Project 3

Due: 28 April 2023

In this project, you'll implement a client for a networked system for playing 3-moku (details below). Because the server will handle the actual game logic, your focus will be on the threading and networking.

Objectives

In the course of this project, the successful student will:

- write a networked program using the TCP/IP network libraries;
- implement a byte-based application protocol; and
- use threads to build a responsive user interface.

The 3-moku game

You can fully complete the project without knowing anything at all about the rules of the game being played, but of course it will be easier to test if you can actually type in reasonable moves.

The game 3-moku (perhaps, "sanmoku"?) is a generalisation of tic-tac-toe, or a variant of gomoku (or a cousin of Connect-Four). Players take turn placing their mark—X or O, for our purposes—on their chosen grid squares, and whichever player first marks three in a row, in any direction, wins the game. For our project we'll play it on a 5x5 grid.

Spec: the user's view

The user will run the client program with two required arguments (server and port, in that order) and an optional third argument to specify a game number that they want to connect to. (If two people choose a game ID they can thus intentionally play against each other.) If they do not specify the third argument, they will be connected to any game on the server.

When they are playing the game, any time the server gives an indication that it's their turn to play, the client will print out the current state of the board and prompt the user to type a move. A suitable format for this view of the board would be like so:

```
  1 2 3 4 5
1 . . . . .
2 . x . . .
3 . . . x .
4 . . . o .
5 . . . . .
```

Exact details can vary, but note that row and column numbers are 1-based.

At any time, whether it is their turn or not, the user can type in any of the four commands available to them, with the following results:

`whose` causes the client to indicate whose turn it is

`stats` produces the IDs of both players and of the game, and indicates how many moves have been made so far (and any other statistics you would like to print as well)

`board` prints the current state of the board

`place` is followed by a row number and a column number (so, perhaps "`place 3 5`"), in that order. If it is the player's turn, this causes the client to transmit the player's move to the server; if it is not the player's turn, a message to that effect is printed. (This includes if the player is still awaiting confirmation from submitting their move—they shouldn't be able to validly enter their next move until either the opponent moves or they've gotten confirmation that a move was invalid.)

The client *may* accept other commands in addition, but in any case mustn't crash, hang, or exit if the input is unexpected or invalid.

When the server indicates that the game has ended (and terminates the connection), the client should print out the board state and who won, and exit the program.

Spec: the protocol

Upon connecting, the server will send to the client a two-byte player id. (This two-byte number, and all numbers in the remainder of the protocol, will be sent in network byte order.)

In response, the client sends two bytes indicating the player's preferred game id (if any). Transmitting 0x0000 indicates that any game is acceptable. The game ID 0xffff is reserved for future use (i.e. the user shouldn't pick that as their game ID).

The server will respond right away with the three bytes '`O`', '`K`', '`v`', and a fourth byte corresponding to the server version number. If the requested game isn't ready to start yet, it will quietly wait until a second player joins (the ok completes the "handshake" so to speak, so the client can sit tight). Then, when the server is ready to start the game with both players (or has decided it can't), it will send three 2-byte numbers:

- The game ID or zero (if zero is sent, indicating no available game, the server immediately disconnects)
- The user's player ID (again)
- Their opponent's player ID

Then the server will send an eleventh byte, which will be one of two values:

16	User is X, send first move
17	User is O, await opponent's move

From then on, any time it is the player's turn, the client will send one byte with the row number of their move and one byte with the column number (each with 1-based numbering, and *not* the ASCII digit). The server will send a one-byte response:

0	Invalid move, try again (still your turn)
33	Move ok, await opponent's next move
34	Game over
35	Opponent has terminated the game

When it is the opponent's turn (and they have made their move), the server will send one byte with the row number, one with the column number, and a third byte:

32	Send next move
34	Game over
35	Opponent has terminated the game

Whenever the server sends a 35 (game terminated), it then immediately terminates the connection. When it sends a 34 (game over), it follows it with one byte indicating who won:

16	X won
17	O won
18	Game ended in a draw

and then terminates the connection.

Other important notes

You may use either C or C++ in your implementation; you still may not rely on the `compile` command (use `gcc` or `g++` as appropriate). You may use a makefile but should document your instructions in any case.

It is subtle in the above instructions so I want to make it explicit here: in order for your code to let the user type commands at any time (like `stats`) and be responsive to that, but also print the server's response as soon as it arrives, you need to split up the work into two to three threads (making sure to avoid race conditions on printing things as well).

