Problem Set 3: Multiplayer Minesweeper

Beta due	Monday, March 31, 2014, 10:00 PM
Code reviews due	Wednesday, April 2, 2014, 10:00 PM
Final due	Monday, April 7, 2014, 10:00 PM

The purpose of this problem set is to explore multithreaded programming with a shared mutable data type, which you should protect using synchronization.

Design Freedom and Restrictions

On this problem set, you have substantial design freedom. However, please do not use any external jar files.

Your solution **must not** change the name, method signature, class name, package name, or specification of the `MinesweeperServer` methods `main()` or `runMinesweeperServer()`.

Also note that the axis definition of your board must match what is defined in "Protocol Specification" section; the (x,y) coordinates start at (0,0) in the top-left corner, extend horizontally to the right in the X direction, and vertically downwards in the Y direction.

Your code will be tested automatically; changing these interfaces or axes will cause our testing suite to fail. Failing the test suite means you get 0 points for your submission: beta, final, or otherwise.

Beyond this requirement you have complete design freedom. For example, you can add new methods, classes, and packages, and rename or delete classes other than MinesweeperServer.

Get the code

To get started, pull the problem set code from Athena using Git. As a reminder, your Git repository for this problem set can be found at:

`/afs/athena.mit.edu/course/6/6.005/git/sp14/psets/ps3/[Athena username].git`

If you need a refresher on how to clone your Git repository, see Problem Set 0 (../ps0/#clone).

Overview

You will start with some minimal server code and implement a server and thread-safe data structure for playing a multiplayer variant of the classic computer game "Minesweeper."

You can review the traditional single-player Minesweeper concept rules on Wikipedia: Minesweeper (video game) (http://en.wikipedia.org/wiki/Minesweeper_%28video_game%29)

You can try playing traditional/single-player Minesweeper here (http://minesweeperonline.com/).

Note: You may notice that the implementation in the latter link above does something subtle. It ensures that there's never a bomb where you make your first click of the game. You should *not* implement this for the assignment. (It would be in conflict with giving the option to pass in a pre-designed board, for example.)

The final product will consist of a server and no client; it should be fully playable using the `telnet` utility to send text commands directly over a network connection (see further below).

Notes

We will refer to the board as a grid of squares. Each square is either 'flagged', 'dug', or 'untouched'. Each square also either contains a bomb, or does not contain a bomb.

Our variant works very similarly to standard Minesweeper, but with multiple players simultaneously playing on a single board. In both versions, players lose when they try to dig an untouched square that happens to contain a bomb. Whereas a standard Minesweeper game would end at this point, in our version, the game keeps going for the other players. In our version, when one player blows up a bomb, they still lose, and the game ends for them (i.e. the server ends their connection), but the other players may continue playing. The square where the bomb was blown up is now a dug square with no bomb. The player who lost may reconnect to the same game again via `telnet` to start playing again.

Note that there are some tricky cases of user-level concurrency. For example, say user A has just modified the game state (i.e. by digging in one or more squares) such that square i,j obviously has a bomb. Meanwhile, user B has not observed the board state since this update has taken place, so user B goes ahead and digs in square i,j. Your program should allow the user to dig in that square–a user of Multiplayer Minesweeper must accept this kind of risk.

We are not specifically defining, or asking you to implement, any kind of "win condition" for the game.

Before you begin x

`telnet` is a utility that allows you to make a direct network connection to a listening server and communicate with it via a terminal interface. Before starting this problem set, please ensure that you have `telnet` installed. *nix operating systems (including Mac OS X) should have `telnet` installed by default.

Windows users should first check if telnet is installed by running the command `telnet` on the command line. If you do not have it, you can install it via Control Panel → Programs and Features → Turn windows features on/off → Telnet client.

