CSCB09 2026 Summer Assignment 4

Due: August 6 11:59PM

This assignment is on network stream socket programming and I/O multiplexing (I/O with multiple targets concurrently).

As usual, you should aim for reasonably efficient algorithms and reasonably well-organized, well-factored, comprehensible code.

Multiplayer Online Asynchronous Battleship (MOAB)

You will implement a game server for Multiplayer Online Asynchronous Battleship!

This game is played on a 10x10 board; cell coordinates are 0-based, i.e., the corners are (0,0), (0,9), (9,0), (9,9). Each player owns a 5x1 or 1x5 ship on the board; its location is not revealed to other players until it's hit. Each player can bomb any cell on the board, causing a hit to every ship that occupies that cell.

Any player can join (connect to the game server) any time. When a player joins, there is a registration phase: The player registers their name and ship location. It is OK if multiple players have overlapping ships. (I can make up a sci-fi explanation, but the real reason is simplicity and comedy.) The game server broadcasts the announcement that the player has joined.

Any registered player can bomb any cell at any time. The result is that every ship that occupies that cell is marked as getting hit at that cell. The game server broadcasts a report to all registered players: who bombs which cell, and whose ships are hit (or that the bombing is a total miss).

From the perspective of a ship, we only track which cells have been marked as hit. We don't count how many more times the same cell has been hit.

When a ship has all 5 cells marked as hit, it disappears, and the owner player loses the game: The game server broadcasts who has lost; then the game server deregisters the player and disconnects the connection.

Any player can leave (disconnect) any time. The game server treats it the same as losing.

Message Syntax

Notation

I use the following notation—borrowed from `printf()`—to describe message syntax:

`(format_string, parmeter, ...)`

Example: If I write

`("MISS %s %d %d\n", attacker, x, y)`

then an example message is

`MISS albert 2 4` followed by newline.

Tips: You can basically use these format strings in `fprintf()`, `snprintf()`, `sscanf()`, etc.

Note: We do not send/expect a NUL byte over the network; the last byte of a message is newline.

Real-world note: Many network protocols use a plain text format and ends every line/message with carriage return then newline, i.e., following the Windows convention ironically! Examples are web, email, and IRC. In this assignment we keep it simple and use just newline.

Messages

- Registration: The player sends `("REG %20s %d %d %c\n", name, x, y, d)`.

  name is at most 20 bytes; each only a letter, a digit, or `-` (the minus sign)

  (x, y) specifies the location of the centre of the ship.

  d is `'-'` to mean 1x5 or `'|'` to mean 5x1.

  Example: If $(x, y) = (3, 4)$ and $d =$ `'-'`, then the ship occupies (1,4), (2,4), (3,4), (4,4), (5,4).

  Example: If $(x, y) = (3, 4)$ and $d =$ `'|'`, then the ship occupies (3,2), (3,3), (3,4), (3,5), (3,6).

  The whole ship—all 5 cells—must be within the bounds of the board. Example: $(x, y) = (0, 1)$ and $d =$ `'-'` is out of bounds because one cell is (-1,1).

  If syntax error or any of the above conditions is not met, the server replies `("INVALID\n")`. The player may try again with another REG message.

  Else, if the name is already in use by another registered player, the server replies `("TAKEN\n")`. The player may try again with another REG message.

  Else, the registration is complete. The server replies `("WELCOME\n")`. The server also proceeds to...

- Server join broadcast: `("JOIN %s\n", name)`

  Whenever a player registration is complete, the server broadcasts this to all registered players (therefore including the newly registered player).

- Player bomb request: `("BOMB %d %d\n", x, y)`

  (x, y) specifies the cell to bomb.

  If syntax error, the server replies `("INVALID\n")`.

  We do not invalidate out-of-bound errors. If a player is dumb enough to bomb (-1, 67), let them miss!

- Server hit report: `("HIT %s %d %d %s\n", attacker, x, y, victim)`

  For each bombing, for each ship hit, the server broadcasts this message to all registered players. *attacker* is the player who sent the BOMB message, (x, y) is the bombed cell, and *victim* is the owner of a ship hit.

