CMPUT 379, Assignment 2, Winter 2018

(Sockets, Synchronization, Message Passing)

Objective

You are asked to implement a client/server game on a two-dimensional NxN square "grid world". Developing the solution requires that you get familiar with programming using the sockets API and posix threads, both of which will be covered in the labs. The specification of the assignment includes a suggestion about the protocol between client and server and a minimum of client-side and server-side behavior/interaction. You are free to make additional design decisions to address a number of details left unspecified, as long as you properly justify the rationale behind them and explain their anticipated impact. However, a major constraint you need to satisfy, which is also present in all networked games, is that the state of the game is maintained and updated by the server, *not* by the clients. The game state is broadcast at periodic intervals from the server to all connected clients. As such, the logic of the game is implemented at the server while the clients are just facilitating user input and the rendering of the game state received from the server.

Game Logic

A long-running server process maintains the game state and periodically sends updates of the game board to all connected clients. During the game, the position of each player changes in response to commands sent by the user. No more than one command per client will be handled by the server in each update period. There are two kinds of commands: movement and fire.

When a client is connected to the server, the first (initialization) step is for the server to send to the client (a) the dimensions of the board, (b) a unique number which will be the "player ID" of the client, and (c) a uniformly random location within the grid where the server places the new player/client. Note that (a)-(c) is information coming from the server -- *not* decided by the client processes.

Each single movement command instructs the server to move the location of the corresponding player by *one* grid point left/right/up/down (depending on the direction of move specified by the client) within the grid. We repeat: each move command results in a single grid point move -- that is, it is *not* a "change direction" command for a constant-speed moving object. The server receives the commands from all clients and updates the location of the player(s). The server side enforces the boundaries of the square, so the player cannot move outside the board. For example, if a player is at the right edge and tries to move right, the movement will not happen, i.e. it will remain stuck at the same location. The player can also send a "fire" command which kills any player located up to two grid points "in front of it" in the direction of its most recent move. So, for example if the player moved to the right and then it issued a "fire" it can "kill" any player located in the two grid points to the right of its current location.

Additionally, the server game logic should not allow two players to move into the same grid point. If two (or more) players try to move to the same grid point, those move commands are ignored. Note that when a new player client connects to the server, the server places the new player at an initial position which is chosen in a random uniform manner across all the grid points that do not have a player on them yet (but the chosen grid point *could* be the within the "kill" range of a player). The default initial direction of a player is "up".

A player that is killed, receives from the server a termination message with the score for how many kills this player has achieved. The connection to the client is then terminated.

Minimal Interaction/Interface Requirements

The client/player should be able to tell which way they are "facing" as this matters to which positions they can fire at. We recommend an "ASCII art" interface using at least the use of the four symbols `<` `>` `^` and `v` to indicate the direction each player faces. The server state being periodically broadcast to all clients includes the ID of each player and their location and direction they are facing on the board. Since the identity of the other players does not matter, a minimal rendering at the client is to use a boldface symbol for "self" and non-bold characters for all the other players.

In addition to up/down/left/right commands, a "fire" command must also be supported by the client input and sent to the server. Note that the effect of a fire command is decided by the server (see also the notes section regarding multiple commands received from a client -- this is the reason we cannot tell at the client that a fire has actually happened). The fire(s) that have actually happened in an interval are broadcast by the server as part of the game grid update broadcast to all clients.

The rendering of all fire commands taking place in the same frame (since there could be as many "fire" as clients connected) should be in the form of two of the same characters (we recommend `o`) on the path of the kill. So it would look as `oo` if it is a left/right fire. If "self" is firing, then the path of the kill will be in boldface, e.g., `<b>oo</b>`

The client continuously reads user input and sends the corresponding commands to the server. For the sake of uniformity in testing, use the `i` (up), `j` (left), `k` (down), `l` (right) character convention for movement and use the space bar to fire. An extra character, `x` should trigger the orderly client-side termination. It should result in the client receiving the final score that it had achieved and the termination of the connection to the server.

