An Algorithm For Distributed Tournament *Yahtzee*®

Alex Melville David Scott Jesse Watts-Russell

April 21, 2014

Protocol/Algorithm Description

In the below description, it is intended that `player` processes are referred to with the pronoun "it" as processes are non-sentient, even though a `player` typically represents a user, which could potentially be a human. Anywhere a gendered pronoun is used should be assumed to be an error. The distinction between a `player` [process] and a user is material, if not overly crucial, throughout this writeup.

In our setup, there are two types of proceses: `tournament manager`s and `player`s. We assume that there only exists a single `tournament manager` whose identity is "magically" known to all `player`s (so we are not concerned with uniquely identifying a `tournament manager` in our protocol); there can be an arbitrary number of `player`s. One underlying assumption we make is that `tournament manager`s are infallible; i.e. a `tournament manager` will never conspire with players to allow a certain player to cheat; anytime a choice needs to be made, that choice will be made as fairly possible; and the `tournament manager` will never fail (network-wise) unexpectedly.

The `tournament manager` handles all logic related to:

- User information, including authentication and play statistics. This requires little more than intelligent use of some database-like structures.
- Starting and running tournaments, including storing all necessary information about in-progress tournaments such as the playoff bracket.
- Tracking each `player`'s progress in its current match, including ending the match as soon as one player has won, or "restarting" the match with giving the two players independent die rolls if the match reaches the "draw" condition.
- Playing games of *Yahtzee*®, including rolling dice and keeping track of the most recent turn's die rolls in order to check for cheating. May require some clever use of data structures in order to properly handle games where the `player`s are allowed distinct die rolls (i.e. games where the two players have "equivalent" strategies, as specified in the assignment).

- Handling cheating `player`s (where cheating can be impossibly-scored turns, illegally-scored turns, or retroactively re-scored turns). When a `player` is caught cheating, that `player` immediately forfeits all its current matches, is entirely removed from the collection of players waiting for a tournament, and the user is added to a blacklist. Users on the blacklist are not allowed to enqueue for any future tournaments.
- Handling players who become nonresponsive (for whatever reason). This is handled as specified in the assignment. Timeouts are implemented in a non-blocking fashion. If a user who is actively playing in a tournament logs [back] in within the timeout period, the `tournament manager` ensures that the `player` is prepared to continue playing, and then does so.

This leaves the `player` to handle only:

- Deciding to log in, and creating a password if necessary. The `player` then keeps track of the password in some "secure" fashion for the duration of a "session" (the time from when a user logs in successfully to a `tournament manager` until the `player` disconnects from the `tournament manager` (by experiencing a fault or by the user logging out)).
- Deciding to log out
- Deciding to enter into a *Yahtzee*® tournament
- Deciding to leave a *Yahtzee*® tournament (whether or not that tournament has begun)
- Deciding what actions to take within a game (which dice to re-roll, and how to score the turn). This requires keeping and updating a running score card for each game the `player` is actively playing, which should be done with an appropriate data structure.

Messages

Below we have listed the messages that will be passed around between `tournament manager`s and `player`s.

For messages where a `player` sends a password to the `tournament manager`, unless otherwise specified, the following protocol is observed.

If the password is correct, no response is sent; if it is not, a `not_authenticated` message is sent. This is because a non-malicious `player` who logged in with a correct password must still know the password, so can simply assume that the request succeeded; we don't need to be overly concerned with malicious users, but can be nice and send them a failure message anyway.

{Username, Password, logout}: A `player` sends this message to a `tournament manager` when it wants to log out. The password is required as a security measure, to prevent a malicious player from simply logging out its opponent and thereby winning the game by default.

{Pid, login, Username, Password}: A `player` sends this message to a `tournament manager` when it wants to log in. Password is either a password or the atom `false` if the user does not have a password yet (i.e. is not registered). It is assumed that a non-malicious `player` keeps track of its password indefinitely after a successful login attempt.

{authenticated, ActiveTournaments, PendingTournaments}: A `tournament manager` sends this message to a `player` who has successfully logged in. `ActiveTournaments` is an integer of the number of tournaments the `player` is currently actively playing in; `PendingTournaments` is the number of tournaments the `player` is enqueued for that have not yet begun.