  Note that one bombing may result in multiple HIT messages with the same attacker and (x, y), just with different victims.

- Server miss report: `("MISS %s %d %d\n", attacker, x, y)`

  For each bombing that hits no ship, the server broadcasts this message instead.

- Server GG broadcast: `("GG %s\n", player)`

  Whenever a registered player loses or disconnects, the server broadcasts this message to all registered players.

Dealing with Bad Clients

Client malfunctions happen all the time: We expect the Internet to be full of fools, trolls, and bad actors. Here are the scenerios you should handle as prescribed:

- Premature disconnection: If the player is registered, this is covered above. If the registration phase is not even complete, just discard.
- Sending a message to a client causes an error, SIGPIPE, or would block: For simplicity, disconnect the client, and treat as premature disconnection above.

  Possible causes: The client disconnects; or it is not reading your messages (and therefore the OSes on both sides have a huge backlog).

- Invalid messages: This is mostly covered above, except...
- Message too long: Even the longest valid client message has a maximum possible length. We will assume this: If the server receives more than 100 bytes without a newline, then disconnect the client because we think it is not working (or worse).

Debugging And Error Messages

If you like to print debugging or error messages for your own sake, please send them to stderr only.

What/How to Hand in

Your program should take one command line argument: a port number. Your server should `bind()` to that port number at `INADDR_ANY`.

Unlike past assignments, this time I encourage you to organize your code into modules. You may upload multiple *.c and *.h files. Please do not upload a zip file, unless: Check the checkbox "unzip all zip files in place".

Automarking will simply compile with `gcc -O2 *.c`.

Sample Clinets and Servers

I will have sample clients and servers (exe only) available on Mathlab in
/courses/courses/cscb09s26/laialber/a4/sample

Testing Tips

Randomize Port Number

When you run a server on Mathlab, since other students are doing the same, you should randomly choose a port number between 1024 and 65535. I recommend basing it on your student number: Try

```
student_number % 64512 + 1024
```

If `bind()` gives an "address in use" error, add 1 and try again, etc.

Manual Testing by `nc`

The `nc` program can let you manually act as one side to hand-test the other side. You enter to stdin what to send; you see received data on stdout. Quickstart:

- To act as a client: `nc [-v] [-q 1] HOST PORT`

  HOST can be a domain name or the dot notation of an IP address.

- To act as a server: `nc [-v] [-q 1] -l PORT`.

  Note that this calls `accept()` only once, at the beginning. It only serves one client, then quits.

Mathlab Server, PC Client

Mathlab is behind a firewall. A firewall blocks most ports for safety, including ports we need for testing this assignment. ssh can help get past the firewall.

If you have a server running on Mathlab at port sssss, e.g.:

```
mathlab$ /path/to/server sssss
```

Then "ssh local forwarding" allows you to run a client on your PC. Pick a random port number xxxxx (criterion: available on your PC). Then the ssh command goes like:

```
pc$ ssh -L xxxxx:127.0.0.1:sssss utorid@mathlab.utsc.utoronto.ca
```

Tell your client on your PC that the server address and port are:

```
pc$ /path/to/client 127.0.0.1 xxxxx
```

PC Server, Mathlab Client

Your home router has a firewall; Windows adds an extra one. A firewall blocks most ports for safety, including ports we need for testing this assignment. ssh can help get past the firewalls.

If you have a server running on your PC at port sssss, e.g.:

```
pc$ /path/to/server sssss
```

Then "ssh remote forwarding" allows you to run a client on Mathlab. Pick a random port number xxxxx (criterion: available on Mathlab). Then the ssh command goes like:

```
pc$ ssh -R xxxxx:127.0.0.1:sssss utorid@mathlab.utsc.utoronto.ca
```

Tell your client on Mathlab that the server address and port are:

```
mathlab$ /path/to/client 127.0.0.1 xxxxx
```