Push Documentation

Release 1.0

E. Bunao, N. Calabroso, K. Mendoza

October 09, 2014

Contents

CHAPTER 1

Connection

TCP is the protocol used to utilize its reliability despite its drawbacks in speed compared to UDP. Usually, in this type of games, UDP supposedly is the protocol used but the developers chose not to do so with the knowledge that the game is to be used in a local area setting, therefore rendering the differences in time delay, if not irrelevant, negligible for this setting. Since in-game collisions are of great importance, it is best that the exchange of messages is kept reliable and concurrent.

CHAPTER 2

The Application

Note: the `simplejson` library was used to serialize messages exchanged between end systems

2.1 Server

The server runs and acts as the receiver of messages using the *select method* and not threads. This is to reduce the load for the server during unnecessary waiting time. In the usage of threads, it is necessary for the system to maintain an infinite loop for each connection, and aside from these threads being memory and process heavy, so is idle time, making it not suitable for a time-sensitive singleton-type collision monitoring system. In the *select method*, processing is more efficient because messages by clients are processed by the server immediately only after receiving them, therefore ensuring that processing is done only when necessary.

2.2 Client

As per the application's architecture, the clients act simply as viewboxes or monitors (i.e. *slaves*) that redraws infinitely while expecting messages from the server. This is not the same case as what happens at the server side because no threads are run and the server application runs only one loop in itself, and so this design is still tolerable.

2.3 Connection Initiation

The server application `server_push.py` runs in the background.

The client applications `client_push.py` connect to the machine running the server application and, consequently, the server application itself.

Once the connection has been established between the server and a client via handshaking protocol (a subprotocol feature of TCP), the client sends an initial merrage to the server containing information about the user and its in-game character with the format:

```
[char_name, char_color, [x_pos,y_pos]]
```

where:
`char_name` is a string input from the user
`char_color` is a random color value generated by the client application
`x_pos` and `y_pos` are random float values generated by the client application

After the server receives this message, this list will be appended with a unique identifier in the format:

`ip_address_of_the_client!port_number`
e.g. **192.168.1.103!8080**

The server creates a game object with this information. It will then pass the succeeding logical processes to the game layer (world).

Moreover, with this design, anyone can connect to the game anytime as long as the server application is running and/or a game is ongoing, similar to the protocol of preexisting network multiplayer games like Counter Strike. Likewise, the game follows the mechanics of *last man standing*.

CHAPTER 3

Game Flow

Because the *select method* is followed by the server, it only processes clients that are able to send messages to it.

During the gaming phase, the client sends a message to the server in the format:

```
[unique_id, key_pressed]
```

where:
`unique_id` is a string assigned by the server at the initiation phase
`key_pressed` is an integer corresponding to the keyboard key pressed

At the moment the server receives this message, it will process it immediately, pass it through game logic, and broadcast the new state of the game field in the format:

```
[[object1,object2,...,objectN],game_state]
```

where:
`object1` is also a list in the format `[unique_id, [x_pos,y_pos]]`
`game_state` is a string that may carry a value of “game” or “end”

Game objects are included in what the server sends to the client since they are only viewboxes. We view this design of the software as advantageous, especially when there are only a number of users, since this singleton-type game logic processing allows for uniformity of information among clients and therefore consistent game and *world state*. Other designs can cause inconsistencies among clients because of distribution of logic and intermittent connection for one client can be destructive to the whole game and its clients, i.e. information can appear out of sync.

CHAPTER 4

Mechanics

1. Every character has 5 life points and 5 power units.
2. Life is reduced if a character is pushed by another to the walls.
3. The motion caused by a push will be cancelled out if a character collides with another player.
4. Every push costs the player one power unit.
5. A player is defeated and disappears from the field when its life points reduce to zero.
6. The player that remains alone in the field wins the game (*last man standing*).
7. Power-ups, life points and power units randomly appear in the field.

CHAPTER 5

Source

Fork us on GitHub: http://www.github.com/nmcalabroso/Push

Notes: | The source codes in Python for both client and server applications are in the link above. | The executable version of the client application is located in the *build* folder. Still, the `server_push.py` file must be used to enable network gaming and experience the game fully.

Bunao, Earle Randolph R.
Calabroso, Neil Francis M.
Mendoza, Kristoffer Marion L.