Assignment One for CS5223

This year is the first time that this assignment is used, which means that the assignment has not yet had the chance to incorporate feedbacks from students. If you have feedbacks on this assignment, please let us know so we can improve it for next year.

This assignment should be done in teams, where each team can have at most 3 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. Regardless of which programming language you use, you will need to use the "StressTest" program that we provide during the demo to stress test your program.

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.

2 Designing and Implementing A Peer-to-Peer Distributed Maze Game

2.1 Overview

You are to implement a distributed maze game. The maze is an N-by-N grid, consisting of N*N "cells". Each cell contains either no "treasure" or exactly one "treasure". At the start of the game, there are K treasures placed randomly at the grid.

The number of players is arbitrary. Each player can be viewed as running on a separate node in the distributed system. 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. Each player has a current score which equals to the number of treasures the player has collected.

The number of remaining treasures in the game will always be K — this means that when a treasure is taken by a player, the game should automatically create a new treasure at some random location.

We want the system to be fault-tolerant. Namely, we want the system to be robust against player crashes.

2.2 The Tracker Program

For implementing this distributed game, you should first design and implement a program called the **Tracker**. The tracker will have a well-known IP address and port number (i.e., known to all people who want to play the game). Furthermore, the tracker knows the values of N and K.

Whenever a new player wants to join the game, the new player should contact the tracker first (at the well-known IP address and port number). This allows the new player to obtain information about the existing players (e.g., their IP addresses and port numbers) in the game, as well as N and K. If you are familiar with BitTorrent, our tracker here provides a similar functionality to a BitTorrent tracker. It is up to you what information a new player gets from the tracker. For example, the tracker can return the entire list of current players in the game. Or the tracker can return a random current player in the game.

After the new player gets such information from the tracker, the new player will contact the existing player(s) (based on such information) to actually join the game. Note that since we allow player crashes, it is possible that the information returned from the tracker becomes stale by the time the new player tries to use it. So you should design your program carefully to deal with such cases.

It is important to note that a player is allowed to contact the tracker **only when** it first joins the game. **A player is not allowed to contact the tracker during its gameplay. The tracker should not maintain any information about the current game state (such as the location of the players in the maze and which player is the primary/backup server), other than the list of current players, N, and K.** The reason is that the tracker is a single node, and hence a good design should always minimize the load on the tracker.

Your tracker program should be invoked in the following way:

- If you are using Java, you should have a "Tracker.class" in the working directory after compilation. Then you should run:
  java Tracker [port-number] [N] [K]
- If you are using C, please name your binary file as "Tracker.exe" after compilation. Then you should run:
  Tracker.exe [port-number] [N] [K]

Here port-number is the port over which the Tracker is listening, while N and K are the parameters for the maze. The (implicit) IP address will be the local machine's IP address on which the tracker runs.

2.3 The Game Program

You should design and implement a program called **Game**, which will be the main program in this assignment. A user should be able to execute Game (from any node/computer) to join the maze game as a new player. As explained earlier, this Game program should contact the tracker as a bootstrapping point. Each player has a name, which has exactly two characters.

The player should join the maze game in the following way:

- If you are using Java, you should have a "Game.class" in the working directory after compilation. Then you should run:
  java Game [IP-address] [port-number] [player-id]
- If you are using C, please name your binary file as "Game.exe" after compilation. Then you should run:
  Game.exe [IP-address] [port-number] [player-id]

Here IP-address is the well-known IP address of the Tracker and port-number is the port over which the Tracker is listening. The player-id is the two-character name of the player.

The Game program should have a GUI, showing:

- The name of the local player (should be shown at the top bar of the window, i.e., the title of the window).
- The current score of all the players (should be shown at the left part of the window).
- The maze as a grid (should be shown at the right part of the window).

- Each treasure in the current maze should be shown as a "*" on the grid.
- For each player, the player's current position should be indicated on the grid, using the player's two-character name.

Additional eye candies in your GUI are allowed but will not earn you extra credits, since the focus of this assignment is on distributed systems.

The Game program should read text lines from standard input:

- If the line equals "0", then the player does not move but refreshes its local state (i.e., so that its local state contains the most up-to-date game state information, including the scores of all the players, the positions of all the treasures, as well as the positions of all the players).
- If the line equals "1", then the player moves West by one step and refreshes its local state.
- If the line equals "2", then the player moves South by one step and refreshes its local state.
- If the line equals "3", then the player moves East by one step and refreshes its local state.
- If the line equals "4", then the player moves North by one step and refreshes its local state.
- If the line equals "9", then the player exits, and all other players should delete this player from the game (so that this player is no longer shown in their GUIs).
- For all other cases, the program does nothing.

