**MARK TODD**

**FINAL PROJECT**

**CPSC 360**

**Milestone 1: Project Proposal**

**Project Title:** Low-memory TCP Server and Client

**Project Description:** The project will contain two essential pieces, a client and a server.

Client- The client should be as barebones as possible with as few basic rules as possible. This is to allow for implementation of clients by others to still have full ability with the server. The client should theoretically be able to be written in any language. The client will be able to connect to a server on a given port, send and receive messages and interact as the server intends.

Server- The server should be a little more complex. The end goal is to have a scalable model to allow for enhancements while still using the least amount of memory possible. A set of simple commands will come with the server. However, the idea is to allow for any feasible command to be easily implemented into the code. The server should accept multiple clients and allow for communication between them. It is a “Mass Client” server which means multiple clients connected to the same port and ip will be able to communicate with each other.

**Phase 1:** Phase 1 will be creating clients that can communicate with each other through a direct to-and-fro communication. Essentially this will involve the client to be able to handle communication as a server would. However, this will only be in the most abstract sense. One client will be a “server” binding a port and listening for a connection. The other will initiate that connection.

This is phase 1 as it involves the communication in its simplest iteration. Sending and receiving messages.

**Phase 2:** Phase 2 will be removing clients “server” capability and creating a server that will bind to a port and accept clients. It should be set up to allow multiple clients at once and facilitate the intercommunication. I honestly don’t know how this will be done. From what I’ve read, connections are exclusive; meaning communication would at first be exclusively between server and client, which is not what I want. Clients should also be able to separate from the server on command to private chat with each other.

**Phase 3:** Phase 3 involves allowing for multiple servers connected. This connection will allow communication between clients on either server with each other while spreading out the bandwidth load between the two. Clients should be able to seamlessly shift between the servers. This phase will also be where a basic chat gui is developed such that clients don’t “have” to use a terminal.

A side effect of phase 3 will be the ability to log data.

**Expertise:** Mark’s has decent capability with Java (the environment of choice for the project) and is extremely intrigued by the idea. While he networking experience is pretty much exclusively this course, course up to now have presumably prepared him at least in finding answers to problems that surface and researching where ignorance causes issue.

**Milestone 2 – Project Writeup and Submission**

**Introduction:**

I chose this project because I thought this would be a fun way to continue learning about networking through a practical program. It also seemed intensely interesting for reasons I can’t define. The project is pretty close to the proposal in what it does (which has been included above). It is a ‘mass client’ chat application (handles a number of clients restricted only by the server’s hardware in theory).

**High Level / Conceptual Design –**

Requirements: In order to run the server, the computer must be connected to a network, be given an IP address and decide on a port that is \*open\*. Once created, the server can safely be left alone while clients interact through the network. As with the server, the client must be connected to the same network (ie internet). It must also know what IP address and port the server it intends to connect to.

The computers running the client server must have access to a terminal/command line and have Java Runtime installed the computer (depending on the system, Environment Variables must also be assigned).

Illustrations:

**Detailed Design –**

Client Design:

**Pseudocode:**

|  |
| --- |
| createClient(){  Ask for port and server address from user through System.in  Set main and auxiliary server and port variables to input.  chat();  } |

|  |
| --- |
| chat(){  while(test){  countdown pingtimer;  state change ‘switch statement’{  if in ‘start’ state: setup sockets, data stream, and buffer reader;  set state to ‘receive’;  if in ‘receive’ state: ping on pingtimer = 0 and reset pingtimer;  accept messages from server (via buffer reader) if they exist;  read from System.in and send to server if user used input;  if System.in entered ‘endchat’ command, set state to ‘disconnect’  if System.in entered ‘changeserver’ command  change mainServer port and ip to input and set state to ‘disconnect’  if in ‘disconnect’ state:  if mainServer information isn’t equal to auxiliary (or current) then change and set state to ‘start’  else close all connections and test = false;  }  }  } |

