Assignment One for CS5223

This assignment should be done in teams, where each team can have at most 2 students. Throughout the document, "you" means "your team". You can use Java, C, or variant versions of C such as C++ or C#, for this assignment. If you would like to use a programming language other than those mentioned above, you are required to let me know and obtain permission from me no more than one week after the assignment starts.

1 Background

The objective of this assignment is to give you hands-on experience in designing and implementing distributed systems. Distributed systems tend to easily grow in complexity. Make sure that you finish the basic goals before moving on to more complex functionalities. A major design choice is whether to use RMI or sockets. It is not an obvious choice and each has its pros and cons for this project. You should apply what you learned in class to make your decision. You are allowed (and encouraged) to discuss with other students. For example, you are welcome to do so on the general student discussion forum on IVLE. But you should build your system independently and should never copy any code. Consult me if you are unclear with this policy.

This assignment has two steps, and you should complete both steps, sequentially. Both steps are mandatory. If you do complete both steps, you will be evaluated ONLY for the second step (which is a superset of the first step). If you only complete the first step, you will be evaluated for the first step, and get corresponding partial marks.

2 Step 1: Designing and Implementing A Distributed Maze Game – Client-Server Version

2.1 Overall Functionality

You are to implement a simple maze game in Java. The maze is an N-by-N grid, consisting of N*N "cells". Each cell can be the location of a "treasure". At the start of the game, there are M treasures placed randomly at the grid (multiple treasures at the same cell are allowed).

The number of players is arbitrary. Each player aims to collect as many treasures as possible. A player collects one or more treasures when the player's location is the same as the treasure's location. A player cannot occupy a cell that another player occupies. The game is over when there are no more treasures on the grid.

The game is distributed, i.e. each player connects to a remote game server (or simply, "the server") via either socket or RMI. The players know the server's host name and its port numbers. The server, on the other hand, does not know a priori the number of players or their IP addresses. The parameters N and M are given as command line arguments to the server program, and they are passed from the server to the players. 20 seconds after the first player requests a new game from the server, the game starts. Note that by that time several players may have joined the game. After the start of the game, no additional players are accepted by the server.

The server keeps track of each player's location and number of collected treasures, as well as of the location and number of available treasures on the grid. The players, on the other hand, only control the direction in which they move at the grid. At each step of the player, a player contacts the server giving the direction (S/E/N/W) in which the player wants to move. A player can move one cell at a time, or stay at the same cell. The grid boundaries

are solid, e.g. a player cannot move south if he/she is already at the south-most row. After a player's move, the server replies with the location and number of treasures for each player in the game, together with the number and location of available treasures. Clients should use this information to draw the current status of the maze. For Step 1, you can assume that the server never crashes, and that the network will not introduce any errors or losses.

2.2 Specific Requirements

Your system should achieve the functionality as described in the previous section. You implementation should consist of two separate programs, the player program and the server program, while satisfying the following additional requirements:

- The player program is either single-threaded or multi-threaded, and communicates with the server using sockets or RMI.

- The server should maintain the entire update-to-date game state (e.g., positions of all the players and all the treasures). This state can be sent back to the individual clients when they make moves so that each client can know the current game state.

- When a player moves, it will obviously communicate with the server. When the server processes the request from that client, the server is not allowed to communicate with other clients. In particular, **the server is not allowed to immediately broadcast the updated state to the other players**. The updated game state should be sent to the other players only when they make their moves. In other words, the other players will often see a game state that is slightly stale. This requirement serves to make sure that your design is realistic – in the real world for large-scale games, immediately updating all players incurrs prohibitive overhead.

- The server is a multi-threaded program (remember that RMI is implicitly multi-threaded as well). If you are using sockets, then for each new player, the server should create a corresponding "player thread". A player thread has to keep track of that player's location and number of collected treasures. All playing threads access the same information for the number and location of available treasures, and for the location of other players. Consequently, you need to provide mutual exclusion when accessing these shared variables/structures. If you are using RMI, the threads become implicit, but you still need to ensure properly mutual exclusion when access global data.

