Design Document

Adaptable R.P.S.

Paul Council, Anand Patel

Before stating what will be handled in the design phase, it is important to note that the Adaptable R.P.S. project has had a few slight modifications concerning the overall structure since the initial specifications were released. The project will be refactored using the latest version of Python 2 (at this time is Python 2.7.x) as the we have decided this form of Python possesses a larger selection of network packages. Since Python 2 has been around for much longer than Python 3, we feel that any other packages that should be added on in the future would be easier to find, more robust overall, and more thoroughly tested given the amount of time since release.

The other change to mention is that the final release of this project will not support cross language Player implementation explicitly. Instead, this project will use an implementation of JSON (JavaScript Object Notation) in order for the Player class (that will be built in Python) to communicate with the framework. Since JSON is a general object, it is implied that a programmer with some experience using JSON should be able to create a Player class and apply this object to the project using their preferred version of a client supported JSON application. What this means is that a programmer proficient in Ada could design a player in Ada but would also need to implement a network interface adapter in order for the player to connect and speak with the running server.

We had a few different choices of transport mechanisms we looked into using for this project. The 3 main ones we researched were Thrift, ZeroRPC, and bjsonrpc. We made the final decision to use bjsonrpc due to it's simplicity, thorough documentation, and because it was designed to be asynchronous and "bidirectional". Another advantage is that bjsonrpc is being worked on and still receives updates unlike many options which appeared inactive or abandoned. Bjsonrpc is still somewhat limited in terms of server ability. At the time of writing this, bjsonrpc doesn't allow a server to perform operations on clients without the client currently making a call to the server. This means that the server can't gather a list of connections and send commands to any/all of these connections. Due to this limitation, we will have to refactor the current project style as it is based around the Observer/Observable pattern.

It should be noted that connections can not contact each other through normal means. Also, the server can not contact a connection that is not currently calling a function server side. These limitations can be overcome with intelligent backend code.

We ran through the initial tutorial to setup up a server to test how well the server could handle multiple calls in alternating orders from multiple clients and found it was simple and quick.

Client code:

```
import bjsonrpc
import time


c = bjsonrpc.connect()
for i in range(10):
    print "::>", c.call.hello("Client #")
    time.sleep(1)
```

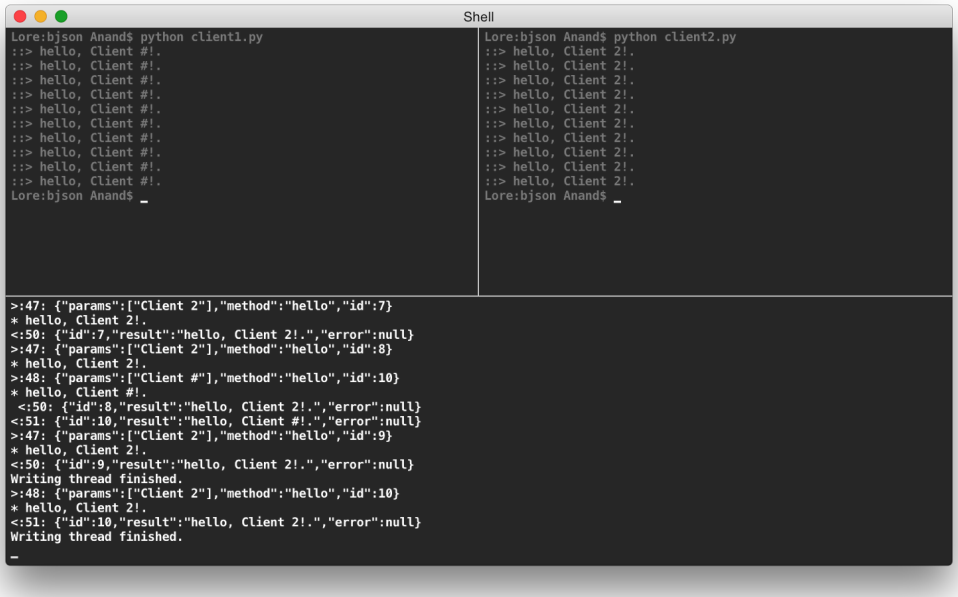
Server code:

```python
import bjsonrpc
from bjsonrpc.handlers import BaseHandler

class MyServerHandler(BaseHandler):
    def hello(self, txt):
        response = "hello, %s!." % txt
        print "*", response
        return response

s = bjsonrpc.createserver(handler_factory = MyServerHandler)
s.debug_socket(True)
s.serve()
```

Terminal output:



Development Questions and Answers

In order for the overall goal of this project to be clear, there were questions (applied in the form of statements) written in the specifications form of this project. The following is a summary of those questions (post changes) that this design document will attempt to answer:

1. *What types of languages will be required to implement this project?*

    a. This framework will be designed in the latest version of Python 2.7. Python was chosen due to its flexibility as a language type. Python possesses implicit object oriented design features as well as being a powerful scripting language. Python 2 in particular has a large library to choose packages from for this project.