Server Design –

**Pseudocode**

|  |
| --- |
| MainServer(int port, String serverName){  create new instances of the ArrayList and HashMap that will store client information, and the Lists that will hold announcements;  setup server socket then move to threads()  } |
| threads(){  while(not set to shutdown)  listen for clients. When one connects, give it a name, create a new thread for it, add it to HashMap and ArrayList, then start the thread;  } |
| **(From ThreadMaintain internal class)**  ThreadMaintain(Socket newS, String name) {  Set up Reader and Output Stream for the client to listen (and report to) the socket)  add it to the server’s announcement list to accept incoming messages  } |
| **(From ThreadMaintain internal class)**  run(){  while(not told to stop){  read through announcements to see if any not yet said to client (if so add to ‘reports’ list);  if client sent data, read it and handle commands on it. Put message into announcements with client name attached; (Special side note: ‘whisper’ command allows to clients to privately message through announcements)  if ‘reports’ list is not empty, send all data in it to client then empty it.  }  } |

***More* English Explanation:**

Client handles input from System.in (the terminal/command line window) and reports what it reads to server. Server then takes that data and adds it to an “announcements” list. Every client-thread (on Server) goes through this list and reads out any data not yet reported to client. The announcement is effectively the chat working outside the threads accessible to all individual threads.

There are a few ‘commands’ that start with a ‘/’ prefix that will perform specific functions described when ‘/commands’ in inputted. Whisper deserve special mention only because it acts within the ‘announcements threads’ but only allows access to the intended recipient instead of everyone connected.

The interaction between the announcement list and the individual threads is where the bulk of the work occurs on the server.

**Results:**

*Not completed items:*

There is no gui. I simply didn’t have the time the relearn how Swing works and ensure it didn’t falter through network interaction. I do plan on adding this on my next iteration (as I actually want this to be something I’ve followed through to absolute completion).

Clients cannot direct connect. (It turned out a whisper command was easier and made more sense as the clients still receive other chats while getting whispers, something you couldn’t get while direct connected).

There is no multiple server support. It turned out to be significantly more complicated than time allowed. The essential idea, however, is that the announcements list will be shared between the servers, on checking for changes.

*To Continue:*

I would love for a simple gui to exist on the client. I would also love for there to be server interaction. Also, the way threads work now, as the server scales it becomes extremely memory intensive. I would look at ways for handling a set number of clients per thread and flesh out memory over time (As it stands, messages in the announcements are never deleted, which, over time will get too big for memory to hold. This could be fixed by clearing out based on time.)

**Appendix:**

*How to run the program:*

Run the server first by: Terminal

‘*java –jar “Chat App”.jar’*

Type server(*enter)*, then choose a port(*enter)* and name(*enter)*.

Leave server running and create clients by: Terminal (on same or different computer)

‘*java –jar “Chat App”.jar’*

Type client (*enter)*, then use same port as server(*enter)*, then type in the server’s address(*enter)*

In client’s terminal, you can test by doing the following:

Typing any command listed including:

-“/online” –reports who is connected to server

-“/whisper <client name> <message>” –sends a private message to ‘client name’

-“/commands” – reports all available commands (Through server)

-“/name <new name>” – changes your client name to new name

-“/log” – disconnects you from server (Client specific command)

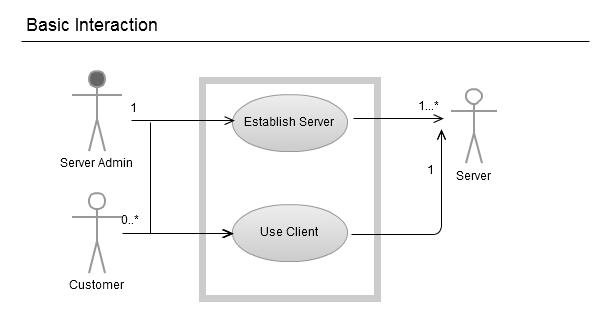
-“/server <server address> [port]” – moves you to server at desired address and port (client specific comman)

