**Project Overview**

This project aims to deploy a simple UDP game server on a Kubernetes cluster. The server is written in Python, and its primary purpose is to receive messages from clients and send a response back to the clients.

The server will run for 10 minutes, after which it will terminate.

The application is containerized using Docker and deployed on a Kubernetes cluster.

[1 Chosen Architecture 1](#_Toc132558957)

[1.1 Architecture Diagram 3](#_Toc132558958)

[1.2 Impacts in Production 4](#_Toc132558959)

[1.3 Issues in the Current Implementation 4](#_Toc132558960)

[1.4 Further Development Recommendations 4](#_Toc132558961)

[2 Code Review 5](#_Toc132558962)

[2.1 Server.py 6](#_Toc132558963)

[2.2 Client.py 7](#_Toc132558964)

[2.3 Dockerfile 8](#_Toc132558965)

[2.4 Main.tf 8](#_Toc132558966)

[3 Difficulty encountered during the development of the project 9](#_Toc132558967)

# Chosen Architecture

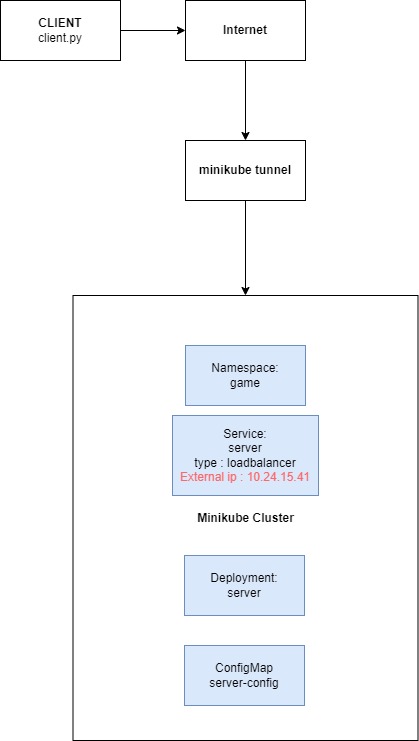
The chosen architecture for the game server is a microservices-based architecture, with the server being deployed as an independent service on a Kubernetes cluster. This architecture was chosen due to the following benefits:

1. Scalability: Deploying the server on a Kubernetes cluster allows it to be easily scaled based on demand, ensuring that the application remains performant even as the number of players increases.
2. High availability: Kubernetes provides high availability by automatically managing and restarting failed containers and distributing the application across multiple nodes.
3. Resource management: Kubernetes can efficiently allocate resources (CPU, memory, etc.) based on the needs of the deployed services.
4. Simplified deployment: Kubernetes allows for easy deployment and updates of the application, without requiring manual intervention or downtime.

In this section, we will discuss the chosen architecture for the game server deployment.

The project uses Minikube, a lightweight Kubernetes implementation, which simplifies local development and testing.

## Architecture Diagram



The primary components in the architecture are:

* Clients: Game clients that send and receive messages to/from the game server.
* Minikube Cluster: The Minikube environment where the game server is deployed. Minikube allows developers to run a single-node Kubernetes cluster on their local machine, making it easier to develop and test Kubernetes applications.
* Namespace: A dedicated Kubernetes namespace that isolates the game server's resources from other applications in the cluster.
* Deployment: A Kubernetes Deployment that manages the game server containers, ensuring high availability and easy scaling.
* Service: A Kubernetes Service that exposes the game server to the clients, handling load balancing and distributing traffic to the server containers.
* ConfigMap: A Kubernetes ConfigMap that stores the server's configuration (IP, port, etc.), making it easy to manage and update without modifying the server code.

## Impacts in Production

There are several potential impacts when deploying this game server in a production environment:

1. Increased complexity: The use of Kubernetes can introduce additional complexity when managing and deploying the application.
2. Resource utilization: Depending on the number of players and the server load, the application may require more resources than initially anticipated. Kubernetes can help scale the application, but additional infrastructure might be needed.
3. Security: Exposing the game server to the public internet can introduce security risks. Proper security measures should be implemented, such as network policies, authentication, and encryption.

## Issues in the Current Implementation

The current implementation has a few issues that should be addressed in further development:

1. Error handling: The server code does not include error handling for unexpected situations, such as receiving malformed data from clients. Adding error handling would improve the stability of the server.
2. Logging and monitoring: The current implementation lacks proper logging and monitoring, which are essential for troubleshooting issues and ensuring the health of the application in production.
3. Graceful shutdown: The server currently terminates abruptly after 10 minutes, which may cause issues for connected clients. Implementing a graceful shutdown process would allow the server to notify clients before shutting down and handling any ongoing requests.

