design choice came a second design choice regarding how the TCP server "broadcasted" gossip messages to all known peers. The server opens a new socket to every peers IP and port and send the gossip message to them at that IP and port, so each peer connecting to the server will need to have the ability to receive messages on their specified port (the port given in the PEER message containing the peer's information) even after their connection used to initially contact the server has been closed, since each connection is closed after the message it provides is received. However, in the case of the PEERS?\n message being received, the TCP server will send the reply message listing all known peers to the requesting peer before closing the socket connection.

iii) Data File Format Documentation
Our project uses simple text (.txt) files to store data about known peers and gossip messages. To keep the implementation and parsing simple, the project uses two different files to store the data in; one for known peers, and one for gossip messages. The file for known peers is named peersFile[#].txt and the file for gossip messages is named gossip[#].txt. In both cases, the file is created in the directory specified with the command line argument "-d /data/file" as specified in the project description. The last character before the file extension in each file name is a dynamically generated integer between 0 and 999, inclusive. This is done to ensure that, even if a file with the same name already exists in the specified location, the server will start with an empty and known file, without overwriting any existing files. Each file has its own formatting system for how the data is stored in the file, and each is detailed below:

gosssip[#].txt:
Because the gossip messages are forwarded to known peers in the exact format that they are received in, there is no need to change the way that the data is arranged for storage in the gossip data file. Each unique gossip message received is stored in the gossip data file with the format:
   <encoding>:<dateTime>:<message>%
The first field, GOSSIP, does not need to be included since all data in the gossip data file is already known to be about gossip messages.

peersFile[#].txt:
Each peer only requires one line since the newline character cannot appear in the information about a peer, so the format is simple. Each line in the peers data file contains information about a single peer in the format:
<name> <port> <ip>
The three fields can be separated by any amount of whitespace, but must always be separated by at least some whitespace so that they can be read properly.

==========
IV - User Manual
==========
i) Class 'Main'
- main()
   - parses command line options, consisting of the following:
      - p <portno> (required)
      - d <directory> (required)
      - v (optional)
   - creates data directory and files

   - creates Semaphore locks for file access, logging
   - starts two new threads, one for UDPServer and one one for TCPServer

ii) Class 'GossipServer'
- String receiveMessage()
   - blocks until an Error has been encountered, or a recognizable message has been parsed
   - returns the incoming message as a String, interpreted using latin-1
   - returns null if the message cannot be parsed
   - throws an IOException if the connection is terminated partway through receiving a valid message
- boolean sendMessage(String message, Peer peer)
   - sends 'message', encoded in latin-1, to 'peer'
   - returns true if the transaction was successful, false if an error occurred
   - Note: In the case of TCPServer, a Socket is opened, then closed. It can be argued that this simply mimics UDP, and is suboptimal for TCP connections. While this is true, maintaining a Socket object for each reachable peer could be wasteful depending on the level of traffic. Additionally, maintaining said Socket objects would not be truly iterative
- String runCommand(String[] fields)
   - Performs the command specified by the received message
   - 'fields' is the incoming message, split into sections separated by the ':' character
   - returns the response to be sent to the source of the message, or null if there should be no response
- void processGossip(String[] fields)
   - runCommand delegates to this function if the first field is GOSSIP. If the gossip message is already known, DISCARDED is printed to std error. If it is not known, it is filed, then forwarded to all know peers with sendMessage
- void processPeer(String[] fields)
   - runCommand delegates to this function if the first field is PEER. If the peer is not known, it is recorded. If it is known by name, but the ip address or port differ between the recorded data and the message, this information will be updated
- LinkedList<Peer> getPeers()
   - returns a list of all known peers, encapsulated as Peer objects.
   - returns null in the case of any File IO Error
void log(String s)
   - prints 's' to std output, if the server was launched with the -v option
String getPeerListing()
   - reads the peers file
   - returns the formatted String to be sent as a response to the PEERS? query
T withGossip(Supplier<T> fileAction)
   - Performs an action once the gossip data file has been locked (Semaphore)
   - returns the type of the given Supplier. In some cases, this return value is not used
T withPeers(Supplier<T> fileAction)
   - like withGossip, but locks the peers data file

iii) Class TCPServer : GossipServer
   *Functions already described in GossipServer section. Architecture ii describes
implementation level details.
iv) class UDPServer : GossipServer

*Functions already described in GossipServer section.

v) Gossip
- String getContent()
-returns the textual portion of a gossip message (not including hash or time)
- String getFileFormat()
- returns the gossip object, formatted to text for use with the filesystem
- String getMessageFormat()
- returns the gossip object, formatted for sending over the network

vi) Peer
- String getName()
- returns the name of this peer object
- int port()
- returns the port number of this peer
- String getIp()
- returns the address of this peer
- boolean equals(Object o)
- compares this object to 'o' based on name, ip, and port
-String toMessageFormat() ... String toFileFormat()
- identical to Gossip's methods of the same name

==========
V - Conclusion
==========

In conclusion, this project is designed to be a first and simple iterative version of the GOSSIP P2P system. It is just that and meets the specifications given in the project description fully, accepting and sending the specified messages over TCP and UDP connections, one connection and message at a time. Due to the fact that the server is supposed to be a simple iterative server, it is not implemented in a very realistic way, especially with regards to TCP connections, but it is not meant to be a realistic implementation, just a working one that matches the project description as closely as possible.

==========
VI - References
==========
This project uses command line option parsing code adapted from Dmitry Kulakov's example provided during his in-class presentation. The home page of the parsing code is: http://commons.apache.org/proper/commons-cli/