Quiz Game Using RabbitMQ: Enhancing Distributed Communication

Introduction

In the realm of the CALC unit (Intensive Calculation: Data Distribution and Computation), a project harnessing the capabilities of RabbitMQ was conceived. This endeavor aimed to showcase the middleware's abilities in facilitating communicating distributed applications, emphasizing fault tolerance, dynamic reconfiguration, and more, through an illustrative case study.

Originally considering a distributed machine learning algorithm, inspired by a past course, the project's direction shifted during an online gaming session with friends, pondering the creation of an online game like UNO. This project not only serves as a platform to master the middleware but also endeavors to craft an engaging game for friends and potentially a wider audience.

How to Use

Starting the Project

Ensure Docker is installed. Clone the repo. Initiate the gameplay by accessing the project and executing "docker compose up --remove-orphans --build". Access a web browser and navigate to the "localhost:80" website. Multiple pages can be opened, each representing a player.

Game Rules

- Each player joining or leaving triggers the start of a new game.
- Player addition during gameplay is prohibited.
- A maximum of ten rounds is allotted for each game.

Enjoy the game!

Access

For the front : http://localhost:80

For rabbitMQ with user & password "guest" : http://localhost:15672

Project Choices and Constraints

Choices Made

The project aimed to explore communication between two distinct programming languages using RabbitMQ. Additionally, it served as an avenue to refine JavaScript skills for an ongoing internship. Consequently, the project took shape as a website (utilizing HTML, CSS, CommonJS, & Node.js) with a Python3 server.

Constraints Encountered

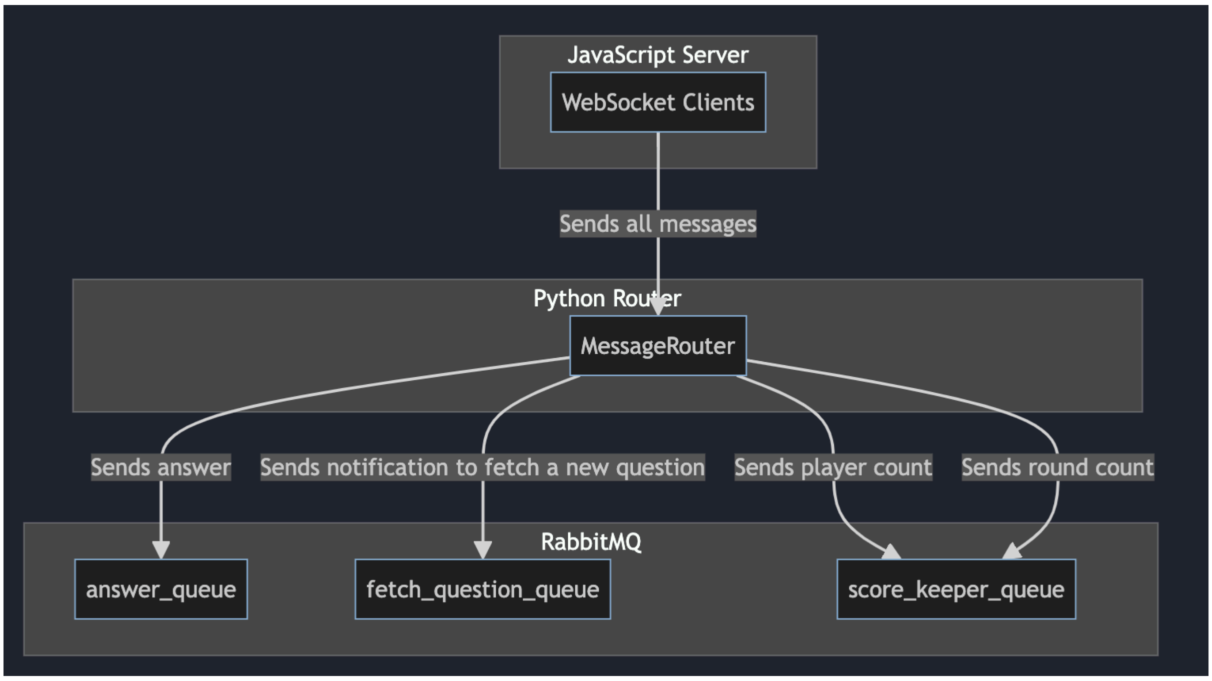
Adopting Node.js posed specific constraints, necessitating the use of a WebSocket to facilitate communication between HTML and Node.js due to JS's inherent limitations with amqpd.