The grid boundaries are solid, e.g. a player cannot move south if he/she is already at the south-most row. Players should be able to move in an asynchronous fashion. For example, it should be possible for one player to move five 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.

2.4 The Game State

The *game state* includes the current positions of all the treasures and the current positions/scores of all the players. Since we are designing a peer-to-peer distributed game, there will be no servers storing the game state. Rather, the players themselves have to do the job and maintain the game state. 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 collect revenue while incurring minimum operational cost.

To do a peer-to-peer game, among all the 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* (in addition to being a player itself). The notions of primary server and backup server are logical here, since there are actually no dedicated game servers. **The player currently acting as the primary/backup server should indicate so in its GUI.** When the game initially starts, whoever joins first should be the primary server, and whoever joins second should be the backup server.

Both the primary server and the backup server should maintain up-to-date game state. When a player moves, it is allowed to communication with the primary server and/or the backup server to update the game state as well as retrieve the latest game state. 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. 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 incurs prohibitive overhead.

Your primary server and backup server are also responsible for creating new treasures when existing treasures are collected by the players. Recall that the total number of remaining treasures in the maze game should always be K.

2.5 Server Multi-threading

The server logic should be multi-threaded. If you are using RMI, then RMI implicitly is already multi-threaded and you need to ensure properly mutual exclusion when access shared data among different threads.

If you are using sockets, then for a new player, the server should create a corresponding "player thread". A player thread should keep track of that player's location and score. All player threads access the same data structure 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 data structure.

Furthermore, 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 relay the message to that player thread. We understand that this requirement will not impact the end functionality. However, real multi-threaded servers are usually done in this way because doing so is somewhat more efficient and stable. Hence we 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.** For RMI, it internally already has a dispatch thread, and hence directly satisfies this requirement.

2.6 Fault-tolerance

We want the game to be fault-tolerant. The tracker is assumed to never fail. But a player may crash at any point of time, and we want to be able to tolerate player crashes. "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. "Tolerate player crash" means that the game (including all the other players that have not crashed) should continue as usual despite that some players have crashed. The crashed player should be removed from the maze so that we don't end up having many "dead bodies" in the maze.

Since we are having a peer-to-peer architecture, dealing with player crashes will be tricky since the crashed player may be acting as the primary server or the backup server. We want to make sure that those players who have not crashed can continue playing the game, regardless of who else has crashed. In particular, you will need to do the following:

- If the player acting as 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 player acting as the backup server crashes.
- As long as there are at least two players, your primary server and backup server should not be on the same player.

To make your job easier (or perhaps to make your job *possible*), for dealing with player crashes, you are allowed to assume:

- A player that has crashed never revives.

- Messages never get lost (under TCP and RMI) and message propagation delay is at most 0.2 second.
- No two players crash at the same time – there is at least 2-second gap between two successive crashes (so that you can have enough failure-free time to do what you need to do).

3 Some Hints

- **Think thoroughly before you implement.** This is the single most importance hint I can give you. Think through your entire design, think through all possible cases and in particular corner cases, think through all the possible alternative designs, before start programming. Resist the temptation to start programming until you have thought thoroughly. Doing so will save you a lot of trouble later. I have seen numerous cases where the students do not think through, and end up with an unnecessarily complicated design that takes a lot of time to implement, or end up having a buggy design that is very hard to debug.
- 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 avoid 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 someone 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 possible number.)
- 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 players and the tracker 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.

4 Deliverable and Assessment

4.1 Schedule

- Monday 22 August 2016: Assignment starts.

- 11:59pm, Friday 30 September 2016: 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 for this assignment. The reason is that demos will be done on the next two days, and without the source code we cannot let you do the demo. **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 1-2 October 2016: Each team will be assigned a time slot on one of the two days, during which the team will show TA a demo of the implemented system by **doing the StressTest (described below)**, together with the source code. In rare occasions, you may be also requested to explain your code to me later. 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 for that team member, even if the demo could be completed by the other team members.**

- If you prefer, you are allowed to submit your code and to do the demo with the TA before the deadlines. Doing so does not give you extra credits, though it does allow you to move on to Assignment Two. If you would like to do so, please contact the TA directly to arrange a time to do the demo. You are required to submit the code to IVLE before doing the demo with the TA.

4.2 Source Code Submission Instructions

- You should download "AssignmentOneSubmissionSummary.docx" from the "AssignmentOne" folder on IVLE Workbin, and fill in the information in that file. The completed softcopy 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.

