PacMan Online Multiplayer Game

E/13/073 Theekshana Dissanayake, E/13/043 Shashike Dissanayake

ABSTRACT

Pac-Man is an online multiplayer game developed using Java EE. At the beginning, each player establishes a connection to the server using HTTP requests and Server-Send Events (SSE) used to broadcast game state for all contestants. The game will only start when there are four contestants established the connection. Keypad events send to the server using appropriate JavaScript functions. Java thread lifecycle functions were used to control the broadcasting events. The four players score points by collecting colored dots on the grid. Red, green and blue dots count for one, two, three points respectively. When two players collide each other they lose three points each and each of them placed in different random locations. The game will end when all the food has been collected by players and it is automatically reloaded. To make the front end to generate the game state easily JSON object notation is used to exchange data via SSE. At a given event the corresponding JSON object response will contain the states of players and coordinates of the food. The players have a name and a current location and a score. And a coordinate contain with a corresponding color and x, y positions in the canvas.

CLIENT-SERVER ARCHITECTURE

Figure 1: Single client and server connection

Considering one single connection of a client, first, the client should send an HTTP request to the game server. Then the server responds with HTTP headers that indicate an event stream. Then the client waits for an event to generate. The event stream responds with a JSON object that contains the information about the contestants and the distribution of the current grid. It must be noted that there should be four connections (contestants) to start the game. Until four contestants being connected to the game server it will send events every 100ms interval. The contestants can see the number of contestants that are connected to the server via HTML message. After four contestants connected the server it will switch to the games play loop that game play state, the Thread is waiting for a POST request from the client. Then after the game grid is updated the thread will notify all the threads that are waiting. Then those threads call *sendData*() function to send the event stream.

PROTOCOL IMPLEMENTATION

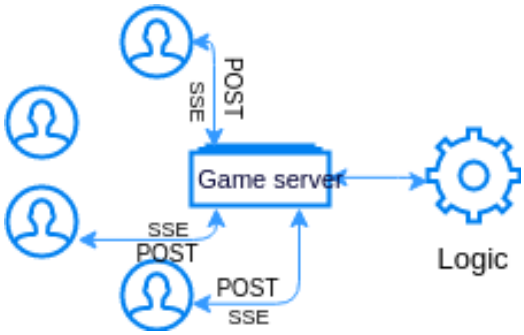

Figure 2: Multiple clients and server connection

At the moment when a client sends an HTTP request to the server, it checks weather the corresponding HTTP session is a new one or not. Depending on that it will create a session attribute called number indicating the player id and send headers for establishes a SSE. This id is an integer value generated using game logic. Then this session attribute is used to identify the client from his HTTP post request. Each player should wait for the event stream update. Also player should send a post request to the same server to move on the grid. Then in the server side it will determine the movement is valid or not. At the end of the game the server will reload the grid again with random arrangement.

FRONT-END LOGIC

In the front end of the code, appropriate JavaScript functions were written to display the grid. This grid is generated using a canvas object for a given size. The front end creates an EventLisner and waits for an event trigger using *event.onmessage*(). When an event triggered then it will get the JSON object from the event stream and look for "PLAYER" and "POINTS" attributes. Then those attributes were passed to corresponding JavaScript functions to draw the grid. Also in the same front end code, there is a keyboard

event listener. Event listener sends the keystroke value UP, DOWN, LEFT, RIGIT as an integer in 37 to 40 range. When an event is generated it will send that event to the game server using XMLHttpRequest post requests. This allows the user to send HTTP requests without loading the page.

GAME LOGIC

When the beginning of the server initialization it create a new game logic object from the class GameLogic. This logic class will generate a random grid for the contestants. Then it will wait for four contestants to connect. Then each player will get his own player id and the player can access the game using it. Update of the game grid has made synchronized since the updates should be thread safe. Then for a valid movement, the game grid will reduce the variable number of food for identifying the game over state. It must be noted that *gridtoCoordinates()* function and *getPlayers()* functions were not made synchronized because of those functions are accessing the grid after the update.

int SIZE	Size of the grid
int grid[SIZE[SIZE]	2D array used to store the information
int DOT_COUNT	Number of food in the grid
Random random	Java Random number generator
List<Players> players	Players information

Table 1: Important variables of GameLogic class

Table:1 shows the main set of variables that used in game logic. To generalize the grid SIZE is defined to change the size of the grid. The variable DOT_COUNT is used to identify the game over state. The hidden function *randomgrid*() generates exactly equal numbers of dots in the grid. The players of the game were stored in an ArrayList. The class diagram of players is given in the Table: 2. Then mainly game logic contains with 4 public functions as shown in the Table: 3. As mentioned *gameUpdate()* is a synchronized function used to update the grid (2D array). Then *gridtoCoordinates()* and *getPlayers()* used to form the JSON object. *ToString()* method of each of player and point were overridden to make the *toString()* of the list JSON compatible. *addplayer()* is the function that returns a player Id to a given HTTP session. This function is also synchronized to make thread safe.

int ID	*int Score*	*int x-coordinate*	*int y-coordinate*

Table 2: Class diagram of Player

gameUpdate(playerID, ketstroke)	The function used to update the game
gridtoCoordinates()	Return the points in JSON object
getPlayers()	Return the players in JSON object
addPlayer()	Add a player to the grid and return the ID

Table 3: Important functions of GameLogic class

EXTENDING THE GAME

- In this implementation, only four players are allowed to play the game at given time. But in a real scenario, the system should support any number of players. In that case, there should be a mechanism to make groups of players. In this implementation, there is only one GameLogic object. In a real scenario, there should be a List of GameLogic objects. To implement that the HTTP sessions should have another attribute something like the grid number. This method could be used to differentiate between several grids. Since this implementation can't check on that type of real scenario those logics were limited.
- Also in this implementation, the players do not get chances. They can play any time they want. In some time one player can eat all the food because of other three is not playing. This problem should also be handled.
- After all the food is finished the game automatically generates another random grid for players. So it will be more fun to players if the grid food is doubled in each level. It implements some kind of new logic and levels for the game.
- Also at a given instant only one player can make changes in the grid. Therefore the cases that two players colliding at the same time are eliminated. Also in this implementation when a 5^{th} tries to access the game server that person can't establish a SSE session with the server. This raises the problem that described before.