Architecture of the Project

By using Vanilla JavaScript on the client side and a WebSocket-enabled server communicating through RabbitMQ with Python workers, we create a system that's fast, responsive, and adaptable. Vanilla JavaScript ensures a smooth user experience, while WebSockets enable real-time updates without constant page refreshes. RabbitMQ ensures reliable communication, crucial for handling more users, and Python workers help distribute heavy tasks efficiently. This straightforward setup, supported by a friendly developer community, allows for easy maintenance and future upgrades, making it an accessible and effective solution for real-time applications.

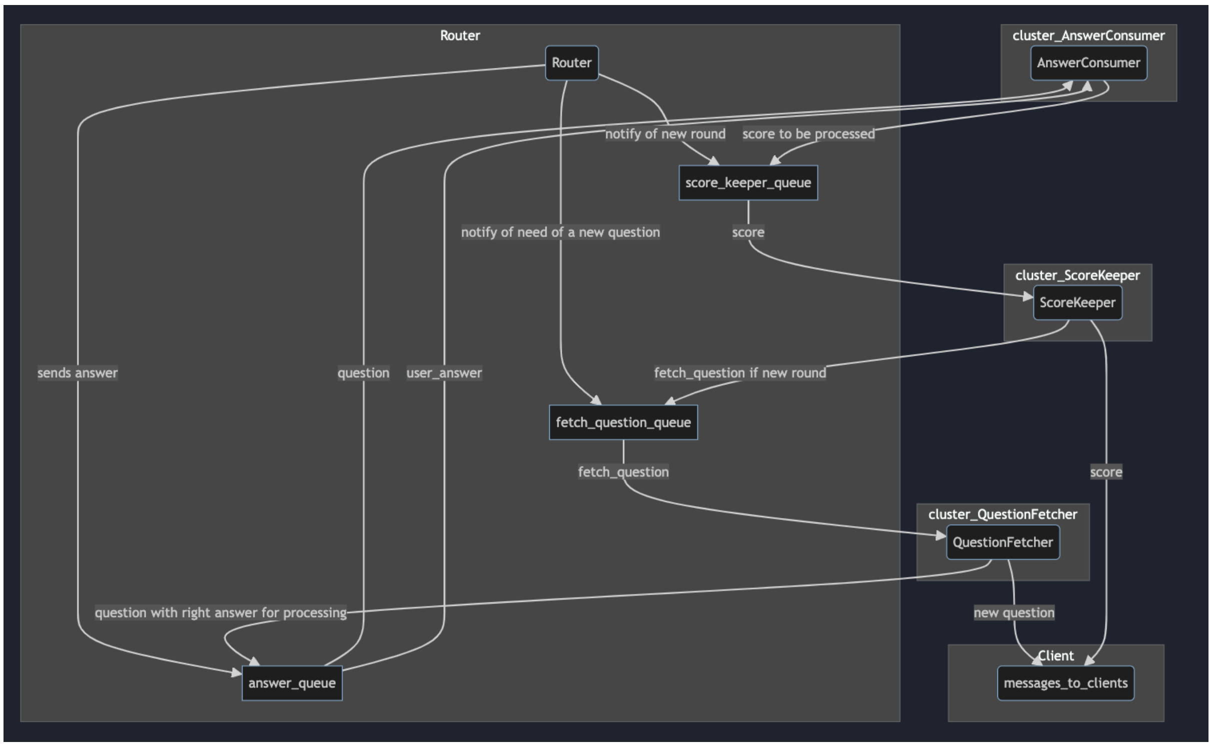
Python Router

I decided to use a router to receive all the messages from the client side server and redirect it to the workers. It’s a simple publish and subscribe.



