Report of CmPE48A Term Project

By Jorge Velázquez Jiménez

Links

Github: https://github.com/jorgeveji112/Chess-Project

Web ip: http://35.222.110.123

Video: https://drive.google.com/file/d/1dR4T2un5nbSxbPQ7vfgMt31BB_M_vZyK/view?usp=sharing

Introduction

This report outlines the development process and the technical implementation of a **Multiplayer Chess Application**, emphasizing the use of **Kubernetes** for deployment, scalability, and fault tolerance. The document delves into the architecture, cluster setup, and testing strategies, alongside an analysis of challenges faced and solutions implemented.

Project Overview

Objective

The primary goal of this project is to design and implement a Multiplayer Chess Application that enables users to engage in real-time gameplay. The application is engineered to ensure efficiency, scalability, and fault tolerance by leveraging Kubernetes as the deployment and orchestration platform.

Key Features

Real-Time Communication via Socket.IO

- **Technology:** The application employs Socket.IO, a robust and versatile library built on WebSockets and other real-time protocols.
- **Functionality:**
    - Manages bidirectional communication between players for seamless game synchronization.

- Facilitates low-latency interactions, ensuring that moves and chat messages are updated instantly.
- Implements event-driven architecture for key actions, such as creating/joining rooms, making moves, and handling disconnections.

Scalable Deployment Using Kubernetes

- **Kubernetes as Orchestrator:**
  - The application is containerized and deployed on a Kubernetes cluster, enabling it to scale dynamically based on traffic.
  - Ensures high availability by distributing the application across multiple pods and worker nodes.
  - Horizontal Pod Autoscaling (HPA) adjusts the number of pods to meet demand during peak usage.
- **Resilience:**
  - If a pod fails, Kubernetes automatically restarts it, minimizing downtime.
  - Load balancing across pods ensures even distribution of traffic.

Application Architecture

General Structure

The application follows a microservices architecture, dividing the system into modular and independent components. This approach ensures scalability, maintainability, and fault isolation. Each component is containerized using Docker and orchestrated through Kubernetes for efficient deployment and management.

Components Overview

Frontend

- **Technology:**
  - Developed with React.js for dynamic and efficient UI rendering.
  - Styled using SCSS, enabling modular and maintainable design.
- **Functionality:**

- Provides an intuitive and responsive interface for players.
- **Key Features:**
  - **Room Management: Al**lows users to create or join game rooms.
  - **Game Board:** Interactive chessboard that visualizes moves in real-time.
  - **Chat System:** Real-time communication between players during the game.
- **Containerization:**
  - The React application is built into a Docker image and exposed via a Kubernetes service (LoadBalancer).

Backend

- **Technology:**
  - Built with Node.js, leveraging Socket.IO for real-time communication.
- **Functionality:**
  - **Manages core logic and state, including:**
    - Game Logic: Validates moves, tracks game progress, and enforces chess rules.
    - Room States: Handles room creation, user assignments, and state synchronization.
    - User Connections: Manages player sessions using unique identifiers (UUIDs).
  - **Real-Time Communication:**
    - Socket.IO ensures low-latency updates for game states and chat messages.
    - Implements automatic reconnection and synchronization for disconnected players.
  - **Containerization:**
    - The backend service is encapsulated in a Docker container and exposed to other components via an ip service in Kubernetes.

Communication Flow

1. **Frontend and Backend:**
    - The frontend communicates with the backend over HTTP for APIs and WebSocket (via Socket.IO) for real-time updates.
2. **Kubernetes Services:**
    - Kubernetes services manage internal and external communications between pods, ensuring secure and efficient routing.

Benefits of Microservices Architecture

- **Scalability:**
    - Each component can scale independently based on resource demand.
- **Fault Isolation:**
    - Failures in one service (e.g., backend) do not directly affect other components (e.g., frontend).
- **Ease of Deployment:**
    - Containers allow consistent and portable deployments across environments.

Kubernetes Cluster Configuration

Overview

The Multiplayer Chess Application is deployed on a Google Kubernetes cluster to ensure high availability, scalability, and fault tolerance. Kubernetes dynamically manages the deployment, scaling, and operation of the application, providing a seamless user experience even during peak traffic periods.

