1 Introduction

Communication is done entirely through stdin/stdout. Stderr is reserved for logging debug info. This makes clients language-independent and communication as simple as possible (hopefully). I'm imagining "bots" are binaries that satisfy the input/output protocol as defined below, and would be submitted in binary form to the server which then executes a copy per instance at a table.

For now, the game is limited to 3-player Kuhn Poker (4-cards, one card dealt to each person, ante of 1, and a maximum bet of two). Hopefully we can extend to proper Hold-Em at some point.

A **round** looks like a cash game with an end condition: a probability per hand to be the last hand. Bots have infinite bankroll, and the absolute wins/debt against the bankroll determines score in a round. Probability of ending on a given round TBD (should be long enough that the bots have a chance to learn and adapt to other play styles).

At the start of a round, all clients are fired up and given the `init_round` message with the blind info (for Kuhn Poker, blind of 1 for everyone) and starting totals (for infinite bankroll we set these to zero and just allow negative). Inital button is chosen uniformly at random and hands are played out (`init_hand`, `play`, and `end_hand` messages). The random end condition is tested after each round, and once it is hit the `end_round` message is sent out so the bots can write out any debug info if desired.

2 Protocol

Typewriter font stuff below is verbatim format for sent/received messages, with angled bracket labels indicating things to fill in. Players are labeled 0, 1, and 2, with 0 the bot being interacted with and going clockwise about the table from there. In other words, the server will be sending messages to each bot differently so that everything lines up with their view of the table. All numbers are integers. Generally the upcoming message protocol is sent in `lower_underscore`, information is sent in `CamelCase` fields and actions are indicated by `ALL CAPS`.

2.1 `init_round`

Server sends starting money, blinds, button location, and probability a/b of ending the round after each hand:

```
init_round
Money: <m0>,<m1>,<m2>
Blinds: <b0>,<b1>,<b2>
Button: <player id>
EndProb: <a>,<b>
```

Client sends back:

```
READY
```

2.2 end_round

Server sends final change in bankroll (i.e. score) for each player and total number of hands played:

```
end_round
Bankrolls: <b0>,<b1>,<b2>
NumHands: <num hands>
```

The client, being the polite AI that it is, sends back

```
Thank you dealer, have a nice day!
```

The client program should then exit after possibly writing out some logging info to stderr as desired.

2.3 init_hand

Server sends current hand index (zero-indexed) and dealt card:

```
init_hand
Hand: <hand number>
Cards: <A or K or Q or J>
```

Client confirms:

```
READY
```

2.4 play

Server sends list of actions for player 0, 1, and 2 in that order:

```
play
Action: <BLIND or PASS or BET or FOLD> <value>
Action: <BLIND or PASS or BET or FOLD> <value>
Action: <BLIND or PASS or BET or FOLD> <value>
```

This includes, as the start of the list, the action the client last played (because player 0 is always the client itself). If this is the start of the hand, most actions will be listed as `BLIND` and the value paid, even though the bot never explicitly sent that command.

The client then responds with their action:

```
<PASS or BET or FOLD> <value>
```

The `<value>` field is always an integer and should be exactly the amount of money in front of you regardless of your action. This gives a check that the bot understands the state of the world, and makes it easier for other bots to play "in the moment" without lots of tracking state (the server checks this for us). For example, if the action is a raise of 25 over 15 previously bet, then the bot

would output `BET 40`. As another example, if the bot has ante'd 1 and wants to fold, the output should be `FOLD 1`.

The action `PASS` should only be sent if the client cannot play (for example if already folded, or all in). If checking, send `BET` with the current value put forward.

2.5 end_hand

Server sends the last iteration of actions, list of revealed cards and a list of pots won (at least one and maybe side pots):

```
end_hand
Action: <PASS or BET or FOLD> <value>
Action: <PASS or BET or FOLD> <value>
Action: <PASS or BET or FOLD> <value>
Showdown: <c0>,<c1>,<c2>
Pots: <value0>,<winner0> [<value1>,<winner1> ...]
```

where `<ci>` is one of the card values or '`-`' to indicate folded/mucked. To simplify things, the server implements automatic mucking. As with the initial set of actions, some of these actions may have been skipped over because the play ended at a certain point in the betting orbit. Those are all reported as `PASS <value>` where `<value>` is the amount in front of that player as usual.

Because the `<value>` always represents the money in front, for a game like Kuhn Poker with only one betting round, the total of all `<value>` reported in the last actions should match the total of all `<valuei>` pots.

Client responds with either confirmation or request to rebuy (while we're just doing Kuhn, this should always be `OK`):

```
<REBUY or OK>
```

TBD: For other games, how should rebuys be constrained? Should players be forced to rebuy if they go broke while the round is still going?

After receiving responses from all clients, the server sends out all of these actions to everyone:

```
EndAction: <REBUY or OK>
EndAction: <REBUY or OK>
EndAction: <REBUY or OK>
```

Finally, the client responds with their believed totals for each player as another way for the server to check that the bots understand the game state:

```
Money: <m0>,<m1>,<m2>
```

2.6 Interaction Structure

Bots can expect to always receive `init_round` immediately on start, followed by the first `init_hand`. For Kuhn Poker there is only either one or two `play`

messages, followed by the `end_hand` for each hand played. Each `end_hand` will be followed by either the next `init_hand` or `end_round` determining whether the game lives on or has been called off by the random number generator.

2.7 Example

It's probably best to include an example of a round of Kuhn Poker, with three bots playing. Black is server text, red is player 0's responses.

```
init_round
Money: 0,0,0
Blinds: 1,1,1
Button: 1
EndProb: 50,100
READY
init_hand
Hand: 0
Cards: Q
READY
play
Action: BLIND 1
Action: BLIND 1
Action: BET 2
BET 2
end_hand
Action: BET 2
Action: FOLD 1
Action: PASS 2
Showdown: -,-,K
Pots: 5,2
OK
EndAction: OK
EndAction: OK
EndAction: OK
Money: -2,-1,3
init_hand
Hand: 1
Cards: A
READY
play
Action: BLIND 1
Action: BLIND 1
Action: BLIND 1
BET 2
end_hand
Action: BET 2
```

```
Action: FOLD 1
Action: FOLD 1
Showdown: -,-,-
Pots: 4,0
OK
EndAction: OK
EndAction: OK
EndAction: OK
Money: 0,-2,2
end_round
Bankrolls: 0,-2,2
NumHands: 2
Thank you dealer, have a nice day!
```

In this round, only two hands were played because the end probability was so high (1/2). The bot playing broke even after the two hands. In the first hand, the button was on player 1, so there are initial `BLIND` actions for players 0 and 1 in the first `play` and a filler `PASS` for player 2 in the `end_hand` message. In the second hand, the button was on player 2, so all initial actions were reported as `BLIND` to player 0 since they were the first to act.

3 Handling Errors

In the name of easy debugging I think we should do fail-fast. In other words, if a bot responds or plays incorrectly during a round the entire round is cancelled and the error is reported. The server might have some sort of quarantine to test submissions against random bots and make sure a few rounds play through okay before adding the bot to the pool.

4 Reporting

I imagine running all submitted bots in a "tournament" format every hour (resources permitting) and publishing updated results in a small web server alongside the main game server. MIT's XVM should have enough resources to handle this, so we have a central location to host everything.

URL: `http://hssp.xvm.mit.edu/pokerbots/`