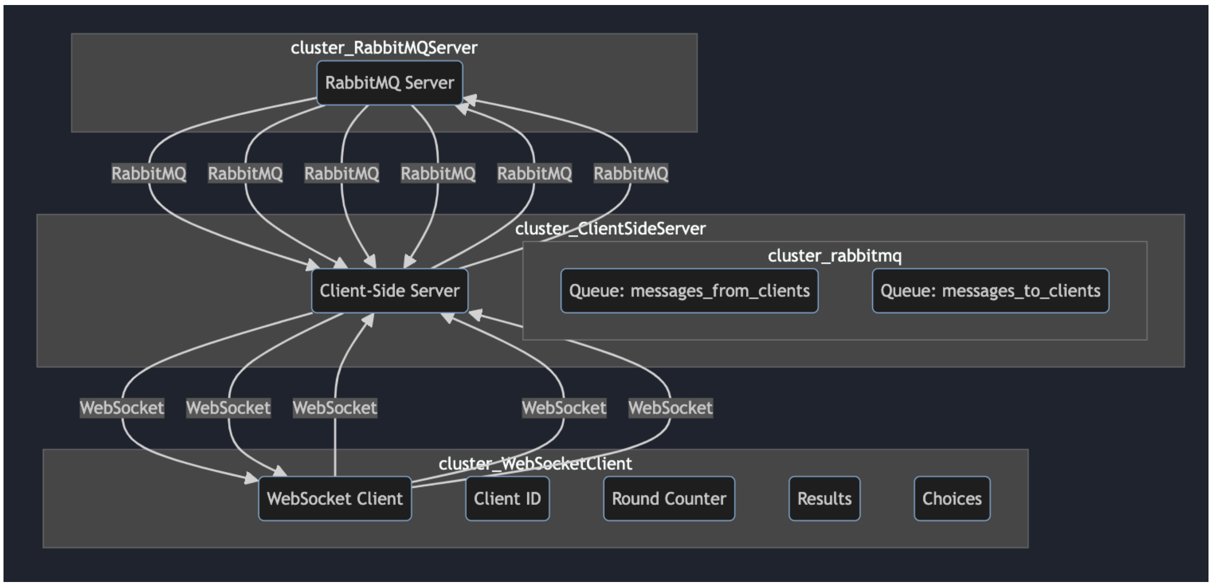
Python workers

The python workers then communicate between themselves to compute the score and also to fetch questions. It’s a simple publish and subscribe, but there are also topics.



Client side server and front

The **Client-Side Server** plays a pivotal role in managing WebSocket communication between the server and the client-side components of our Quiz Game. Responsible for handling real-time updates, user interactions, and facilitating seamless bidirectional communication, this server ensures a dynamic and responsive gaming experience. Utilizing the WebSocket protocol, it establishes persistent connections with the client-side WebSocket Clients, represented in the graph as the **WebSocket Client** cluster. These clients, comprising elements such as Client ID, Round Counter, Results, and Choices, are integral to rendering real-time game updates and user interactions. The communication between the Client-Side Server and the WebSocket Clients is orchestrated through RabbitMQ, as depicted by the arrows connecting them. This architecture enables efficient message passing, allowing for instant updates on game states, player actions, and other critical information, contributing to an engaging and interactive gaming environment.



Why RabbitMQ

RabbitMQ, chosen as the backbone middleware for this project, offers advantages that align seamlessly with the goals of enhancing distributed communication in the context of our Quiz Game. Here's a closer look at why RabbitMQ was the optimal choice:

> **Message Queues and Asynchronous Communication:** RabbitMQ excels in implementing message queues, providing a robust mechanism for asynchronous communication between different components of a distributed system. This proves instrumental in decoupling various parts of our game, enhancing fault tolerance, and allowing for scalable and efficient message processing. This is also useful in event driven architecture.

