Spring 2014 CSCI 182E—Distributed Systems
Homework 6
Tournament Yahtzee[1]

Value: 30% of Homework Assignments Grade

Daniel M. Zimmerman
Harvey Mudd College

Assigned 9 April 2014
Design Due 21 April 2014, 13:15:00
Implementation Due 2 May 2014, 23:59:59

This assignment is to be completed in teams of 2 or 3

The main purpose of this assignment is to exercise various distributed algorithm techniques we have discussed in class by designing a protocol for, and building agents that implement, a distributed Yahtzee tournament.

This assignment, like the previous assignment, requires you to consider issues of fault tolerance. It also requires you to consider issues of trust and process knowledge (i.e., how can a "player" tell that another "player" in the game is not cheating).

You are strongly recommended to use Erlang and its distributed messaging for this assignment. You may choose to implement your programs in Java using Erlang messaging instead, but it's not a good idea.

Yahtzee Overview

General Rules *Yahtzee* is a game that can be played by one or more players. A game of Yahtzee requires five six-sided *dice* and a *scorecard* for each player. The scorecard contains a list of thirteen *dice patterns*. Next to each pattern on the scorecard is a *box* in which to enter a *score*; all boxes are empty at the start of a game. A game of Yahtzee consists of thirteen turns, one for each pattern on the scorecard.

Each turn in the game consists of an *assembly phase* and a *scoring phase*. The assembly phase is started by rolling all dice. After the first roll, there are two *modification attempts*. In each modification attempt, the player chooses a subset of the dice (the *keepers*) and rerolls the other dice. In the scoring phase, the player chooses a pattern on the score card with an empty box and enters a score in the box. The score is determined by *pattern scoring rules* (shown below) based on the final roll of the assembly phase and the chosen pattern.

[1] Yahtzee is a registered trademark of Milton Bradley, which is owned by Hasbro, Inc.

When all boxes have been filled with scores, the game ends. The final score is determined by the final scoring rules, based on the final state of the score card. As with most non-golf games, the aim of Yahtzee is to finish each game with the highest score.

Notes

- In a modification attempt, the player may choose *any set* of dice as keepers, including all dice or no dice. Keeping all dice is essentially skipping the modification attempt.
- The two modification attempts are *independent*. There is no restriction on the second set of keepers; in particular, the second set of keepers does not have to be a superset of the first set of keepers.
- The player may choose any dice pattern for scoring as long as its box is empty. The resulting score can be 0 if the conditions for the box are not met or if the score calculation for the box yields 0.

Pattern Scoring Rules The patterns on the scorecard, and their scoring rules, are shown in the following table. a and b denote *distinct* values shown on dice (e.g., the sequence a, a, a, b, b means three dice with value a and two dice with value $b \neq a$). x, y and z denote values shown on dice that may not be distinct (e.g., the sequence x, x, x, y, z means at least three, but possibly four or five, dice with value x).

Upper Section		
Pattern	**Condition**	**Score Calculation**
Aces	none	total value of all aces (ones)
Twos	none	total value of all twos
Threes	none	total value of all threes
Fours	none	total value of all fours
Fives	none	total value of all fives
Sixes	none	total value of all sixes
Lower Section		
Pattern	**Condition**	**Score Calculation**
Three of a Kind	x, x, x, y, z	total value of all dice
Four of a Kind	x, x, x, x, y	total value of all dice
Full House	a, a, a, b, b	25
Small Straight	$x, x+1, x+2, x+3, y$	30
Large Straight	$x, x+1, x+2, x+3, x+4$	40
Yahtzee	x, x, x, x, x	50
Chance	none	total value of all dice

The indicated pattern score is obtained only if the condition is met; otherwise, the score is 0 unless the extra Yahtzee rules (below) apply.

Extra Yahtzee Rules Extra Yahtzee rolls (of 5 identical values on the dice) are treated specially in the pattern scoring rules, in two ways:

1. **Extra Yahtzee Bonus:** Each Yahtzee roll earns a bonus of 100 points, regardless of the pattern for which it is scored, as long as 50 points have already been scored for the Yahtzee pattern. If 0 points have been scored for the Yahtzee pattern, Yahtzee rolls earn no bonus. The Extra Yahtzee Bonus does *not* count toward reaching the threshold for the Upper Section Bonus (below).
2. **Extra Yahtzee as Joker in Lower Section:** A Yahtzee roll with value v on the dice can be scored for the patterns Full House, Small Straight, or Large Straight at those patterns' full point values, as long as 50 points have already been scored for the Yahtzee pattern *and* the v pattern in the Upper Section has already been scored (even if it was scored with a 0). In this case, the Yahtzee roll acts as a *Joker*.