- If you are using sockets, then the server's main thread should demultiplex each incoming message to the corresponding playing thread. In other words, **only the main dispatch thread should directly receive messages from clients**. When a message arrives, the dispatch thread has to notify (wake-up) the corresponding player thread to process that message and reply the message to that player thread. I understand that this requirement will not impact the end functionality. However, real servers are usually done in this way because doing so is somewhat more efficient. Hence I need the students to do the assignment in a way that is similar to real-world servers. **Hint: You may want to consider using non-blocking I/O in the main thread.**

- Your program should be able to handle player crashes (you can assume that no crashes will happen during the 20-second waiting periods). "Player crash" here means that the player process is killed, or the power cord of the computer running the player program is unplugged. Normal player exit is not considered as crash. Ask your lecturer if you are still not sure. "Handle player crash" means that the game (including the server and other players that have not crashed) should continue as usual despite that some players have crashed. (Of course, those crashed players will not move any more.)

- 20 seconds after a first player requests a new game from the server, the game should start. Note that by that time several players may have joined the game. After the start of the game, no additional players should be accepted by the server.

- Players should be able to move in an asynchronous fashion. For example, it should be possible for one player to move 5 steps when another player only makes one move. How fast a player moves should only be constrained by how fast the user inputs the directions.

2.3 Some Hints

You do not have to follow the following hints. You can use these hints only if you feel (based on your own judgment) that they are correct and only if you believe that they help you to achieve your goals.

A player can send the following request messages or RMI invocations to the server:

- **joinGame**: A player sends this message to the server (or RMI call) when the player wants to start or join a new game. If you are using sockets, then after the initial 20 seconds wanting period finishes, the server should send back a response message to all clients. The response message should contain a unique ID for each player, a randomly chosen starting position of the player, the location of all the other players, and the location of all the treasures. All this information will enable the client to draw a complete picture of the game state. If you are using RMI, the nature of RMI may not allow you to easily control when the responses are sent. You may consider two solutions. The first is for the server to always respond immediately to the joinGame request, and then at the end of the waiting period, the server can use RMI to callback all the client to indicate the start of the game. The second option is for the server to block for 20 seconds during the first joinGame RMI. "Block" here means that the server will not process other joinGame RMI's during this window. You need to figure out the details yourself.

- **move(id, direction)**: A certain player with a certain id wants to move one cell in the specified direction. The direction can be one of the following values S, E, N, W, NoMove. Note that the move may not be always possible. The server should send back a response containing the new location of the player, whether the player newly collected a treasure (and how many), the total number of treasures collected by each player so far, the location of all the other players, and the location of all the treasures. All this information will enable the client to draw a complete picture of the game state. The "NoMove" direction serves to simplify the design – sometimes the player is not moving, but it still wants to know the updated status of the game.

Finally, as a hint on the timeline, you should plan to finish Step 1 about halfway through the total allowed time for this assignment. Otherwise you will likely have trouble finishing the assignment on time.

3 Step 2: Designing and Implementing A Distributed Maze Game – Peer-to-Peer Version

Step 2 is much harder than Step 1 and builds upon Step 1, so make sure that you complete Step 1 before moving on to Step 2.

3.1 Overall Functionality

We want to improve the previous design in Step 1 to a "peer-to-peer" architecture (while still achieving all the functionalities in Step 1). Namely, you no longer have a server that is guaranteed not to crash. There are compelling

commercial reasons to get rid of the server – if your company can support distributed games without maintaining servers, it means that your company can reak in revenue pretty much without incurring any operational cost.

Our goal here is simple: There will be no dedicated server, and the system only has players where each player is also called a peer. We want to make sure that those players who have not crashed can continue play the game, regardless of who else have crashed. You are allowed to assume:

- No player crashes during the first 20 seconds. After the first 20 seconds, every player may potentially crash.