Cluster Setup

Nodes

- **Master Node:**
    - Manages the control plane of the cluster, including scheduling, resource allocation, and cluster state management.
    - Runs essential Kubernetes components such as:

        - **API Server:** Handles communication between users and the cluster.
        - **Scheduler:** Allocates workloads to worker nodes.
        - **Controller Manager**: Ensures the desired state of the cluster.
        - **etcd:** A distributed key-value store for cluster configuration and state data.
- **Worker Nodes:**
    - Hosts the application pods.
    - Includes components like:
        - **Kubelet:** Communicates with the control plane to manage the lifecycle of pods.
        - **Kube-Proxy:** Manages networking rules for pod communication.

Namespaces

Namespaces are used to logically separate and manage resources within the cluster:

- **frontend:**
    - Contains pods for the React.js application and associated services.
    - Exposes the application to users via a LoadBalancer or Ingress.
- **backend:**
    - Hosts Node.js services that manage game logic and real-time communication using Socket.IO.
    - Includes internal ClusterIP services for secure communication with the database and other backend components.

Pods

Each application component is containerized and deployed as pods in the Kubernetes cluster:

- **Frontend Pod:**
    - Runs the React.js application and serves static assets.
    - Configured with resource requests and limits to optimize CPU and memory usage.

- **Backend Pod:**
    - Hosts the Node.js server with Socket.IO.
    - Handles API requests, Socket connections, and game state synchronization.

Services

Kubernetes services provide stable networking for pods:

- **LoadBalancer:**
    - Exposes the frontend and backend services to the internet.
    - Automatically distributes incoming traffic across multiple pods, ensuring fault tolerance and high availability.
    -

Deployment Details

1. **Horizontal Pod Autoscaler (HPA):**
    - Configured to dynamically scale pods based on CPU or memory utilization.
    - Ensures sufficient resources during traffic spikes.
2. **Vertical Pod Autoscaler (VPA)** dynamically adjusts CPU and memory allocations for pods based on real-time usageMonitoring and Logging:
3. **The Google Cloud console** provide an easy way to monitor all the services.

Benefits of Kubernetes Deployment

- **Scalability:**
    - HPA and VPA ensures that the cluster can handle sudden increases in user traffic.
- **Resilience:**
    - Pods are automatically restarted if they fail, minimizing downtime.
- **Resource Optimization:**
    - Resource requests and limits prevent over-provisioning and ensure efficient usage of hardware.
- **Isolation:**

- Namespaces segregate frontend, backend, simplifying management and troubleshooting.

Workflow

Containerization:

- Each application component (frontend and backend) is containerized using **Docker.**
- The container images are pushed to a secure container registry, **Docker Hub,** for seamless integration with Kubernetes.

Cluster Deployment:

- The application is deployed on a **Kubernetes cluster** by applying manifest files (Deployment YAML files) to define the desired state of the application.

Monitoring and Scaling:

- **The Vertical Pod Autoscaler (VPA)** is configured to dynamically adjust resource limits and requests, ensuring optimal performance during traffic spikes.
- **Google Kubernetes Engine (GKE)** console is used for real-time monitoring and visualization of cluster metrics, resource usage, and pod statuses.

Testing and Debugging

Stress Testing

- Tested the application's ability to handle high concurrent users by scaling up pods dynamically.
- In order to do this I developed a Node.js script that allows to simulate the socket connections to the backend. Simultaneus players join the game, half of them create a game while the others join them. Then they play a complete game until they finish.
- In the image, we can see two tests. The first one was conducted with 100 users, and the program worked properly without any issues. However, in the second test, with 1000 users, the program crashed due to the large number

of simultaneous users. Nevertheless, it is evident that after the crash, the pod restarted itself automatically.

Challenges and Solutions

Challenge 1: **Updating the Image of the Containerized Frontend in Kubernetes**

Issue:

During the deployment process, an issue arose when attempting to update the frontend container's image in the Kubernetes cluster. Despite building and pushing a new image to the container registry, the changes were not reflected in the application. This was due to Kubernetes continuing to use a cached version of the previous image.

