🎮 *NexusQuiz* - Distributed Trivia Game

Authors: Imesh Nimsitha, Que Hung Dang, Randeep Bhalla, Nathen Fernandes
Department: Computer Science, Western University
Date: 2025-04-04
Tags: computer science distributed systems

📝 Problem Description

Our project addresses the challenge of creating a *distributed multiplayer trivia game* that maintains *consistent game state* across *multiple nodes* while being *fault-tolerant*. The application needed to handle:

- Maintaining *consistent game state* for all players
- *Recovering* from primary node *failures* gracefully
- *Distributing player load* across *available* nodes
- Providing *real-time* gameplay interaction
- Ensuring *fault tolerance* without losing game progress

🛠️ System Design and Architecture

We designed a *distributed system* with the following key components:

1. **Primary-Backup Architecture with RAFT Consensus:**
   We implemented the *RAFT* consensus algorithm to handle *leader election* and ensure *consistent game state* across nodes. When a primary node *fails*, RAFT *elects* a new leader to take over without *disrupting* gameplay.
2. **Consistent Hashing for Data Distribution:**
   Player connections and game data are *distributed* across nodes using *consistent hashing*. This approach minimizes redistribution when nodes join or leave the *cluster*, improving system stability during scaling events.
3. **Real-time Communication Layer:**
   WebSockets (via Socket.IO) enable *real-time bidirectional* communication

between servers and clients, allowing immediate *updates* for *game state* changes, questions, answers, and scores.

4. **Frontend-Backend Integration:**
   A *Next.js* frontend communicates with our *distributed* backend, handling both *REST API* calls for game management and *WebSocket* connections for real-time updates.

The system follows this overall workflow:

- Players create or join game sessions through *any available node*
- Requests are *forwarded* to the current *RAFT leader* if necessary
- Game state changes are *replicated* to all nodes via *RAFT log*
- Players receive real-time updates through *WebSocket* connections
- If a node fails, *RAFT elects* a new leader and gameplay continues

📌 Implementation Details

Our implementation includes:

Backend:

- *FastAPI* framework for HTTP endpoints and WebSocket integration
- *Custom RAFT* implementation for leader election and log replication
- Node health *monitoring* and automatic *failover*
- Game *session management* with timed question rounds and scoring

Frontend:

- *Next.js* application with responsive UI for different devices
- *Socket.IO* client integration for real-time updates
- *State management* to handle game flow and player interactions
- *Dynamic routing* for game sessions and player navigation

🚧 Challenges and Solutions

Leader Election and Consensus:
Implementing *RAFT* correctly was challenging, particularly handling *edge cases* like network partitions. We *simplified* our approach while maintaining *core* safety properties, ensuring we always had *exactly one* leader managing each game session.

State Consistency:
We faced challenges maintaining *consistent game state* across all nodes. Our solution was to process all game state *changes* through the *RAFT log* sequentially, preventing *race conditions* and ensuring all nodes had *identical* views of each game.

Node Failure Recovery:
When nodes *failed* during gameplay, we needed to *recover* without *disrupting* the user experience. We implemented mechanisms for nodes to *catch up* on *missed* log entries when rejoining, ensuring seamless *recovery*.

Real-time Performance:
Maintaining *low latency* for answer submission and score updates was critical. We optimized our *WebSocket* configuration and minimized *message payload sizes* to ensure responsive gameplay even with *multiple concurrent* game sessions.

📊 Results and Evaluation

Our distributed trivia game successfully demonstrates:

1. **RAFT-based Leader Election:**
   When the primary node *fails*, a new leader is *automatically elected* to continue managing game sessions. This process typically completes within *2-5 seconds*, minimizing disruption to gameplay.
2. **Consistent Hashing for Load Distribution:**
   Player connections are effectively *distributed* across *available* nodes, with minimal redistribution when nodes join or leave the *cluster*.

3. **Fault Tolerance:**
   The system *continues* functioning even when individual nodes *fail*. Game state is preserved through *replication*, and players can reconnect to continue gameplay.
4. **Real-time Multiplayer Gameplay:**
   Players can join trivia sessions, answer questions within time limits, and see leaderboard updates in *real-time*, all while the underlying distributed system handles *consistency* and *fault tolerance*.

The system was tested with *multiple concurrent* game sessions and *simulated* node *failures* to verify its robustness.

✅ Conclusion

This project successfully implements a *distributed multiplayer trivia game* using *RAFT* consensus for *leader election* and *consistent hashing* for data distribution. The system demonstrates practical application of distributed systems principles in a real-world context.

Through this implementation, we gained valuable experience with distributed systems challenges, particularly in maintaining *consistent state* across nodes and *handling failures* gracefully. The *RAFT algorithm* proved effective for *leader election* and *log replication*, while *consistent hashing* provided efficient data distribution.