Command Line

The server execution should start by invoking the executable `gameserver379` which should subsequently became a daemon server process. This "daemonization" will be described in the labs and it involves the so-called "double fork" trick. An example invocation is:

```
gameserver379 16 0.2 8989 1547543314
```

where the first parameter is the dimension of the square grid (`16` means a 16x16 grid), followed by the update period (in seconds) the server uses to periodically send the game state to all clients (`0.2` meaning a fifth of a second in this example), followed by the port on which the server should be listening (`8989` in this example) and the last number is the "seed" for the random number generator (`1547543314` in this example). The use of the seed will be explained in the lab. It essentially allows the repeatability of the random generation of the initial points for each connecting client (same seed -> same (pseudo)random numbers).

The server should terminate gracefully when it receives `SIGTERM`.

The client execution is simply

```
gameclient379 127.0.0.1 8989
```

where the first parameter is the hostname OR the IP address of the host on which the server is running and the second is the port number where the server listens.

Design Issues and Questions

The deliverable includes a design document (`DESIGN.txt` or `DESIGN.md`) which will have to address the following broad questions, outlining the specific solution strategy, and pointing to the implementation details necessary (e.g. by referencing particular functions in your code) and what are the implications of your design decisions.

- How do you ensure that input response time (keystroke read, transmission of command, receiving at the server) is minimized?
- How do you minimize the amount of traffic flowing back and forth between clients and server?
- How do you handle multiple commands being sent from a client between two successive game board updates? (see also notes)
- How do you ensure that the same (consistent) state is broadcast to all clients?
- How do you ensure the periodic broadcast updates happen as quickly as possible?
- How do you deal with clients that fail without harming the game state?
- How do you handle the graceful termination of the server when it receives a signal? (Provide and implement a reasonable form of graceful termination.)
- How do you deal with clients that unilaterally break the connection to the server? (Assume the client can exit, e.g. crash, without advance warning.)
- How can you improve the user interface without hurting performance?

Notes

To address the fast response to user input, you will need to both deal with "raw" input issues (keystroke-at-a-time) as well as with networking (sending things quickly through sockets) issues.

A basic approach to deal with multiple commands received from a client between two successive game board updates is to have the server pick only one of them, i.e., to enforce one-command-per-client-per-period. It is highly recommended that the server selects the last (most recent) command received from a client during the interval. For example, if the player on one client managed to type `jjk k` (left, left, down, fire, down) then the server will only consider the last one received (down) and the previous moves will be discarded.

To produce "ASCII graphics" you should use the `ncurses` library. Simple examples of using `ncurses` can be found online and some such examples will be presented in the labs. The `ncurses` libraries are already installed on the lab machines but not on the VM. Information for installation on the VM will be provided in the labs.

Sample Screen

The following is a slightly decorated screen representing the game state seen at a client in a game involving four players (the boundary is extra, providing a frame to a 10x10 grid)

```
+----------+
|   v      |
|          |
|      >   |
|          |
|          |
|          |
|          |
|    >     |
|    o     |
|    o     |
|    ^     |
+----------+
```

Note the awkward aspect ratio. This client is at the location of the boldface character being fired at by the one at the bottom.

Deliverables

The language of implementation is C and in particular the C99 style.

In the lab machines, always remember to terminate any server daemon processes you have left running before leaving. Generally, ensure that you have not left any stray processes running on your machine before leaving. Violations of this rule may be penalized with mark deductions.

Your assignment should be submitted as a single compressed (`zip` or `tar.gz`) archive file. The archive should contain all necessary source files as well as a `Makefile`. Calling `make` without any arguments should generate both the client and the server executables. Your submission must include the `DESIGN.md` (plain text or markdown format) document addressing the design issues. If you have non-obvious options to the user interface, beyond the minimal ones indicated here, please provide an additional document, `UI.md`, documenting the interface. A short file `README.md` (plain text or markdown markup) included with the archive to describe how you balanced the work across the two group members. Also indicate whether your code was tested (and running) on the VM or on the physical hosts in lab CSC 219.

You are expected to deliver good quality, efficient, code. That is, **the quality of your code will be marked**.

Monday, February 5th, 2018