Final Scoring Rules The final score is the total of all pattern scores plus 35 points (the *Upper Section Bonus*) if the total of the Upper Section scores, excluding any Extra Yahtzee Bonus scores in the Upper Section, is 63 (the *threshold*) or higher. Note that the threshold can be reached by scoring Three of a Kind for each value 1–6 in the Upper Section, though this is not the only way to reach it.

Tournament Yahtzee Differences In *Tournament Yahtzee*,[2] each game is played by two players. The sequence of 15 possible die values in each round (the number of dice rolled if no dice are ever kept between rolls) is *identical* for the two players, each of whom can observe the other's rolls and choices for that round *only after the round is complete*. For example, consider the following game round (dice kept from the previous roll are denoted in ***boldface italics***), which has the 15-die sequence 2, 3, 6, 4, 6, 2, 6, 5, 1, 4, 3, 2, 6, 1, 4:

	Player One					**Player Two**				
initial roll	2	3	6	4	6	2	3	6	4	6
modification attempt 1	***2***	***3***	2	***4***	6	2	6	***6***	5	***6***
modification attempt 2	***2***	***3***	5	***4***	1	1	***6***	***6***	4	***6***

Player One finishes the round with a Large Straight (1, 2, 3, 4, 5) for 40 points (among other possibilities). Player Two finishes the round with Three of a Kind of sixes (1, 6, 6, 4, 6) for 22 points (among other possibilities). Both players had the same sequence available to them; however, Player One only saw the first 9 dice in the sequence and Player Two only saw the first 10 dice.

[2] There is no official "tournament Yahtzee" to the best of the instructor's knowledge; it is being invented for this assignment.

Neither player has access to the entire sequence during the round; it is assumed that a player learns roll values during the round only by actually rolling dice. Thus, player 1 can be confident that player 2 is not cheating (i.e., lying about dice values or "rewinding" dice values to retroactively make better choices), and vice-versa, by comparing the choices player 2 claims to have made with the actual 15-die sequence revealed at the end of each round. In a physical game this could be done in several ways; for example, each player could write the choices for each roll on a piece of paper and put it in a sealed box in sight of the other player before making the next roll. Alternatively, a referee could monitor the game to ensure player honesty.

A tournament in Tournament Yahtzee is run with a standard single-elimination tournament bracket (see Wikipedia's entry on single-elimination tournaments), where each match consists of an odd number $k > 0$ of Tournament Yahtzee games; the first player to win $\lceil k/2 \rceil$ games wins the match. Tie games are discarded and played again (with new dice rolls). An undecidable match, where two players have identical strategies and tie $\lceil k/2 \rceil$ consecutive games, is replayed under *standard* Yahtzee rules (where the rolls for the two players are *not* identical) to determine a winner (the replayed match still consists of k games).

Assignment Overview

For this assignment, you will design and implement a distributed system consisting of both processes that play Tournament Yahtzee and processes that organize and run Tournament Yahtzee tournaments. Because this involves a significant amount of protocol design, the assignment is split into two phases: a *design phase* and an *implementation phase*. Each phase has its own submission requirements, described later.

The design that you submit for the design phase will be discussed in class on Monday, 21 April, and all groups should be prepared to describe their designs to the rest of the class at that time. The goal of the discussion will be to settle on a protocol that *everybody* can implement, so that all the player processes will be interoperable and the graders can run tournaments as part of the grading of the assignment.

Note that this assignment is primarily about the interesting distributed systems aspects of the problem (managing all the games in the system, keeping track of tournament status, dealing with crashing players, ensuring honest play, etc.) and not about Yahtzee strategy! There are multiple resources related to optimal Yahtzee strategies available on the Internet; you are *explicitly forbidden* from using these resources when developing your player implementations. Bonus points are available on the assignment for performance in tournaments (assuming that tournament play is successfully implemented!); however, other than behavioral correctness considerations and these bonus points, Yahtzee play quality will not be considered in the grading.

Note also that this assignment is *significantly* more open-ended than previous assignments. You are encouraged to ask questions early and often!

System Requirements

You must design a system consisting of *at least* two different kinds of process (at least one *tournament manager* process and an arbitrary number of *player* processes).

A player process must do the following:

- Register with and communicate with at least one tournament manager.
- Play Tournament Yahtzee matches against other players, as directed by the tournament manager(s), and report the results to the tournament manager(s). A player should be able to play multiple simultaneous matches.

A tournament manager must do the following:

- Handle authenticated registration and login/logout requests from players. The first time a player registers with the manager, it gives a name and a password or key;[3] if the player crashes or logs out, it can subsequently use the same name and password/key to log back in while maintaining its win/loss record and tournament status.
- Run multiple tournaments simultaneously. This involves constructing tournament brackets, communicating appropriately with players to start matches and receive the results of matches, detecting when the players in a match have crashed, etc..
- Record the results of tournaments and maintain player win-loss records (for both matches and tournaments) and rankings.