You can have `telnet` connect to a host/port (for example, `web.mit.edu:80`) from the command line with

```
telnet web.mit.edu 80
```

Since port 80 is usually used for HTTP, we can now make HTTP requests through `telnet`. If you now type

```
GET /
```

`telnet` should retrieve the HTML for the webpage at web.mit.edu. If you want to connect to your own machine, you can use `localhost` as the hostname and whatever port your server is listening on. With the default port of 4444 in this problem set, you can connect to your Minesweeper server with

```
telnet localhost 4444
```

Protocol and specification

You must implement the following very precise protocol for communication between the user and the server, as specified below.

Messages from the user to the server

Each message starts with a message type and arguments to the message are seperated by a single SPACE character and the message ends with a NEWLINE. The NEWLINE can be either a single character “\n” or two character sequence “\r\n”

The user can send the following messages to the server:

LOOK message

> The message type is the word “look” and there are no arguments.
>
> Example: | l | o | o | k | \n |
>
> Returns a BOARD message, a string representation of the board’s state. Does not mutate anything on the server. See the section below “Messages from the server to the user” for the exact required format of the BOARD message.

DIG message

The message is the word “dig” followed by two arguments, the X and Y coordinates. The type and the two arguments are seperated by a single SPACE.

Example:

d	i	g		3		1	0	\r	\n

The dig message has the following properties:

1. If either x or y is less than 0, or either x or y is equal to or greater than the board size, or square x,y is not in the ‘untouched’ state, do nothing and return a BOARD message.
2. If square x,y’s state is ‘untouched’, change square x,y’s state to ‘dug’.
3. If square x,y contains a bomb, change it so that it contains no bomb and send a BOOM message to the user. Then, if the debug flag is missing (see Question 4), terminate the user’s connection. See again the section below for the exact required format of the BOOM message. Note: When modifying a square from containing a bomb to no longer containing a bomb, make sure that subsequent BOARD messages show updated bomb counts in the adjacent squares. After removing the bomb continue to the next step.
4. If the square x,y has no neighbor squares with bombs, then for each of x,y’s ‘untouched’ neighbor squares, change said square to ‘dug’ and repeat *this step* (not the entire DIG procedure) recursively for said neighbor square unless said neighbor square was already dug before said change.
5. For any DIG message where a BOOM message is not returned, return a BOARD message.

FLAG message

The message type is the word “flag” followed by two arguments the X and Y coordinates. The type and the two arguments are seperated by a single SPACE.

Example:

f	l	a	g		1	1		8	\n

The flag message has the following properties:

1. If x and y are both greater than or equal to 0, and less than the board size, and square x,y is in the ‘untouched’ state, change it to be in the ‘flagged’ state.
2. Otherwise, do not mutate any state on the server.
3. For any FLAG message, return a BOARD message.

DEFLAG message

The message type is the word “deflag” followed by two arguments the X and Y coordinates. The type and the two arguments are seperated by a single SPACE.

Example:

d	e	f	l	a	g		9		9	\n

The flag message has the following properties:

1. If x and y are both greater than or equal to 0, and less than the board size, and square x,y is in the ‘flagged’ state, change it to be in the ‘untouched’ state.
2. Otherwise, do not mutate any state on the server.
3. For any DEFLAG message, return a BOARD message.

HELP_REQ message

The message type is the word “help” and there are no arguments.

Example: `help\r\n`

Returns a HELP message (see below). Does not mutate anything on the server.

BYE message

The message type is the word “bye” and there are no arguments.

Example: `bye\r\n`

Terminates the connection with this client.

For any message from the server which does not match the server-to-user message format as given, do nothing, discard the data until the NEWLINE is reached, and start processing the next message.

Messages from the server to the user

There are no message types in the messages sent by the server to the user. The server sends the HELLO message as soon as it establishes a connection to the user. After that, for any message it receives that matches the user-to-server message format, other than a BYE message, the server should always return either a BOARD message, a BOOM message, or a HELP message.

For any message from the user which does not match the user-to-server message format as given, discard the incoming message and send a HELP message as the reply to the user.