Reliability and Fault Tolerance: In a distributed environment, ensuring the reliability of communication is paramount. RabbitMQ's ability to persist messages and handle fault tolerance scenarios ensures that even in the face of unexpected errors or outages, the communication channels remain robust and resilient.
Dynamic Reconfiguration: The dynamic nature of distributed systems demands a middleware that can adapt to changes seamlessly. RabbitMQ supports dynamic reconfiguration, allowing components to join or leave the system without disrupting the overall communication flow. This flexibility is crucial for maintaining a responsive and adaptable gaming experience.
Scalability: The potential growth of the gaming application necessitates a scalable solution. RabbitMQ's support for scalable architectures enables our game to handle an increasing number of players and concurrent events without sacrificing performance.
Routing and Message Filtering: RabbitMQ's advanced routing capabilities provide an elegant solution for directing messages to specific components or services. This is particularly valuable in our game, where distinct messages need to be efficiently routed to different players or game instances, enhancing the overall responsiveness and user experience.
Wide Language Support: The project's decision to utilize two different programming languages benefits from RabbitMQ's extensive language support. This flexibility allows seamless communication between components developed in disparate languages, facilitating a diverse and efficient development environment.

In conclusion, RabbitMQ's feature-rich architecture, coupled with its reliability, scalability, and extensive community support, makes it an ideal choice for powering the distributed communication needs of our Quiz Game project.

Why a WebSocket

The decision to incorporate a WebSocket into the project, bridging the communication between the Node.js server and the Vanilla JavaScript on the client side, is driven by several compelling reasons:

Real-time Bidirectional Communication: WebSockets facilitate real-time, bidirectional communication between the server and clients. In the context of our Quiz Game, this instantaneous exchange of messages is essential for updating game states, player actions, and ensuring a seamless and interactive gaming experience.
Low Latency and High Performance: Unlike traditional HTTP requests, WebSockets maintain a persistent connection, eliminating the need for

repeatedly establishing and tearing down connections for each interaction. This low-latency, high-performance communication is crucial for delivering timely updates to players, especially in a fast-paced gaming environment.
Efficient Resource Utilization: WebSockets are designed to be lightweight, requiring minimal overhead for communication. This efficiency is particularly beneficial for our game, where a large number of concurrent connections are expected. The reduced resource consumption ensures a smoother gameplay experience for all participants.
Bi-Directional Event Handling: The WebSocket protocol enables both the server and the client to initiate communication. This bidirectional nature is invaluable for handling various events in the game, such as player moves, game state updates, and maybe chat messages, allowing for a dynamic and responsive user interface.
Compatibility with Vanilla JavaScript: Vanilla JavaScript seamlessly integrates with WebSockets, providing a straightforward and standardized approach for handling real-time communication. This compatibility simplifies the development process and ensures a consistent experience across different browsers and platforms.
Event-Driven Architecture: WebSockets align with the event-driven architecture commonly employed in modern web applications. This architecture is well-suited for our interactive game, enabling the server to push events to clients as they occur, such as notifying players of a new round or updating scores in real-time.
Reduced Server Load: By offloading real-time communication to WebSockets, the server is relieved of the constant need to handle numerous HTTP requests for updates. This reduction in server load enhances overall system scalability and ensures that server resources are optimized for other critical tasks.
Support for Cross-Browser Compatibility: WebSockets enjoy widespread support across modern browsers, ensuring a consistent and reliable communication channel for our game. This broad compatibility enhances the accessibility of the game, allowing players to participate seamlessly regardless of their choice of browser.
Used in class: During this UE, we mentioned using WebSocket multiple times. This was a chance for me to implement one in JS and learn how to use one in a personal project.

In summary, the adoption of WebSockets in our Quiz Game project serves as a strategic choice to deliver real-time, efficient, and bidirectional communication between the Node.js server and Vanilla JavaScript on the client side. This decision enhances the overall interactivity, responsiveness, and performance of the gaming experience.

Project Specifications & Future Improvements

- Implement Core Functionality
  - Answer a question
  - Get result
  - Number of players
- Refactor the JS
- Integrate Advanced Features
  - Implement Real-time Updates
  - Explore Gamification Options
  - Add rooms to create and choose from and create workers accordingly.
- Enhance User Interface