Note that "process", as used here, is *not* necessarily the same as an Erlang process; in particular, a player or tournament manager might consist of several communicating Erlang processes within the same Erlang VM. One reasonable strategy would be to spawn one Erlang process per game/tournament being played/managed, with the results being reported back to a "main" process for the player or tournament manager.

Similar to the logging requirements on previous assignments, all the processes in your system must log sufficient information about messaging and execution to understand their behaviors and reveal their states.

[3] You might, if sufficiently ambitious, want to use actual cryptography to handle these requirements; however, it is far more likely that you will simply do this with plain text under the assumption that you *could* do it right in a real-world implementation.

Behavioral and Fault Tolerance Constraints

The following constraints on the behavior expected from your system should make the design and implementation tasks more manageable.

- You may assume that tournament managers, and any auxiliary process types you introduce, never crash. Hopefully your implementations will live up to this assumption; it is stipulated here mainly so that you don't have to worry about fault tolerance with respect to tournament manager data.
- You may assume that players can exhibit only two types of failure: crash failure and "cheating". Cheating, in this context, means manipulation of the reported scores and dice rolls to gain an advantage. You may further assume that if a process cheats, it gives the same information to all relevant parties; for example, if a cheating player communicates both directly with another player and with a referee process, it tells both of them consistent lies. The upshot: you do not need to solve the Byzantine Generals problem on this assignment.
- You may assume that crashed players can be detected by Erlang monitoring or by timeout (with some reasonable timeout value). Crashed and logged out players are dealt with in the following way:
  - A crashed player forfeits any game it is currently playing, but *not* the entire match (if matches consist of more than 1 game); the non-crashed player and any auxiliary processes should wait a reasonable amount of time (at least 1 minute) for the crashed player to restart and log back in to the tournament manager, and there must be a mechanism by which the crashed player can be brought up to speed with the previous events of the match (including the fact that it forfeited a game). If the crashed player does not restart and log back in within a reasonable amount of time, it forfeits the match.
  - If a crashed or logged out player p is *not* actively playing a match and the tournament manager wants to start a new match involving p, the tournament manager must allow a reasonable timeout period (at least 1 minute) for the player to restart if necessary and log back in. If p does not log back in, p's opponent wins the match and it counts toward both players' statistics.
  - If both players in a match have crashed and the next game in the match cannot be started, the match is voided and does not count toward either player's statistics. It is restarted by the tournament manager if the players restart and log back in.
  - If a match cannot be finished (because both players crashed and did not log back in within a reasonable timeout period), there is no winner and "bye" advances to the next round (the "bye" mechanism is also a convenient way to deal with setting up brackets for tournaments with numbers of players that are not powers of 2). An actual player automatically wins any match

scheduled against “bye”, but it does not count toward that player’s statistics. If a match is scheduled between “bye” and “bye”, “bye” advances to the next round. It is possible for a tournament to be won by “bye”, but the tournament manager does not keep statistics for “bye”.

- You should treat a player caught cheating identically to a player that has crashed with respect to determining outcomes of games.
- You may assume, if you find it useful for the purposes of designing your protocol, that all communications are authenticated (in the sense that we used the term in the Byzantine Generals solution with encryption and signatures), even though you will likely not implement them that way.

Implementation Requirements

The requirements for a tournament manager implementation are as follows:

- The module containing the main function must be called `yahtzee_manager` and must export at least the function `main/1`; it may also export other functions as you find necessary.
- It must take one command line parameter: a name to register itself globally with (using Erlang’s global naming service). This name should be all lowercase with no `@` signs, as on previous assignments, and should also be used as the node short name to start the network kernel.
- It must be able to start a tournament of a given size, with a given (maximum) number of games per match, among a random selection of the registered players, upon request from the environment (the precise mechanism for this will be part of your protocol design), and to run multiple simultaneous tournaments.
- It must generate sufficient console output to show exactly what is happening with respect to message sends/receives and state changes. These should be described in human-readable fashion on standard output (including information about the particular process that is sending/receiving the message or changing its state, since you will likely have more than one process per Erlang VM). In particular, it must be absolutely clear when tournaments are being started; who wins matches and tournaments; when players are registered, log in, and log out; and when/how players’ win-loss records are changed.

  This will likely generate an extremely large amount of output.
- It *must not* use the global locking and transaction facilities provided by the Erlang `global` module.