The action to take for each different kind of message is as follows:

HELLO message

The message should print the message *Welcome to Minesweeper. Board: X columns by Y rows. Players: N including you. Type ‘help’ for help.* where *N* is the number of users currently connected to the server and the board is *X* × *Y*. This message should be sent to the user exactly as defined and only once, immediately after the server connects to the user. Again the message should end with a NEWLINE.

Example:

```
Welcome to Minesweeper. Board: 8 columns by 8 rows.
 Players: 12 including you. Type 'help' for help.\r\n
```

BOARD message

The message has no start type word and consists of a series of newline-separated rows of space-separated characters, thereby giving a grid representation of the board's state with exactly one char for each square. The mapping of characters is as follows:

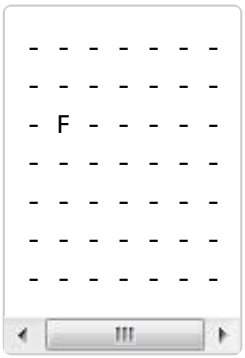
- "-" for squares with state 'untouched'.
- "F" for squares with state 'flagged'.
- " " (space) for squares with state 'dug' and 0 neighbors who have a bomb.
- integer COUNT in range [1-8] for squares with state 'dug' and COUNT neighbors that have a bomb.

Here is an example board message with 3 rows and 8 columns:

				2	-	-	\r \n
1	1		1	F	-	-	\n
F	1		1	-	-	-	\n

Notice that in this representation we reveal every square's state of 'untouched', 'flagged', or 'dug', and we indirectly reveal limited information about whether some squares have bombs or not.

In the printed **BOARD** output, the (x,y) coordinates start at (0,0) in the top-left corner, extend horizontally to the right in the X direction, and vertically downwards in the Y direction. (While different from the standard geometric convention for IV quadrant, this happens to be the protocol specification.) So the following output would represent a flagged square at **(x=1,y=2)** and the rest of the squares being untouched:



In order to conform to the protocol specification, you'll need to preserve this arrangement of cells in board while writing the board messages. If you change this order in either writing the board message, or reading the board from file (Problem 4) your implementation will fail in autograding.

BOOM message

The message is the word *BOOM!* followed by a NEWLINE.

Example:

B	O	O	M	!	\r	\n

The last message the client will receive from the server before it gets disconnected!

HELP message

The message is a list of non-NEWLINE characters (your help message) followed by NEWLINE.

Example: R T F M ! \n

Unlike the above example, the HELP message should print out a message which indicates all the commands the user can send to the server. The exact contents of this message is up to you – but it is important that this message not contain multiple NEWLINEs.

Formal Grammars for the messages

The following is a precise and comprehensive description of the messages using grammar notation.

- Wikipedia description of the grammar notation (http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form)
- Wikipedia description of the right hand side expressions of these statements, which uses regular expression format (http://en.wikipedia.org/wiki/Regular_expression)

Grammar for messages from the user to the server:

```
MESSAGE ::= ( LOOK | DIG | FLAG | DEFLAG | HELP_REQ | BYE ) NEWLINE
LOOK ::= "look"
DIG ::= "dig" SPACE X SPACE Y
FLAG ::= "flag" SPACE X SPACE Y
DEFLAG ::= "deflag" SPACE X SPACE Y
HELP_REQ ::= "help"
BYE ::= "bye"
NEWLINE ::= "\r"? "\n"
X ::= INT
Y ::= INT
SPACE ::= " "
INT ::= [0-9]+
```

Grammar for messages from the server to the user:

```
MESSAGE ::= BOARD | BOOM | HELP | HELLO
BOARD ::= LINE+
LINE ::= (SQUARE SPACE)* SQUARE NEWLINE
SQUARE ::= "-" | "F" | COUNT | SPACE
SPACE ::= " "
NEWLINE ::= "\r"? "\n"
COUNT ::= [1-8]
BOOM ::= "BOOM!" NEWLINE
HELP ::= [^\r\n]+ NEWLINE
HELLO ::= "Welcome to Minesweeper. Board: " X " columns by " Y " rows. Players: " N
          " including you. Type 'help' for help." NEWLINE
X ::= INT
Y ::= INT
N ::= INT
INT ::= [0-9]+
```

Problem 1: setup the server to dealw ith multiple clients

We have provided you with a single-thread server which can accept connections with one client at a time, and which includes code to parse the input according to the client-server protocol above. Modify the server so it can maintain multiple client connections simultaneously. Each client connection should be maintained by its own thread. You may wish to add another class to do this. You may continue to do nothing with the parsed user input at this time.