- You should zip all your source files (.java, .c, .h, etc.), together with "AssignmentOneSubmissionSummary.docx", into a single zip file. You should NOT 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**.

- The submission file name must be in the following form: "[Team member 1's matriculation #]_[Team member 2's matriculation #]_[Team member 3's matriculation #].zip". For example, if the matriculation numbers are HT11, HT22 and HT33, then the file name should be HT11_HT22_HT33.zip. Deviation from such naming scheme may cause mistakes when processing your submission. For teams with less than 3 students, the submission file name should just include all your team members. For example, the file name can be HT44_HT55.zip for two-member team with matriculation numbers HT44 and HT55.

- 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 graded 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 Files.

4.3 Stress Test for the Demo

The demo will be done in an automatically way using the "StressTest" program. The source file, named "StressTest.java", is given to you at the beginning of the assignment. **You should read through "StressTest.java" very carefully, at the very beginning of the assignment, to understand what it does. The comments in "StressTest.java" tells you how to do the demo properly.** Before the demo, you should set up RMI registry (if needed) and the Tracker program properly. During the demo, you will be asked to compile and then execute the "StressTest" program. (Other than running the "StressTest" program, there is nothing else you need to do during the demo.) The StressTest will create many players (by invoking your Game program), inject move requests to the players, kill players, and so on. We will not change the "StressTest" program in any way during the demo, and hence you can view this as an "open-book" test. The maze size for the StressTest must be 15*15, and the number of treasures must be 10.

You should do the demo on your own computer. If your team does not have a computer, please inform the TA at least two weeks before the demo so that the TA can arrange a computer for you.

4.4 Completing Stress Test before the Demo

You should execute the "StressTest" program well before you come to the demo and make sure everything works since you will not be given opportunities to debug your program during the demo. Remember that you are required to write done how many checkpoints your program can correctly pass under this StressTest in your "AssignmentOneSubmissionSummary.docx" before you can submit your code to IVLE.

While "StressTest.java" is given to you at the beginning of the assignment, note that the steps done in this program will be "stressful" to your program. Namely, the execution path it results will be extremely complicated. (This is precisely the reason why we do not need to hide this test program from you. :)) Hence if your program breaks under StressTest, we suggest that you do not try to immediately debug your program under StressTest – doing so can be frustrating. Rather, you should try to debug your program separately using simpler test cases. (In fact, this is another beauty of our StressTest – in real life, if your distributed system breaks under a complex workload injected by real users, it may not be possible to debug by replaying that complex workload – hence we want our StressTest to mimic such property.)

This semester is the first time we use this "StressTest" program. While we have tried our best to do complete testing of the program, there is still chance that the "StressTest" program itself does not work well on certain platforms (especially since we do not know what hardware/software/OS platform you will be using). We will have to rely on your help: For any team that finds a bug in the "StressTest" program and fixes the bug, for this assignment, all team members in that team will be given 2 extra points (subject to the condition that the total marks for Assignment One CANNOT exceed 40 – this means that if you get 40 marks already, then these extra points will be wasted and will not further increase your mark). Such "bug discovery" is based on first-come-first-serve principle: For each bug, the 2 extra points goes to the team that *first* notifies us about it. Later teams will not get any extra points.

4.5 Grading Guidelines

You will be graded based on how many checkpoints your program can correctly pass (the TA will verify at each checkpoint that your game still runs correctly) under the "StressTest" program. **It is important to remember that the grading is done based on the StressTest, and hence just showing us how your game works will not earn you any marks.**

The grading is not all-or-nothing, since it is determined by the number of checkpoints you can correctly pass. If your system can correctly complete only the first checkpoint, you will still get partial marks for this assignment. **Namely, if you are struggling, do not give up and just finish whatever you are able to achieve!**

While you will be graded on how many checkpoints your program can correctly pass, your system should still work correctly in the normal case (i.e., while not being invoked from StressTest) and under a distributed setting. In particular, you should NOT tailor your program just for StressTest, to a point where it works under StressTest but not under other settings. For example, you should NOT reverse-engineer the sequence of player moves, player joins, and player kills in StressTest, and then feed the sequence into your Game program to make it easier for your Game program to deal with these moves, joins, and kills (since then your Game program can "predict" what will happen). Deviating from this requirement will be viewed as cheating and will result in substantial reduction of your marks for this Assignment. More specifically, your final mark may be reduced by 50%, 75%, or 100% if you don't follow this requirement. We may examine your source code and/or run additional tests to confirm your compliance to such requirement.

5 THIS IS THE END OF THE DOCUMENT