If `ActiveTournaments` is positive, then the player should expect to resume the match they were a part of by soon receiving a `dice` message from the tournament manager.

{not_authenticated}: A `tournament manager` sends this message to a `player` who attemped to log in with a username that is already logged in, or if the `player` tried to perform some action that requires authentication but gave an incorrect password.

{new_user}: A `tournament manager` sends this message to a `player` who has signed in using an unregistered username.

{new_password, Username, NewPassword, OldPassword}: A `player` sends this message to a `tournament manager` when the `player` wants to change their password. For a new player, `OldPassword` should be simply the atom `false`.

{Username, Password, enqueueMe}: A `player` sends this message to `tournament manager` when it wants to enter another tournament. The `tournament manager` simply adds the given user to a collection of waiting players from which players will be pulled when the next tournament is set to begin. This collection must allow duplicate elements (in some fashion), and is likely to be implemented as a queue, though this is not necessarily the case; there is no guarantee that players will begin a tournament in the same order they signed up. (In fact, this cannot be the case; if it were, it would be possible for a player to enter the same tournament multiple times).

{Username, Password, dequeueMe}: A `player` sends this message to a `tournament manager` when it wants to be enqueued one less time for a new tournament.

{dice, TournamentRef, GameRef, DiceSequence}: A `tournament manager` sends this message to `player` whose game is in progress. `DiceSequence` is a collection of up to 5 integers between 1 and 6 representing die rolls. `TournamentRef` and `GameRef` are `ref`s uniquely identifying the particular tournament or game, respectively. Neither is generated until the particular tournament or game is actually started.

{Username, Password, modification, TournamentRef, GameRef, DiceSequence}: After a `player` receives a collection of dice, if it is allowed more modification attempts in the given turn, it may send some number of dice back to the `tournament manager`. When a `player` does this, it expects to receive in response another `dice` message.

{Username, Password, end_turn, DiceSequence, ScoreUpdate}: When a `player` has used up its two allowed modification attempts or otherwise decided to end its turn, it tells the `tournament manager` what dice it ended up with and how it decided to score that turn. This is entirely to make it easy for the `tournament manager` to check for cheating: both the `player` and `tournament manager` keep track of an entire scorecard for every turn, and at each update the `tournament manager` ensures that the update was legal. This also makes it easy for the `tournament manager` to determine the game's winner at the end.

{Username, Password, drop_tournament, TournamentRef}: When a `player` wants to drop out of a particular tournament that it is actively playing in, it sends this message to the `tournament manager`. The `player` can simply ignore any further messages concerning the given tournament since even if the `drop` request fails, the error-handling protocol implemented by the `tournament manager` will deal with the player appropriately after it fails to respond within some timeout.

{tournament_over, TournamentRef}: Sent by a `tournament manager` to a `player` who has finished participating in a tournament, by either being knocked out or by winning the whole thing.

{Username, stats_request}: Sent by a `player` to the `tournament manager` when it wants to know the game-win statistics of the given user. The given user can be anyone since all player statistics are public information.

{stats, Username, UserStats}: The response to a `stats_request` nessage sent by a `tournament manager` to a `player`. `UserStats` in some way encapsulates relevant statistics about the given user.

{start_tournament, MaxPlayers}: A message sent from a mysterious source in the æther to a `tournament manager` telling the `tournament manager` to begin a tournament using as many of the "next" (for whatever definition of "next" the `tournament manager` finds appropriate) distinct `MaxPlayers` enqueued players as possible. Since it is impossible for a `player` to gain an advantage from illegitimately initiating a tournament, there is no need for authentication here.

Correctness

Most of the correctness of this protocol is explained throughout the description of the protocol. The most important claim, though, is that this protocol enables the construction of a fully-functional application that conforms to the design specifications and allows for the implementation of every required feature.

Since the above description primarily concerns itself with the communication protocol and not the implementation details of any algorithms, all major potential sources of failure (user-password authentication, game-playing strategy, cheating detection) are abstracted away in this description, where any such algorithms are simply assumed to be correct. Ensuring that a particular implementation of this protocol is correct and satisfies all the requirements will require additional work to ensure the correctness of implementation-specific choices in algorithms.