2. *How will this game framework be implemented such that it is flexible enough to apply different types of Games and Tournaments without coding to every combination possibility?*

    a. The Tournament, Game, and Player classes are to be implement by the programmer using a concrete class. As long as this concrete class is implemented correctly, the project doesn't have any need to program to each possible combination. This would seem like essential object oriented design knowledge, but there are a couple of issues that we must consider when writing these classes:

        i. New types of tournaments are being created. If this project is to be as robust in the future as it is today, then our coding of each interface must handle the features that each tournament possesses. For example, instead of the Tournament class having a simple play() function, it has a series of functions that the implementing class can (and should) call such as start_match(), start_round(), play_match(), play_round(), etc. This format protects our current structure from needing to be reworked in the future.

        ii. New types of games are going to be created. Our framework does have limitations on the types of games that can be implemented, but the general rule is that if the game is similar to rock-paper-scissors, it should be playable in Adaptable R.P.S. Due to the potential complexity of the rules

of some games, this class is the most 'abstract' in terms of existing code. The Game class only possesses 3 functions:

1. num_players_per_game()
    a. The default for this function is two players.
2. get_result()
    a. Requires a moves tuple from the player. The programmer implementing the Game class (or extending it) must spend most of their time here. The game must know what to do if there is a tie, win/loss per player, and how to handle illegal moves.
3. is_legal()
    a. Determines if the moves tuple provided by the Player is in the legal list of expected moves. The get_result() function determines how to handle illegal moves in the Game.

3. *How will potential network issues be handled server/client side?*
    a. Our project is designed to be used over local network connections which does limit our possible network errors, though it doesn't eliminate them. Considering this limitation, we only need to consider errors in the server connection and errors in the client connection.
        i. Server Side
            1. There will be a TournamentServer class to open and maintain connections to the client's players. The server connection will require "states" (likely implemented in try/catch form) in order to try and stay in a persistent state. Our networking platform is bjsonrpc which explicitly states that stateful networking, when

used sparingly, can be used to hold a connection. The exceptions should be pushed up until they reach the GUI where the user will be alerted to the error and provided small pieces of advice to correct this. Any information that would have been changed due to this function call is still in the same state as the error structure made sure that it couldn't be altered.

2. The tournament should never be held in place entirely (unless there is a fatal error) which means that other matches can continue while the issue with the current player is being resolved. All scores before the current match will maintain value.

ii. Client Side

1. Handling this side is somewhat tricky as we are explicitly searching for JSON objects from the Player. The JSON is sent over in the form of a request and should be handled carefully in terms of error handling client side. A different error should occur if the server is never reached compared to the error when the server is reached but the request is timed out.

4. *What are the visual elements of this project?*

a. This project will have three graphical user interfaces (GUI), one displayed on the computer acting as the server, another displayed on the game administrators side, and the third displayed on the player's machine.

b. The server-side GUI has two responsibilities, as follows:

i. Find a unique ip address for the clients to connect to as well as a unique port address.

ii. Run regardless of the error encountered. The entire tournament shouldn't be stopped because a player is submitting the wrong information.

c. The player side GUI has multiple responsibilities:
    i. Allow the player to select the ip/port to connect to.
    ii. Give the client the ability to change their player to the selected game.
    iii. Print tournament information to the console.

d. The game admin GUI has multiple responsibilities:
    i. Allow the admin to select the ip/port to connect to.
    ii. Provides the admin the ability to change the tournament type and/or the game type.
    iii. Set the maximum number of players to the tournament.
    iv. Get the number of players currently registered.
    v. Print tournament information to the console.

5. *How will the tournament allow multiple matches to be played simultaneously in a timely fashion?*

    a. Our goal is to run the maximum number of matches at a given time. The normal way for a server to handle multiple events at once is to use threads. BjsonRPC has the options to enable threading, which will create a new thread to handle each incoming JSON item. However, there does not appear to be a simple way to limit the amount of threads created. Since the server can't call the connection without a connection currently calling the server then the clients will have to continuously ping the server for information. This will have to be handled gracefully to avoid timing errors as well as avoid overloading the server with requests.

6. *What errors will this project be able to handle?*

    a. Connection errors including error messages and an interruption in connection.

b. Incorrect input from the user. The player will receive a notification, and the host will have the option to remove the player from the tournament if the issue persists.
c. Incorrect output from the game. If this is a persistent issue it would require a rewriting of the given game class, but a singular issue could be fixed by having the match or round rerun.