The requirements for a player implementation are as follows:

- The module containing the main function must be called `yahtzee_playerX` where *X* is a number from 1 through 9 inclusive (you are *allowed* to submit up to nine player modules to implement different strategies; you are only *required* to submit one), and must export at least the function `main/1`; it may also export other functions as you find necessary.
- It must take the following command line parameters:
  - A name to start the network kernel with; this name should be all lowercase with no **@** signs, as on previous assignments. You are not required to globally register players, though you may choose to do so if you find it useful for your protocol.
  - A username to use when registering itself with tournament managers.
  - A password/key to use when registering itself with tournament managers.
  - One or more tournament manager names (the ones globally registered by the tournament managers). If more than one is specified, the player registers itself with all of them.
- It must be able to play a game of Tournament Yahtzee upon request from a tournament manager with which it is registered (the precise mechanism for this will be part of your protocol design), and must be able to play multiple simultaneous games.
- It must generate sufficient console output to show exactly what is happening with respect to message sends/receives and state changes. These should be described in human-readable fashion on standard output (including information about the particular process that is sending/receiving the message or changing its state, since you will likely have more than one process per Erlang VM). In particular, it must be absolutely clear when games are being started; who they are being played against (if your protocol provides that information); what decisions are made during the game with respect to rolls and scoring; when games are won or lost; and when tournaments are won or lost.
- It *must not* use the global locking and transaction facilities provided by the Erlang `global` module.

Grading and Submission

40% of the grade for this assignment is based on your protocol and process design: the state and behaviors of the players and tournament managers and any auxiliary processes your system requires, as well as the message types that are exchanged and how they are used. Note that you do *not* need to include any information about your Yahtzee strategy, or about how exactly you are keeping track of Yahtzee game state in the players, determining tournament brackets, etc.; those are implementation details for which the requirements are already well defined, and this part of the assignment is primarily about the distributed systems aspects of the problem that are not already well defined. You should also provide (brief) arguments for the correctness of your design; in this context "correctness" means "conformance to the general assignment requirements", because there are many choices you can make that would result in wildly different actual runtime behavior while still fulfilling these requirements.

Your design must be submitted to the Assignment 5 page on Moodle **before the beginning of class on Monday, 21 April**, by uploading one of `design.pdf` or `design.txt`, a PDF or text file (respectively) containing a writeup of your design and a list of your team members (this is *extremely important*). You can use any document preparation software you choose to generate a PDF or you can submit a plain text document; no other document formats (LaTeX source, Word, Pages, RTF, etc.) will be accepted. In addition to this submission, you must come to class on 21 April prepared to discuss your design.

The remaining 60% of the grade for this assignment is based on your implementation and testing. Your implementation must be based on the protocol agreed upon during the 21 April class (most likely to be distributed online shortly thereafter), so that both your tournament manager and your player can interoperate with those of all the other groups of students in the class. You must also perform (and describe) sufficient testing to convince us that your implementation works. You may include up to 9 different player implementations in your submission (to improve your chances of gaining bonus points in tournaments, or simply for fun).

To submit your implementation, upload 2 files to the Assignment 6 page on Moodle by **23:59:59 Thursday, 2 May**:

- `yahtzee.zip`, a Zip file containing your Erlang source files (you may only have one, but you may also choose to put support routines in another module or to include Erlang-based testing code), details about your testing strategy (in a text file), any test programs/shell scripts/etc. you have written, and any other files (such as logging output) that you wish to provide as evidence that your program works.

- `readme.txt`, a text file containing a list of your team members (this is *extremely important*), any information you think the graders need to know about your code, and a brief description of your experiences implementing the homework assignment.

Only one team member needs to upload the files (for both parts of the assignment) to Moodle; be *absolutely sure* that `design.pdf`/`design.txt` and `readme.txt` include the names of all team members!

Evaluation

Your design will be evaluated on its completeness and correctness, determined by your provided descriptions and your correctness arguments. The similarity of your design to the one agreed upon for everybody's implementation will not be taken into account during grading.

Your implementation will be evaluated on its functional correctness (90%), determined from your testing strategy as well as from our ability to use your implementations in tournaments, and on its code style/readability (10%).

Assuming we are able to successfully run tournaments, we will run 5 tournaments with all the submitted player implementations. Each team submitting a player that wins at least one tournament match will earn 5% extra credit on the assignment. In addition, the team submitting the winning player for each tournament will earn 3% extra credit on the assignment. Thus, if your player implementation is truly excellent and wins every tournament, it is possible to earn 20% extra credit on the assignment.

If you put no effort into your `readme.txt` file, you may be penalized up to 10% of the value of the assignment. As long as you write some genuine reflections about your experiences working on the assignment, you will not incur a penalty.

If the names of all team members (including the one who submitted the files) are not listed in `design.pdf`/`design.txt` or `readme.txt`, all team members will receive a 0 on the assignment.