- To bootstrap, there is one player whose IP/port is known by all other players. Note that this player may also crash after the first 20 seconds.

- Messages never get lost and message propagation delay is at most 1 second.

- A player that has crashed never revives.

- No two players crash at the same time – there is at least 1 minute gap between two successive crashes (so that you can have a large enough failure-free window to do whatever you need to do).

3.2 Specific Requirements

You should design/implement your system conforming to the following overall architecture/requirements, **in addition to those requirements specified in Step 1**:

- Among all the n players, one player should act as the Primary Server (in addition to being a player itself), and another player should act as the Backup Server.

- Both the Primary Server and the Backup Server should maintain up-to-date game state. They should both print out some debug information when other players communication with them (so that we can check during the demo).

- When a player moves, it is allowed to communication with the Primary Server and/or the Backup Server. The Primary Server and the Backup Server may communicate with each other as well if you would like them to. However, the Primary Server and the Backup Server should not communicate with other players (i.e., other than the player making the move request). In particular, **they are not allowed to immediately broadcast the updated state to the other players**. The updated game state should be sent to the other players only when they make their moves. Note that this requirement is similar to the corresponding one in Step 1.

- If the Primary Server crashes, your system should be able to "regenerate" the Primary Server on another (uncrashed) player. You need to properly make sure that the game state on the new Primary Server is brought up to date. (You need to think how to achieve this.)

- The same applies if the Backup Server crashes.

- Your system also needs to be able to deal with player crashes (for those players who are not acting as Primary Server or Backup Server).

3.3 Some Hints

Here are some hints on issues that you need to consider when designing your system:

- You will need to worry a lot about consistency: Since you have two copies of the game state, how do you ensure that they are consistent? In particular, you need to avoiding running into the situation where player A gets the treasure in one copy, while player B gets the same treasure in another copy. Should you have the player directly update the two copies, or should you have the player update the primary and let the primary update the backup copy? In particular, what if some one fails while you are updating the states? These are by far not trivial problems, and even just thinking about them will help you to get more out of this module.

- If you create a new Primary (Backup) Server, how do you bring its state up to date?

- The Primary (Backup) Server can crash at any point of time. In particular, it may crash while it is processing a request. How do you handle that?

- There are a lot of parallelism in the system. For example, while your system is creating a new Primary (Backup) Server, the players may still be issuing requests to the servers. What should you do?

- You can test your system in various interesting ways. For example, you can start with 5 players, and then keep killing the Primary (Backup) Server, until you have only 2 players left. (Since you need a Primary Server and a Backup Server, 2 players is the minimum number possible.)

4 General Advices

The main focus of this assignment is on the threads, synchronization, and communications aspects. Do not spend too much time on the graphics and game algorithm. Even text output (instead of graphics output) is good enough. But you do need to generate sufficient and easy-to-understand output, otherwise there is no way for us to grade your demo.

Proper mutual exclusion access on global shared data on the server is critical for correctness. Make sure that you have a clear thinking of such mutual exclusion. Try-and-debug will not help you solve problems introduced by improper mutual exclusion, and will suck up ALL your time.

For debugging, you should start from simple cases where the server and the clients are different Java processes running on the same computer. This will save you the trouble of using multiple computers. You should NOT use virtual machines. Virtual machines have impact on networking, and may require some extra tuning to make things work properly.

5 Computing Resources

You may use any machine for this assignment. If you would like, you can use the linux cluster (lan-connected) within SoC. For more information, please go to `https://docs.comp.nus.edu.sg/node/1254`. The cluster shares the same account as your SoC unix account. However, you have to enable it (see website for details). For non-SoC students, please apply online for an SoC unix account at `https://docs.comp.nus.edu.sg/node/1258`.

6 Deliverable and Assessment

6.1 Schedule

- Monday 31 August 2015: Assignment starts.