## Further Development Recommendations

To improve the game server and address the issues mentioned above, the following recommendations should be considered:

1. Implement proper error handling in the server code to handle unexpected situations and increase the stability of the server.
2. Add logging and monitoring to track the server's performance, resource usage, and any issues that may arise during operation.
3. Utilize Kubernetes ConfigMaps or Secrets for configuration management, allowing for easier updates and management of the server configuration.
4. Implement a graceful shutdown process to notify clients before the server shuts down and handle ongoing requests without disrupting the user experience.
5. Enhance security by implementing network policies, authentication, and encryption for the game server's communication with clients.
6. Consider load balancing and session persistence in the Kubernetes cluster to ensure seamless user experience and efficient distribution of the server load.

**Testing and Continuous Integration/Continuous Deployment (CI/CD)**

1. Implement a comprehensive testing strategy that includes unit tests, integration tests, and end-to-end tests for the game server to ensure the reliability and stability of the application as new features are developed and deployed.
2. Set up a CI/CD pipeline that automates the building, testing, and deployment of the game server. This would ensure that any code changes are automatically built, tested, and deployed to the Kubernetes cluster, allowing for rapid iteration and minimizing the risk of human errors.

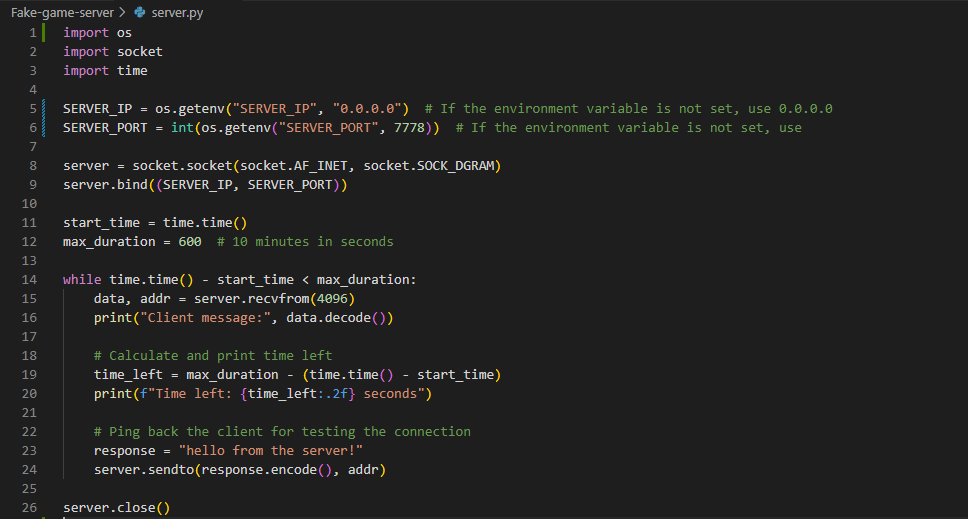
**Performance Optimization**

1. Profile the game server's performance to identify any bottlenecks or areas of the code that could be optimized for better performance, particularly in handling a large number of concurrent clients.
2. Investigate alternative server architectures or technologies that may provide better performance or scalability, such as using an event-driven architecture or implementing the server using a language known for high performance, like Rust or Go.

# Code Review

Now let’s take a look in our source code to explain how it’s working

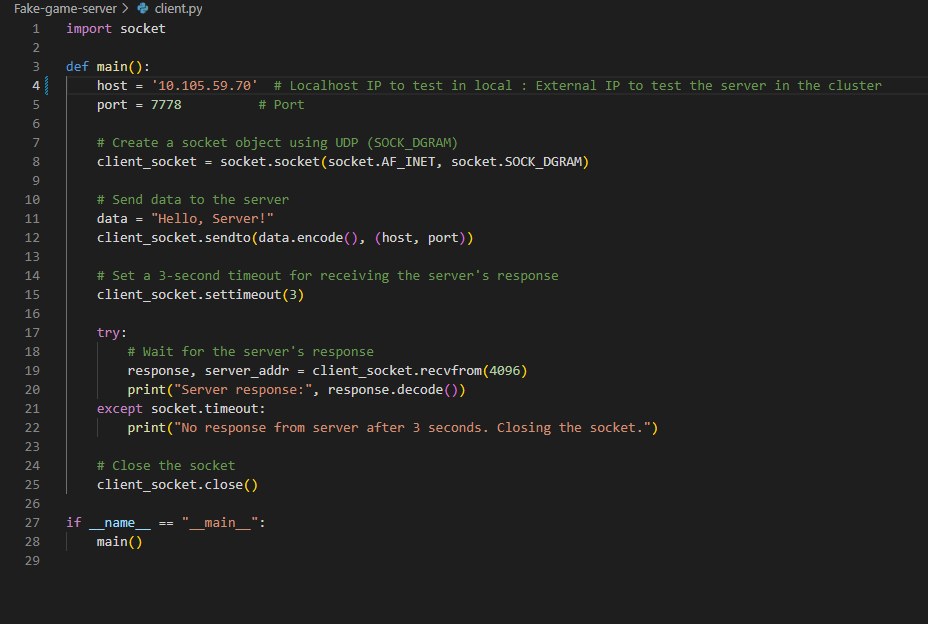
## Server.py



This Python script creates a simple UDP server that listens for incoming messages and responds back to the clients.