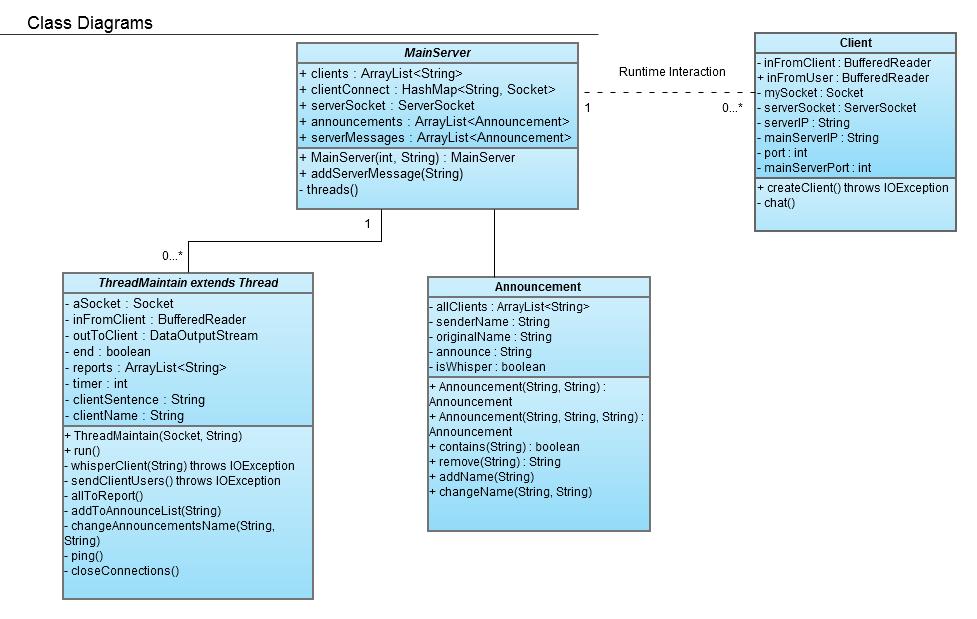
Typing anything other than the above will add to chat window for every connected client to view.

I fully recommend open multiple clients to ensure that the chat works.

(*note: when a client connects to a server, it is assigned a name “anonymous#<#>” where <#> is the total count of clients so far connected to the server. The /name commands makes it easier to communicate while preventing multiple clients with the same name)*

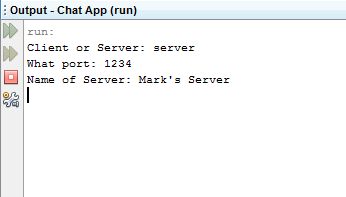
**Description-**

****

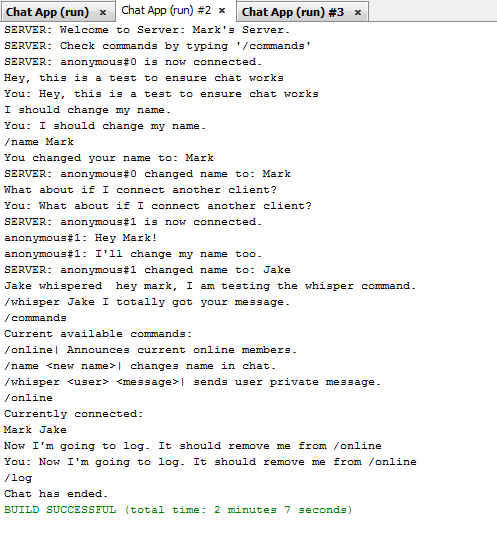
****

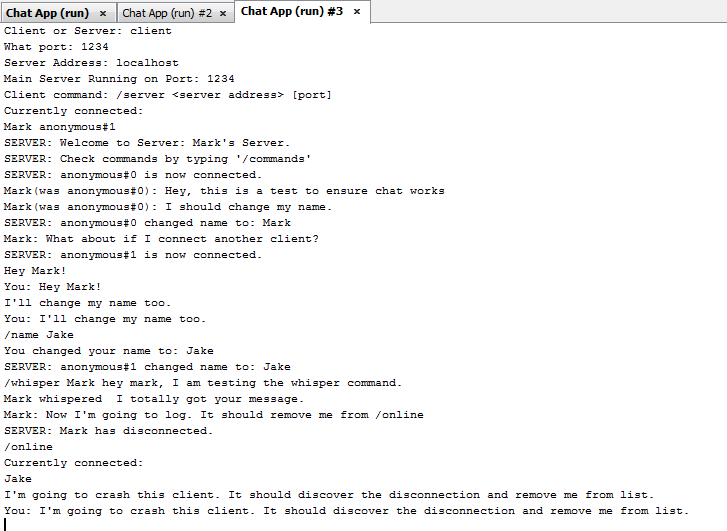
**Results Pictures:**

**Server-**

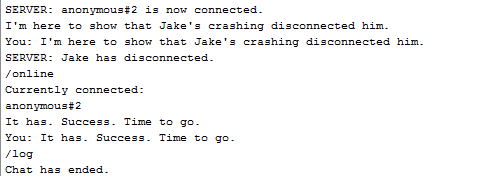
****

**Client 1-**

****

**Client2-**

**Client3-**

****