- 11:59pm, Thursday 1 October 2015: Deadline for uploading your source code to IVLE. This will allow us to examine your code, and also to check for plagiarism, **both before and after your demo**. Late source code submissions onto IVLE will result in ZERO mark (out of 30 marks) for this assignment. The reason is that demo questions are secret, so if we allow a student to submit a week later, the student will have enough time to collect complete information for other students. **With such a policy, even if you do not finish on time, you should submit whatever you have by the deadline. Doing so will earn you some partial marks – otherwise you will get ZERO mark.**

- Whole day, Saturday and Sunday 3-4 October 2015: Each team will be assigned a time slot on one of the two days, during which the team will show me/TA a demo of the implemented system, together with the source code. You will be requested to explain your code to the me/TA during the demo. I am not looking for eye candy in the demo, and visual factors will not affect your marks. You may choose to do the demo on your own computer or via remote access to the SoC linux cluster. ALL team members must be present for the demo. **A team member not showing up in the team's slot will result in ZERO mark (out of 30 marks) for that team member, even if the demo could be completed perfectly by the other team member alone.** Every demo/student will be examined/interviewed by both myself and TA. TA does not give out marks, only I give out marks. We want to finish all demos in two days so that we do not leak question sheet (to the extent possible).

6.2   Source Code Submission Instructions

Read through all the requirements. You will be penalized if you don't stick to these instructions.

- You should download "AssignmentOneSubmissionSummary.txt" from the "AssignmentOne" folder on IVLE Workbin, and fill in the information in that text file. The completed softcopy text file MUST be submitted together with your source code. **Without this file submitted, you will not be allowed to do the demo.**

- You should only submit the final version of your code. **In particular, if you did both Step 1 and Step 2, then you should submit ONLY the code for Step2.**

- You should zip all your source files (.java, .c, .h, etc.), together with "AssignmentOneSubmissionSummary.txt", into a single zip file. You are NOT allowed to include any other files. No need to include instructions on how to compile/run your code. We will not compile/run your code – we will only examine your source code.

- All your source files must be in **the same directory with no subdirectories** when you zip them – **NO multiple directories or nested directories allowed**.

- The submission file name must be in the following form: " [Team member 1's matriculation #]_[Team member 2's matriculation #].zip". For example, if the matriculation numbers of team member 1 is HT111111, and the matriculation number of team member 2 is HT222222, then the file name should be HT111111_HT222222.zip. Deviation from such naming scheme may cause mistakes when processing your submission, and may result in the loss of your submission. In addition, explicit penalty will be imposed if you deviation from such naming scheme.

- No multiple submissions allowed. **Each team should make sure that the team only submits exactly once.** If a team submits two versions, we will retain the earliest version and discard all later versions. The team will then be grade on the earliest version. For this reason, if you want to update your submission, you should first delete your old submission and then submit a new one.

- Your zip file should be uploaded to the "Student Submission for Assignment One" folder in IVLE Workbin.

6.3 Grading Guidelines

You will be graded based on i) whether your system achieves the desired functionality while sticking to all the requirements, and ii) whether you can fully explain your code and explain the design choices. The grading is not all-or-nothing. Namely, if your system can achieve a subset of the functionalities, you will still get partial marks for this assignment. **So if you are struggling, do not give up and just finish whatever you are able to achieve!**

Independent of what functionality your code achieves, you will be penalized **severely** if you cannot explain your code. More specifically, your final mark may be reduced by 25%, 50%, 75%, or 100% if you cannot fully explain your code. **Again, I keep my promises and this clause has been enforced before, in previous offerings of CS5223, with a 75% punishment factor.**

A note on GUI. If you are going to do the demo via remote access to the linux cluster, it is you responsibility to make sure that your GUI (i.e., windows) will tunnel back to the machine you are using. In other words, if you are using computer A to access the cluster, all you GUI should show up properly on A's monitor. If you have trouble doing this, you may consider using a pure text user interface, which will always be safe when accessed remotely.

7 THIS IS THE END OF THE DOCUMENT