You need to locally maintain a model of the board, which is not transmitted as part of the protocol, but you do not need to do any game logic whatsoever. If the user enters an invalid move, the server will say so. If the game-ending conditions are met, the server will say so.

Prep work

Your initial prep work is to get your program to make the initial connection. Your prep work program should:

- Run without crashing when provided either two or three arguments
- Connect to the specified server and port
- Print the player ID it's been assigned
- Transmit the preferred game ID (or zero if none was specified)
- Stay connected at least until the OK message is sent.

(See below about testing.)

Hand in the prep work (as `proj3`) when you've got that working, no later than 4pm on Monday the 10th.

Design work

Once you've got the TCP connection working, you can start thinking about the larger system and communication design. In particular, you should write/draw:

1. a diagram showing the back-and-forth communication between one client and the server for the beginning of a possible game, up through the second move. *Every byte* that is transmitted in either direction should be accounted for.
2. a separate diagram illustrating the interaction between threads: starting somewhere in the middle of a game, it should show inbound data (user input, messages from the server) and outbound data (messages to the server, print statements), each relative to whichever thread is reading or writing it. How many threads do you plan on having? Which one is responsible for each read/write? How do they coordinate amongst themselves—what data needs to be passed around and/or shared?
3. your data design: what information do you need to store about the ongoing game?

Write your design work on paper (or do it on your laptop, but paper's probably easier for this) and bring it to class; this work is due **at the start of class** on Friday the 14th. If you're really stuck on something, do your best, make a note of it, and move on; we'll be discussing this extensively in class.

The server, and testing

I will generally leave a server running on port number 9908 on turing that can support up to five concurrent games (I may boost that later). You can point your client at that to test it: if "`turing`" and "`9908`" are the first two command line arguments to your program, your program should connect to that server.

If the server goes down (e.g. because of a crash, or a system update), or if it is full, or if it's *not* full and you want to more easily test that your program gracefully handles a full server, you can run the server yourself; it is installed as `3moku-server` on all the lab machines. If you run it yourself, please either run it on a different machine or with a different port number.

Final version

A full-credit final version will be a complete, non-buggy, working implementation of the game client TOGETHER WITH convincing proof that it is correct. The program should be able to run with arbitrary input without crashing even if the user gives *bad* input, and preferably will recover and continue accepting input rather than terminating the connection (and thus the game). The "proof" will take the form of a clean and complete set of test cases, including both input/running instructions and expected results.

Note that I am not able to spend a ton of time with your program (and in fact may not read it at all other than to verify the threading stuff, and definitely won't do your debugging for you), so your documentation will need to tell me anything I need to know to run and test your program. There need to be clear instructions on how to run it in general as well as how to run each/all of the tests and quickly verify that they ran correctly (and which rubric items each one corresponds to). Having complete and correct documentation is an easy 10 points, but if your documentation omits important info or tells me the wrong thing, you'll get less than full credit there.

(10 points), there remain 70 points in the rubric, which will be awarded according to the table below.

NOTE: if your code doesn't compile, or immediately crashes when it's run, you will get zero of these points. Don't let this happen to you!

RUBRIC

Server communication, setup	
10	Client runs with 2-3 arguments, connects to server on specified host and port, receives assigned player ID, sends requested game number (i.e. the prep work).
5	Player IDs and game ID are received and sent in network byte order
2	Gracefully terminates if there is no available game
User interface	
5	Accepts all four user commands and responds to them
5	On non-place commands, responds with correct current info
2	Gracefully responds to invalid input (i.e. don't crash or hang)
Game state storage and display	
2	Displays board with current state in reasonable layout, any time it's supposed to
2	Board shows all moves made by both players up to this point
2	Displays correct character (X/O) on displayed board
Server communication, during game	
5	Sends place move with correct format to server
2	Awaits and receives OK from server and updates accordingly (updates board, now opponent's move)
5	Awaits and receives opponent's move from server and updates accordingly
2	Responds correctly to Invalid (still user's move, no state update)
2	Responds correctly to Game Over (after either player)
2	Gracefully terminates if opponent terminates game
Threading	
5	Some/all of the code runs in a separate thread
5	Threading guarantees user's interface is not blocked when waiting for opponent's move, tells user it's not their turn if they try to place at the wrong time
2	Thread sync variables (semaphore, mutex, etc) are init'ed and available to all necessary threads
5	Threads synched so shared variables, network communications, and screen access are all reliably accessible, without race conditions

Handing in

For both the prep work and the final version, hand it in as `proj3` using the handin script. The final version is due at 4pm on Friday, 28 April.