Problem 2: implement a data structure for Minesweeper

Specify, implement, and test a data structure for representing the Minesweeper board (as a Java type, without using sockets or threads). Create a `Board` class as well as any additional classes you might need to accomplish this.

Your `Board` class must have specifications for all methods, a rep invariant written as a comment, and a rep invariant written as a `checkRep()` method called by your unit tests.

Problem 3: make the entire system thread-safe

a. Come-up with a strategy to make your system thread safe. As you learned in the lectures, there are multiple ways to make a system thread safe. For example, this can be done by:
    - using immutable or thread-safe data structures
    - using a queue to send messages that will be processed sequentially by a single thread
    - using synchronized methods or the synchronized keyword at the right places
    - or combination of these techniques
b. In a comment at the top of `Board.java`, document the thread-safety argument for your board data structure. Depending on your design, the board may ***not*** be thread-safe. If so, document that. You should also document your system thread safety argument at the top of `MinesweeperServer.java`.
c. When you implement the server in problem 5, make sure that you clearly comment the code where your strategy is implemented.

Problem 4: initialize the board based on command-line options

We want our server to be able to accept some command-line options. The exact specification is given in the Javadoc for `MinesweeperServer.main()`, which is excerpted below:

Usage:

```
MinesweeperServer [--debug] [--port PORT] [--size SIZE_X,SIZE_Y | --file FILE]
```

The debug argument means the server should run in debug mode. The server should disconnect a client after a BOOM message if and only if the debug flag argument was not given. E.g. `MinesweeperServer --debug` starts the server in debug mode.

PORT is an optional integer in the range 0 to 65535 inclusive, specifying the port the server should be listening on for incoming connections. E.g. `MinesweeperServer --port 1234` starts the server listening on port 1234.

SIZE is an optional integer argument specifying that a random board of size SIZE_X*SIZE_Y should be generated. E.g. `MinesweeperServer --size 42,69` starts the server initialized with a random board of size 42*69.

FILE is an optional argument specifying a file pathname where a board has been stored. If this argument is given, the stored board should be loaded as the starting board. E.g. `MinesweeperServer --file boardfile.txt` starts the server initialized with the board stored in boardfile.txt, however large it happens to be.

We provide you with the code required to parse these command-line arguments in the `main()` method already existing in `MinesweeperServer`. You should not change this method. Instead, you should change `runMinesweeperServer` to handle each of the two different ways a board can be initialized: either by random, or through input from a file. (You'll deal with the debug flag in Problem 5.)

For a **SIZE** argument: if the passed-in size X,Y > 0, the server's Board instance should be randomly generated and should have size equal to X by Y. To randomly generate your board, you should assign each square to contain a bomb with probability .25 and otherwise no bomb. All squares' states should be set to 'untouched'.

For a **FILE** argument: If a file exists at the given PATH, read the the corresponding file, and if it is properly formatted, deterministically create the Board instance. The file format for input should be:

```
FILE ::= BOARD LINE+
BOARD := X SPACE Y NEWLINE
LINE ::= (VAL SPACE)* VAL NEWLINE
VAL ::= 0 | 1
X ::= INT
Y ::= INT
SPACE ::= " "
NEWLINE ::= "\r"? "\n"
```

In a properly formatted file matching the FILE grammar, each line must contain X values and there should be exactly Y + 1 lines (first line is the board size). If the file read is properly formatted, the Board should be instantiated such that square i,j has a bomb if and only if the i'th VAL in LINE j of the input is 1. If the file is improperly formatted, your program should throw an unchecked exception (either RuntimeException or define your own). The following example describes a board with 4 columns and 3 rows with a single bomb at the location x=2, y=1.