Solution:

1. **Validated Image Update in the Registry:**
   Ensured that the updated image was successfully built and pushed to the Docker registry using the following commands

   docker build -t <image-name>:<tag> .

   docker push <image-name>:<tag>

2. **Updated the Kubernetes Deployment YAML:**
   Modified the Deployment manifest file to reference the new image tag explicitly

3. **Forced Kubernetes to Pull the Latest Image:**
   Added the imagePullPolicy field to ensure Kubernetes fetched the latest version:

4. **Applied the Updated Deployment Configuration:**
   Used kubectl to apply the updated YAML file and trigger a rollout of the new image:

   kubectl apply -f deployment.yaml

5. **Verified the Rollout:**
   Confirmed the changes were applied successfully by inspecting the pods:

   kubectl get pods

Result:

The updated frontend image was successfully deployed, and the changes were reflected in the application without further issues.

Challenge 2: **Testing with K6**

Issue:

While attempting to perform stress testing using K6, a popular tool for load testing, significant compatibility issues arose. K6 primarily supports pure WebSocket connections, but the application relied on Socket.IO, a library built on WebSockets with additional features such as automatic reconnection, multiplexing, and fallback options. This mismatch meant that K6 was unable to effectively simulate user behavior or test the system under load.

Solution:

To overcome this limitation, I developed a custom Node.js script designed to simulate multiple simultaneous users interacting with the application through Socket.IO. This script replicates real-world scenarios such as users creating rooms, joining rooms, making moves, and exchanging messages. The steps taken are outlined below:

1. **Custom Script Development:**
    - The script was written in Node.js to utilize the Socket.IO client for simulating user connections.
    - It included logic to:
        - Establish connections for a large number of virtual users.
        - Divide users into two groups: one half creating rooms and the other half joining them.
        - Simulate gameplay, including making moves until a match concludes.
2. **Key Features of the Script:**
    - Configurable User Count: Allowed easy adjustment of the number of users for different levels of stress testing.
    - Realistic User Interactions: Included delays between actions to mimic real-world behavior.
    - Performance Metrics: Measured response times, connection stability, and server load during peak usage.
3. **Testing Workflow:**
    - Initiated the script to simulate hundreds or thousands of users connecting simultaneously.

- Observed server behavior under load, including pod scalability and resource usage.
- Logged errors or timeouts to identify potential bottlenecks.

4. **Integration with Kubernetes Monitoring Tools:**
    - Monitored the cluster during tests using Google Kubernetes Engine (GKE) and other logging tools.
    - Ensured that Kubernetes scaled pods dynamically in response to the simulated load.

Result:

The custom script proved to be an effective alternative to K6, providing detailed insights into the system's performance under high load. It helped validate the application's scalability and fault tolerance capabilities, especially during peak traffic scenarios.

Technologies and Tools

- **Frontend**: React.js, SCSS.
- **Backend**: Node.js, Socket.IO.
- **Containerization**: Docker.
- **Orchestration**: Kubernetes.
- **CI/CD**: GitHub Actions.
- **Testing**: Node.js
- **Monitoring**: GKE console.

Conclusion

This project highlights the successful implementation of Kubernetes and Docker to build a scalable and resilient real-time multiplayer application. By containerizing the frontend and backend with Docker and orchestrating them using Kubernetes, I ensured the system could handle dynamic workloads while maintaining high availability. Key features like Vertcal Pod Autoscaler (VPA) and namespaces played a crucial role in optimizing resource allocation and managing different components effectively.

Through this process, I gained valuable experience in configuring Kubernetes clusters, deploying containerized applications, and addressing real-world challenges, such as updating container images and managing traffic spikes. These learnings have deepened my understanding of cloud-native technologies and their practical applications in building robust and efficient systems.

This project serves as a strong foundation for leveraging Kubernetes and Docker in future developments, enabling the creation of scalable and fault-tolerant applications with seamless user experiences.