Multiplayer game (a decidir título)

Tiago da Silva Guerreiro and Lucas Costa dos Prazeres

Abstract

Web applications have become more popular in the recent years, due to the "appearence" of modern programming languages and frameworks which allows software developers to create better projects, in terms of performance and reliability. Such scenario has led to the popularization of many computer network expressions like websockets, which has been used more frequently along with platforms such as NodeJS, an asynchonous JavaScript runtime used to implement this programming language on the server side. This research aims to analyze these technologies in a high user friendly use case, a simple multiplayer game, which is the perfect oportunity to understand why would a developer or software engineer prefer to use a realtime communitation protocol on a real project, instead of the simple HTTP request/response. Moreover, we also present a small experiment intended to show how the packet drop parameter of websocket connections behaves in a few different scenarios.

Keywords

websocket, nodejs, socketio, multiplayer games, game architecture, TCP, realtime application

I. Introduction

II. Related Works / Technical Background

Discorrer brevemente sobre websockets (como a interação ocorre, suas etapas, etc). É interessante ser coisas que citaremos na parte da simulação.

III. The Game

The game is a simple web and multiplayer version of the classic snake game, in which a player needs to catch as many fruits as possible to earn more points.

Special thanks to professor Eduardo Cerqueira, ITEC, UFPA, Belém-PA, e-mail: cerqueira@ufpa.br

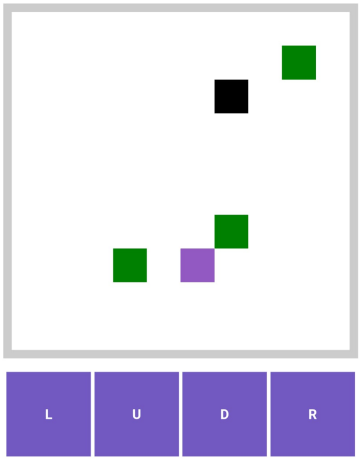

Fig. 1: Gameplay with 2 players and 3 fruits available

This project is composed by a few javascript files containing the main logic and network settings, besides an HTML with embed JavaScript with the game interface and client-side code. The backend was designed using Node.js, while the client code gets executed by the player's browser. It has a well-defined architecure, as shown on Figure 1.

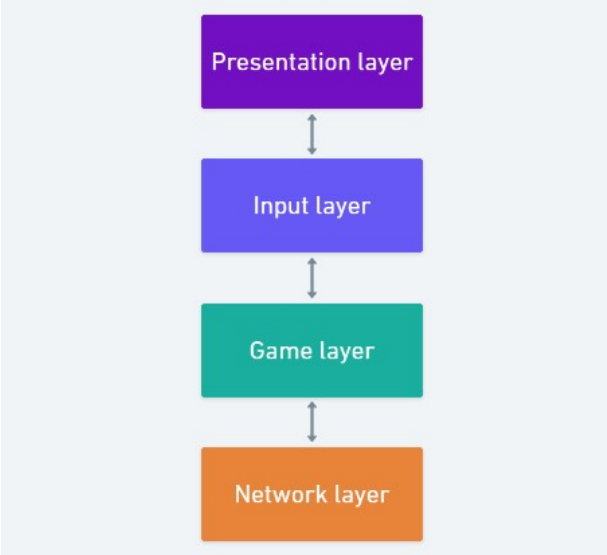

Fig. 2: Game architecture

The first layer from top to bottom is the presentation layer, which contains the HTML responsible for rendering the page, and also the required definitions for the other modules to work. The following layer is the input layer, which handles the user inputs and propagates them to next layer. The third layer is the game layer, composed by the game logic and rules applied to all its players, also being responsible for interpreting the commands sent by the input listenter layer above. At this level, the game state is represented as a data structure containing all player and fruit abstractions, which are manipulated locally, propagated to the server, and then to the other clients. The last one is the network layer, in which is possible to see the actual communication logic. Now, its important to understand how the game is setted

up and how the hosts interact with each other.

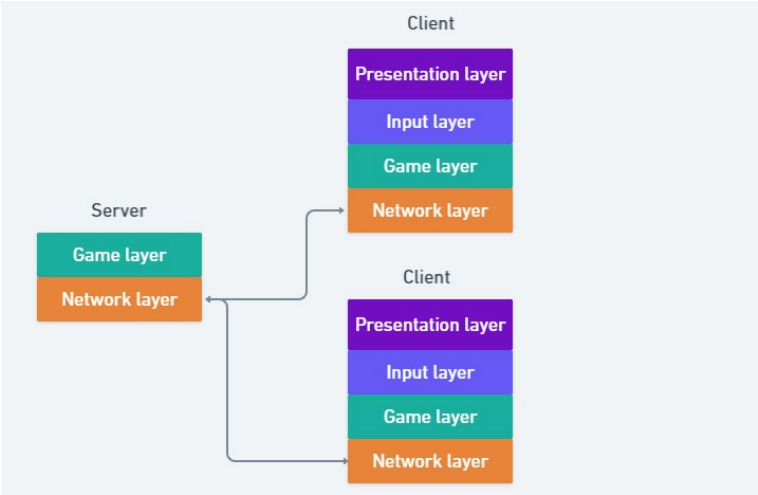

Fig. 3: Interaction between server and client

As shown in Figure 3, the game consists of a server and a few clients. When the server is executed, it uses the game layer to create its own game state and the Node.js libraries "express" and "socket.io" to create a web server and a websocket server, both listening on port 3000. Then, the client request the game page via regular HTTP request, which is reponded with the game HTML file and its javascript dependencies. After the clients receive the page, a websocket connection is started and the first message from server to the client contains the current game state data structure, which is used by the client side to render the game screen. Meawhile, on the server side, a new websocket connection is registered and an "add-player" message its propagated to all the previously connected clients and also changes the game state on the server. Every time a player moves, leaves the page or triggers another kind of event, its own game state is updated and a notification is sent to the server, which is responsible for syncronizing all players states by notifying them as well.

IV. Simulation

Fazer a simulação da perda de pacotes que o professor sugeriu, ou alguma outra interessante (talvez achemos alguma na leitura dos papers).

V. Conclusion

Concluir o trabalho daquele jeito tradicional, comentando sobre os resultados.

[1] Pra não dar erro.

References

[1] M. S. M. Ali, G. M. T. Abdalla, and M. A. H. Abbas, “Modified Ad hoc On-Demand Distance Vector (MAODV)Protocol,” in *2018 International Conference on Computer, Control, Electrical, and Electronics Engineering*

(ICCCEEE), 2018, pp. 1–6.