Advanced Monitoring and WebSocket-Driven In-Game Communication

This guide will help you:

1. Set up monitoring tools (Prometheus/Grafana) to track performance and logs.

2. Integrate WebSocket-driven in-game communication for real-time Pong gameplay.

---

1. Monitoring Tools with Docker

Overview

Prometheus and Grafana provide a robust solution to monitor metrics like CPU usage, memory, response time, and WebSocket connections.

---

Set Up Monitoring with Prometheus and Grafana

Step 1: Update docker-compose.yml

Add Prometheus and Grafana services:

version: '3.8'

services:

backend:

build:

context: .

ports:

- "4567:4567"

volumes:

- .:/app

env\_file:

- .env

depends\_on:

- db

- prometheus

command: ["ruby", "app.rb"]

db:

image: postgres:15

environment:

POSTGRES\_USER: ft\_user

POSTGRES\_PASSWORD: securepassword

POSTGRES\_DB: ft\_transcendance

volumes:

- db\_data:/var/lib/postgresql/data

prometheus:

image: prom/prometheus

volumes:

- ./prometheus.yml:/etc/prometheus/prometheus.yml

ports:

- "9090:9090"

grafana:

image: grafana/grafana

ports:

- "3000:3000"

volumes:

- grafana\_data:/var/lib/grafana

volumes:

db\_data:

grafana\_data:

---

Step 2: Configure Prometheus

Create a prometheus.yml file in the project root:

global:

scrape\_interval: 15s

scrape\_configs:

- job\_name: 'backend'

static\_configs:

- targets: ['backend:4567'] # Backend container

---

Step 3: Start the Monitoring Tools

Run the containers:

docker-compose up --build

---

Step 4: Access Prometheus and Grafana

Prometheus: http://localhost:9090

Grafana: http://localhost:3000

---

Step 5: Configure Grafana

1. Login to Grafana (default: admin/admin).

2. Add Prometheus as a data source:

URL: http://prometheus:9090.

3. Create a dashboard:

Visualize metrics like HTTP request counts, response times, or WebSocket connections.

---

Step 6: Monitor Backend Metrics

Use Ruby to expose Prometheus-compatible metrics:

1. Add the prometheus-client gem:

gem 'prometheus-client'

Install the gem:

bundle install

2. Update app.rb:

require 'prometheus/client'

# Initialize Prometheus metrics

Prometheus::Client.configure do |config|

config.logger = Logger.new(STDOUT)

end

metrics = Prometheus::Client.registry

http\_requests = Prometheus::Client::Counter.new(:http\_requests\_total, docstring: 'Total HTTP requests')

metrics.register(http\_requests)

before do

http\_requests.increment(labels: { path: request.path })

end

get '/metrics' do

metrics.text

end

3. Access the metrics at /metrics:

curl http://localhost:4567/metrics

---

2. WebSocket-Driven In-Game Communication

Overview

WebSockets allow real-time updates for in-game actions like paddle movement, ball position, and scores.

---

Implement Real-Time Pong Communication

Step 1: Define Game State

Store the game state (e.g., paddle positions, ball position) in memory:

game\_state = {

players: [],

ball: { x: 50, y: 50, dx: 1, dy: 1 },

paddles: { player1: 40, player2: 40 }

}

---

Step 2: Update WebSocket Logic

1. Notify players of the current state:

require 'json'

get '/game\_ws' do

if Faye::WebSocket.websocket?(env)

ws = Faye::WebSocket.new(env)

ws.on :open do |\_event|

# Send initial game state

ws.send(game\_state.to\_json)

end

ws.on :message do |event|

data = JSON.parse(event.data)

# Update paddle positions

if data['player'] == 'player1'

game\_state[:paddles][:player1] = data['paddle\_position']

elsif data['player'] == 'player2'

game\_state[:paddles][:player2] = data['paddle\_position']

end

end

ws.on :close do |\_event|

puts "WebSocket connection closed"

end

ws.rack\_response

else

halt 400, 'WebSocket endpoint only'

end

end

---

Step 3: Update Ball Position Periodically

Add a periodic timer to update the ball position:

require 'eventmachine'

Thread.new do

loop do

sleep 0.05

# Update ball position

game\_state[:ball][:x] += game\_state[:ball][:dx]

game\_state[:ball][:y] += game\_state[:ball][:dy]

# Check for collisions

if game\_state[:ball][:y] <= 0 || game\_state[:ball][:y] >= 100

game\_state[:ball][:dy] \*= -1

end

# Broadcast the updated game state to all clients

clients.each { |client| client.send(game\_state.to\_json) }

end

end

---

Step 4: Client Integration

1. Connect to the WebSocket:

const ws = new WebSocket("ws://localhost:4567/game\_ws");

ws.onmessage = (event) => {

const gameState = JSON.parse(event.data);

console.log("Updated game state:", gameState);

};

// Send paddle position

document.addEventListener("mousemove", (e) => {

ws.send(JSON.stringify({ player: "player1", paddle\_position: e.clientY }));

});

2. Render the game state in the browser:

Update the canvas based on received game state.

---

Testing In-Game Communication

1. Start the backend:

ruby app.rb

2. Open multiple browser clients and connect to /game\_ws.

3. Move paddles and watch the real-time synchronization.

---

Next Steps

Refine game mechanics:

Add scoring logic.

Handle player disconnections.

Implement AI to simulate a player.

Add monitoring for WebSocket events.

Would you like help building the AI player or refining the in-game mechanics?