Example:

4	3	\n		
0	0	0	0	\n
0	0	1	0	\n
0	0	0	0	\n

If you were running your server from the command line and the executable was called ‘server’, some command-line arguments might look like:

```
./server --debug
./server --port 1234
./server --size 123,420
./server --file ../testBoard
./server --debug --port 1234 --size 20,14
```

You may specify command-line arguments in Eclipse by clicking on the drop-down arrow next to “run,” clicking on “Run Configurations…”, and selecting the “Arguments” tab. Type your arguments into the text box.

Implement the `runMinesweeperServer` method so that it handles the two possible ways to initialize a board (randomly or by loading a file).

See tips below for useful methods for reading from a file.

Errata

As announced on Stellar (https://stellar.mit.edu/S/course/6/sp14/6.005/announce.html#ann1395146893), the starting code includes a file `MinesweeperServer.javae` – notice the `e` at the end of the file name. You should delete this file from your repository. Changes you make to it will not be considered for grading.

As announced on Stellar (https://stellar.mit.edu/S/course/6/sp14/6.005/announce.html#ann1395169262), the comment on `main` in `Minesweeper.java` included incorrect specs for the command-line arguments and board file grammar.

The correct specs appear above in this handout.

If you want, you may download and apply an official patch (errata).

Problem 5: putting it all together

a. Modify your server so that it implements our protocols and specifications, using a single instance of `Board`.

   Note that when we send a BOOM message to the user, we should terminate their connection if and only if the debug flag (see Problem 4) is missing.

   The tips below may be useful for reading from and writing to socket streams.

b. Near the top of your `MinesweeperServer.java` source file, include a substantial comment with an argument for why your server is thread-safe.

Commit to Git. Pushing your code to your Git repository on Athena will cause Didit to run a single staff-provided test. You can read and download the **source code for the published test** (https://github.com/mit6005/S14-PS3).

Tips

- `BufferedReader.readLine()` (http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html#readLine%28%29) reads a line of text from an input stream, where line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.
- You can wrap a `BufferedReader` around both `InputStreamReader` and `FileReader` objects
- `PrinterWriter.println()` (http://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html#println%28%29) prints formatted text to an output stream, and terminates the current line by writing the line separator string. The line separator string is defined by the system property line.separator, and is not necessarily a single newline character ('\n').
- You can wrap a `PrinterWriter` around a `Writer` or `OutputStream` objects

Beyond Telnet

You are now done with the problem set, but you could imagine not having to use `telnet` to play Minesweeper. A GUI would do a lot to bring your program to life. Here is GUI client from past student **Robin Cheng** that you can use to play your Minesweeper game: Minesweeper GUI (./minesweeperGUI.jar)

This GUI waits until you do a LOOK or other operation before updating the view.

It has UI for spawning clients, so you can actually just run the jar once and then use the UI to spawn a number of clients.

It also displays a printout of all the protocol I/O which is going on, making it particularly helpful for debugging.

You should have your server running before trying to start a client connection from this GUI.

Note: This GUI is unofficial implementation. You may find a bug or incorrect behavior in it in some ways. Do not consider it enough for testing your code.

If you want practice writing GUIs, we encourage you to write your own Minesweeper GUI and email the source to the staff.

Submitting

Make sure you commit AND push your work by 10:00pm on the deadline date.

PS3 takes longer for Didit (https://didit.csail.mit.edu) to process than previous psets, so push early. Remember that feedback is provided on a best-effort basis:

- There is no guarantee that Didit tests will run within any particular timeframe, or at all. If you

push code close to the deadline, the large number of submissions will slow the turnaround time before your code is examined.

- You do not need to have a successful Didit run before the deadline in order to be properly graded.
- Passing the **published test** (https://github.com/mit6005/S14-PS3) on Didit is no guarantee that you will pass the full battery of autograding tests — but failing it is sure to mean many lost points on the problem set.