DOCUMENTATION

I - Introduction

This milestone of the project makes the server concurrent where it was previously iterative, for both the UDP and TCP parts of the server. A few other small changes were made to the server, and this document will describe all changes made to make the server concurrent, and the few other ones made as well. In addition to the changes to the server, this milestone also adds a client program that has a command line user interface that allows users to send gossip messages. This document will also describe the client program. Throughout this document, since it describes both the changes to the server from the previous version, and the client program, each applicable section will be split into two sub-parts: one for the server changes and one for the client.

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 file system rather than a database library for saving information.

We introduced open source code to produce the hash function for gossip messages in the client program. Specifically, the SHA256 class was found on github at the following link:
https://github.com/BaconRemovalUnit/SHA256/blob/master/src/SHA256.java

III - Architecture

---Server---

i) Structure

The server is still centered around the three central classes TCPServer, UDPServer and GossipServer. However, there are two more central classes now that the sever is concurrent: UDPConnectionHandler and TCPConnectionHandler. The TCPServer and UDPServer classes no longer inherit from the GossipServer class, now the connection handler classes inherit from the GossipServer class. The TCPServer and UDPServer class now have a simple functionality, of being a server. They listen on the specified port for connection to be made, and every time a connection is made, they pass the connection (or datagramPacket in the case of UPD) to the connection handler class. The connection handler classes take a single connection each and process it, again passing most of the work to the GossipServer class for file handling and message content processing. The connection handler classes themselves handle the networking, receiving and sending messages to peers via UDP or TCP. The project again begins with the Main.java class that takes in command line input and creates a thread for the UDPServer and TCPServer to run in, and starts those threads. Now, to make the server concurrent, the TCPServer and UDPServer classes will create a new thread of the TCPConnectionHandler or UDPConnectionHandler respectively and start that thread, giving to it the connection or datagramPacket that TCPServer or UDPServer received, then let that thread run on its own and go back to listening. The created threads of UDPConnectionHandler or TCPConnectionHandler will do what their name implies, and handle the connection. There are also two datatype classes that still exist,

Peer.java and Gossip.java. However, now they both implement two new classes, FileFormattable.java and MessageFormattable.java, that specify and provide both the peer and gossip classes with the function for converting the information about the peer or gossip message into a string format suitable for sending as a message over the network, or for storing in the data files.

ii) Design Choices

There were a few design choices that had to be made when converting the server to be concurrent. Probably the most impactful design choice is the choice on how to handle TCP connections made and given to threads of the class TCPConnectionHandler. Each TCP connection made is handled by the thread, retrieving the entire message from the socket and processing it, and then the socket and connection is closed. This is done so that more connections can be handled by the server without causing the machine the server is running on to crash by creating too many threads. If each thread was allowed to maintain its socket connection open, the thread itself could never end, and the server would create too many threads. So, each thread handles and processes the connection, and then closes it. As a result, each message sent via TCP requires its own connection to be opened, and a single TCP connection cannot send more than one message. Another important and concurrency related design choice is the choice to limit the number of threads that the servers can create to 100 threads for TCP connections and 100 threads for UDP connections. This is done to prevent the server from crashing the machine it is running on by creating too many threads. When a server receives a new connection, but is already at the maximum number of threads, it will pause in a loop until one of the threads has died, and then give the connection to a new thread and continue listening.

iii) Data File Format Documentation

There was a change made to how the server stores data about known peers and gossip messages. Previously, the name of the simple text (.txt) file storing the data about peers and gossip messages included a dynamically generated number before the file type, so that each time the server was run, it created a new set of text files to store data in, assuring that the data would start empty and not overwrite any existing files of the same name. Now, the server will generate two simple text files, one for gossip messages and one for known peers, without that dynamically generated number, simply called "gossip.txt" and "peers.txt", and it will start with whatever data those text files hold before the server was started. The specifications again are 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.

