

# Design Document

*Adaptable R.P.S.*

Paul Council, Jonathan Gonzoph, 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 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.

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 its simplicity and thorough documentation and because it was designed to be asynchronous and bidirectional. Another advantage of bjsonrpc is being worked on and still receives updates unlike many options which appeared inactive or abandoned. 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)
```

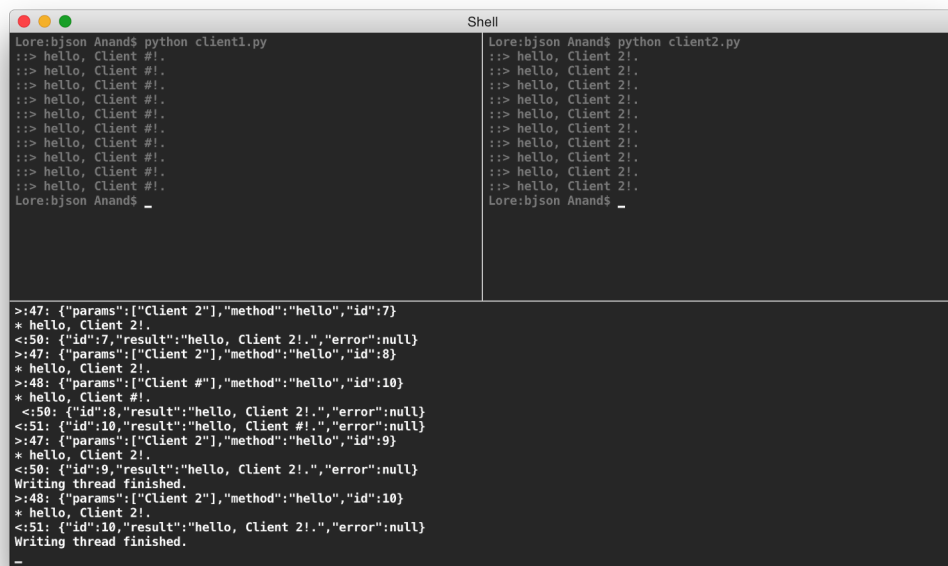
Server code:

```
import bjsonrpc
from bjsonrpc.handlers import BaseHandler

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

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

Terminal output:



```
Shell
Lore:bjson Anand$ python client1.py
[:> hello, Client #!
[:> hello, Client #!
[:> hello, Client #!
[:> hello, Client #!
[:> hello, Client #!
[:> hello, Client #!
[:> hello, Client #!
[:> hello, Client #!
[:> hello, Client #!
[:> hello, Client #!
Lore:bjson Anand$ _

Lore:bjson Anand$ python client2.py
[:> hello, Client 2!
[:> hello, Client 2!
[:> hello, Client 2!
[:> hello, Client 2!
[:> hello, Client 2!
[:> hello, Client 2!
[:> hello, Client 2!
[:> hello, Client 2!
[:> hello, Client 2!
[:> hello, Client 2!
Lore:bjson Anand$ _

>:47: {"params":["Client 2"],"method":"hello","id":7}
* hello, Client 2!
<:50: {"id":7,"result":"hello, Client 2!.", "error":null}
>:47: {"params":["Client 2"],"method":"hello","id":8}
* hello, Client 2!
>:48: {"params":["Client #"],"method":"hello","id":10}
* hello, Client #!
<:50: {"id":8,"result":"hello, Client 2!.", "error":null}
<:51: {"id":10,"result":"hello, Client #!.", "error":null}
>:47: {"params":["Client 2"],"method":"hello","id":9}
* hello, Client 2!
<:50: {"id":9,"result":"hello, Client 2!.", "error":null}
Writing thread finished.
>:48: {"params":["Client 2"],"method":"hello","id":10}
* hello, Client 2!
<:51: {"id":10,"result":"hello, Client 2!.", "error":null}
Writing thread finished.
_
```

## 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 implemented 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 only 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. For our purposes, this implementation should work. Connections managed in states makes our error handling somewhat easier as each state would require certain conditions in order to stay in the state, to move forward to the next state, and to move back the the previous state. This turns our server's error catching into a virtual finite-state machine. At this time, the exact states haven't been decided but it is clear we will need at least the following:

- a. s1\_register()

- i. The first state where we are expecting clients to attempt to connect to our server.

- b. s2\_play\_tournament()

- i. The second state where we have closed registration and the actual gameplay may commence. This state is by far the largest and should be the state that most of the tournament occurs in.

c. `s3_unrecoverable_error()`

- i. The state that this connection will move to if an error occurs that couldn't be handled in either state1 or state2.

- 2. As previously mentioned, most of the error handling will be applied to the second state which is where the game will occur. There will likely be a few states within state2 to handle possible error branches, though we would like to avoid this as too much state programming will result in a loss of data should a state change without sending the proper information across to the next state.

Given that this network connection is to be persistent, there will need to be a function in the TournamentServer that tries to call a `reconnect()` function in the client. This should try and reconnect the player to the tournament. Something like `reconnect` should be given limited automation as it could try to reconnect multiple times for a player that has simply disconnected. This is why the GUI for the tournament will possess an error screen where the one holding the tournament can manually call `reconnect` should a player lose connection for an unexpected reason. When a call to `reconnect` is made, the current match should freeze in place which means that the TournamentServer has the ability to halt processes of other classes that the current players are using. The tournament should never be held in place entirely unless there is a massive error which means that other games can continue while the issue with the current player is being resolved. For this `halt()` function to perform as expected, it must be applied on non-static objects and only halt the object that the player is currently using. The accompanying `resume()` function will attempt to resume the match

where it left off, though it is possible that it may need to be reset; 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. In order for this to be applied in the current form of the project and still possibly be expanded in the future to another language, we must add another layer between the player and the tournament. This layer, JLayer, would take the information send from the player and convert it to a simple JSON object that the rest of the classes can understand. The JSON definition would be defined to handle what the current project is expecting the Player class to send and any further changes would have to be applied by the programmer making the changes in the JLayer as well as the JSON object definitions.

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

- a. This project will have two graphical user interfaces (GUI), one displayed on the computer acting as the server and the other displayed on every computer hosting a player. The server-side GUI has two responsibilities, as follows:
  - i. The first is to allow the host of the tournament to see all the different games in progress, the players of those games, and the current score. The simplest way to do this is to create a visual list where each item shows the appropriate information, and allow this list to scroll if there are more games than would fit onto the screen. To accomplish this, the GUI will request information from the scorekeeper class, as well as information from the game class to keep track of errors.



- ii. The second responsibility is to alert the host of the tournament to any errors occurring in any of the games. This should be displayed on the list mentioned above, but upon clicking should bring up a panel where the host can see details about the error and, depending upon the specific error, also have options to resolve the error.
  - b. The player side GUI has only one responsibility; to alert the player that an error has occurred. Unlike the server-side GUI, it will not open a panel and allow for the player to attempt to resolve this error, as that will be handled by the host instead.
5. *How will the tournament allow multiple matches to be played simultaneously in a timely fashion?*

Our goal is to be running the maximum number of matches that can be going on 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, so a limit may have to be written if we encounter performance issues with larger tournament sizes. This limit can be disabled by the host, and will be indicated on the server-side GUI.

6. *What errors will this project be able to handle?*
- a. Connection errors including lost messages and an interruption in connection. In this event the player and host will both receive notifications but the server will attempt to reconnect to the dropped player or make an attempt to resend the lost message. Additional options by the host include manually sending a reconnect

request or request for the lost message, restarting the match entirely, and ending the match with the still connected player named the winner.

- b. Incorrect input from the user. Both the host and 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.