1. Import required modules:
   * **os**: To access environment variables
   * **socket**: To work with sockets and create a server
   * **time**: To measure time and limit the duration of the server
2. Get the server IP and port from the environment variables, or use the default values if they're not set.
3. Create a UDP socket and bind it to the specified IP and port.
4. Store the current time in **start\_time**, and set the maximum duration for the server to run (600 seconds or 10 minutes).
5. Run a while loop that listens for incoming messages from clients, as long as the elapsed time since **start\_time** is less than **max\_duration**.
6. In each iteration of the loop, the server receives a message and the client's address. It then prints the received message and the remaining time for the server to run.
7. The server responds back to the client with a "hello from the server!" message.
8. After the maximum duration has passed, the server closes the socket.

## Client.py

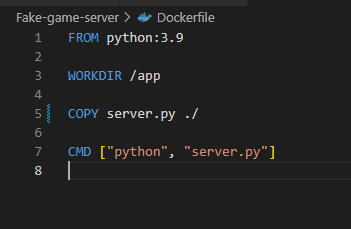


This Python script creates a simple UDP client that sends a message to a server and waits for a response.

1. Import the **socket** module to work with sockets.
2. Define the **main()** function, which contains the logic for the client.
3. Set the server IP and port. In this case, you've provided an IP address to test the connection with an external server or the localhost IP (10.105.59.70) and a port (7778).
4. Create a UDP socket using **socket.AF\_INET** and **socket.SOCK\_DGRAM**.
5. Encode the message ("Hello, Server!") and send it to the specified server IP and port using the **sendto()** method.
6. Set a 3-second timeout for receiving the server's response with the **settimeout()** method.
7. Use a **try**-**except** block to handle the possibility of a timeout. In the **try** block, receive the server's response with the **recvfrom()** method and print the decoded response. If the **except** block is triggered, it means the client has not received a response from the server within 3 seconds, so the script prints a message to inform the user and closes the socket.
8. Close the socket with the **close()** method.
9. Call the **main()** function when the script is executed.

This client script can be used to test the server script you provided in the previous question. Please ensure that the server script is running before running the client script

## Dockerfile



This is a Dockerfile for creating a Docker container running the server.py

1. **FROM python:3.9**: Use the official Python 3.9 base image from Docker Hub.
2. **WORKDIR /app**: Set the working directory to **/app**. If the directory does not exist, Docker will create it.
3. **COPY server.py ./**: Copy the **server.py** file from your local machine to the working directory inside the Docker container.
4. **CMD ["python", "server.py"]**: Define the default command to run when the Docker container starts. In this case, it will execute **python server.py**, which will run the server script.

## Main.tf

1. Configure the Kubernetes provider by specifying the path to the Kubernetes config file.
2. Create a Kubernetes namespace called **game**.
3. Create a Kubernetes ConfigMap called **server-config** in the **game** namespace, which stores the server IP and port as key-value pairs.
4. Create a Kubernetes deployment called **server** in the **game** namespace with the following specifications:
   * Set the replica count to 1.
   * Use the **jackypaul06/server:latest** image (replace this with the actual image name if different).
   * Set the **PYTHONUNBUFFERED** environment variable to "1" to allow logs to be displayed immediately.
   * Use the **server-config** ConfigMap to set the **SERVER\_IP** and **SERVER\_PORT** environment variables.
   * Expose port 7778 with the UDP protocol for the container.
5. Create a Kubernetes service called **server** in the **game** namespace with the following specifications:
   * Select the deployment with the label "app=server".
   * Use the "LoadBalancer" type to expose the service to the internet.
   * Define a service port for the UDP protocol with port 7778 and target port 7778.

# Difficulty encountered during the development of the project

The main difficulty I encountered was that I used WS2 with docker desktop on windows to set up my environment and deploy my project.

The problem with wsl2 and minikube is a network connection problem, I could not communicate from outside my cluster.

When I use minikube tunnel to allocate an external ip to my service, it allocates me a local ip (127.0.0.1)

I tried several methods:

**Nodeport** :

nodeport using a port forward to my UDP port 7778 but Portforward only works with TCP.

**Socat :**

I tried to use socat to proxy my pods, it didn't work either.

I finally abandoned the idea of using WSL2, I reinstalled everything on my windows using hyperv as a driver for minikube and it solved my problem.