---Client---
i) Structure
The structure of the client is based around three main classes: ClientApp.java, UDPClient.java, and TCPClient.java. There is another class called ClientConnection.java that sets up an interface for the UDPClient and TCPClient classes to use. The ClientApp class does most of the work for the client as it serves as the user interface for the client application. It uses the command line as its user interface, printing to and taking input from there. The client is started from the main method in the ClientApp class and takes in command line input for specifying the server IP and port to connect to, whether to connect via UDP or TCP, and optionally a gossip message with an optional date/time to automatically send to the server. The ClientApp class will create an object of the UDPClient or TCPClient class, based on the command line argument indicating which connection type to use, and use that object to send and receive messages to and from the server. All of the networking is done inside the TCPClient and UDPClient classes as the networking is connection type specific. The client makes use of two supporting classes, Hashing.java and SHA256.java, for encoding the gossip message strings it receives from user input into sha-base64 to send to the server. The client also makes use of the Peer.java and Gossip.java classes discussed in the server structure.

ii) Design Choices
Most of the design choices for the client are based on interfacing properly with the server. The most prevalent design choice is to create a new connection to the server for every message to send to the server, as the server requires, for both TCP and UDP connections. This is done by, after each message is fully sent to the server, just creating a new object of the TCPClient or UDPClient classes, and letting the Java garbage collector remove the previous one. The design choice was also made to user the command line for the client user interface as the client was not written to run on an android. Another less impactful design choice made was the choice, when the user specifies that they want to specify the timestamp for a gossip message being sent to the server (done via the 'gt' command versus the 'g' command in the client), to also ask the user whether they want to specify the date and time or just the date. Both options work fine and if the user only specified th data and not the time of day, the current time of day of the machine is used. If the user does not want to specify the date at all, then the current time and date of the machine is used.

==========
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

- these servers will then create new threads for each incoming connection

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 : Runnable

void listen()
  - continually accepts TCP connections, each in a new instance of TCPConnectionHandler

iv) class UDPServer : Runnable
      - continually accepts UDP connections, each in a new instance of UDPConnectionHandler

v) Gossip : FileFormattable, MessageFormattable
- String getContent()
   -returns the textual portion of a gossip message (not including hash or time)

vi) Peer : FileFormattable, MessageFormattable
- 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

vii) FileFormattable
- String getFileFormat()
      - returns the object, formatted as a String to be stored in a File

viii) MessageFormattable
- String getMessageFormat()
      - returns the object, formatted as a String to be sent over the network

ix) Hashing
- String hash(String str)
      - fulfills the purpose of this class. Other functions are merely helpers
      - returns the given String, run through SHA256, then encoded in base 64
- String toBinary(String hexString)
      - Accepts a hex string and returns the data as binary
- String encodeBase64(String str)
      - encodes str in base 64
- String sha256sum(String str)
      - performs the sha256 algorithm on str, and returns the result

x) SHA256
* This is the class that was found on github which we used to help with Hashing *

xi) ClientApp
- main
Parses the following options:
- p - required - port number of server
- T - use TCP (default)
- U - use UDP
- s - required - server domain/ip
- m - optional message to send immediately upon running
- t - send a timeStamp besides the current time (default)

* upon running, the application has clear instructions for how to send additional gossip, peers, etc. *

xii) ClientConnection
boolean send(String message)

- returns the success/fail result of sending 'message' to a server
boolean good()
- returns the status of the connection
void close()
- closes the connection
String recieve()
- returns a message received over the network

xiii) TCPClient : ClientConnection

xiv) UDPClient : ClientConnection

xv) UDPConnectionHandler : GossipServer

xvi) TCPConnectionHandler : GossipServer

xvii) AppUtils
- String readFromTCPSocket(Socket socket, int maxByteSize)
- reads up to maxByteSizeBytes from socket, returned as a String

V - Conclusion

In conclusion, the server had successfully been converted from being iterative to concurrent by creating a new thread to handle each client connection, up to a maximum of 100 for UDP and 100 for TCP. As a result, it has become a more realistic version of the GOSSIP P2P system, while still being a simplified version of it. It continues to meet all of the specifications provided by the project description and has been refined in those areas somewhat as well. The client program successfully fulfils the need of providing a user interface for users to send gossip messages to the gossip server so that gossip messages will be forwarded to known peers. Overall, it is a simple implementation of a client program because it uses the command line as a user interface and does not have a large amount of functionality. It meets the feature specifications given and goes one small bit further by allowing users to specify date and time or just date, but is still simple and can only send the three main types of messages to the server.

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/