pAIcMan: CS262 Final Project
Technical Write-Up

Matthew Vu, Pedro Garcia, Samuel Kim

May 7, 2025

1 Overview and Objectives

Purpose. This project aimed to design and implement a real-time, cross-device Pacman game featuring networked multiplayer gameplay. The system utilizes a fault-tolerant Python server cluster, built with `pysyncobj` for Raft consensus, to maintain global game state and consistency. Lightweight Python clients, using Tkinter for the GUI, connect to the server to render the game and handle player input. Communication between clients and the server cluster is handled via gRPC, enabling potential interoperability between heterogeneous devices.

High-Level Goals. The primary objectives set for this project were:

1. *Networked Multiplayer:* Allow multiple players (one Pacman, one or more Ghosts) to participate in the same game session across different machines.
2. *Real-Time Synchronization:* Maintain a responsive gameplay experience with reasonably low latency between player actions and state updates, targeting a 60Hz refresh rate.
3. *Fault Tolerance:* Ensure the game state persists and gameplay can continue even if a minority of server replicas (up to f in a $2f + 1$ cluster) fail.
4. *Modularity:* Design the system with distinct client and server components communicating via a well-defined API (gRPC).
5. *Technology Exploration:* Gain practical experience with gRPC, consensus protocols (Raft via `pysyncobj`), and asynchronous programming in Python.

2 Technology Stack

The following technologies and libraries form the foundation of the pAIcMan project:

- **Programming Language:** Python 3.9+.
- **Core Libraries:** Built-in modules like `asyncio`, `threading`, `queue`, `logging`, `argparse`.
- **Communication Protocol:** gRPC with Protocol Buffers (using `grpcio` and `grpcio-tools`) for defining the client-server API and handling remote procedure calls.
- **Consensus Algorithm:** Raft, implemented using the `pysyncobj` library, to provide fault tolerance for the server state.

- **Client GUI:** Tkinter, Python's standard GUI toolkit, used via the `helpers.graphicsDisplay` module adapted from the original Berkeley Pacman projects.
- **Logging:** Python's standard `logging` module for recording server and client events.

3 System Architecture

3.1 gRPC API Definition (pacman.proto)

The client-server interaction is defined by the Protocol Buffer specification in `pacman.proto`. This contract outlines the services, RPC methods, and message structures used for communication.

Key Services and RPCs.

- **Service `PacmanGame`:** Encapsulates the game-related operations.
- **RPC `CreateGame`:** Allows a client to request the creation of a new game session on the server cluster. Takes layout name and max players, returns game ID, assigned player ID, and layout name.
- **RPC `ListGames`:** Enables clients to query the server for currently active or waiting game sessions.
- **RPC `PlayGame`:** A bidirectional streaming RPC forming the core of gameplay. Clients send `PlayerAction` messages (containing intended direction), and the server streams back `GameState` updates.

Core Message Structures.

```
// Represents a player's intended action (typically movement)
message PlayerAction {
  string player_id = 1;
  string game_id = 2;
  Direction direction = 3;
}

// Represents the state of a single agent (Pacman or Ghost)
message AgentState {
  string agent_id = 1;        // Unique ID for this agent
  AgentType agent_type = 2;   // Pacman or ghost
  Position position = 3;      // Current position
  Direction direction = 4;    // Current direction
  bool is_scared = 5;         // Whether ghost is in scared state
  int32 scared_timer = 6;     // Time remaining in scared state
  string player_id = 7;       // ID of controlling player
}

// Represents the complete state of a game session at a point in time
message GameState {
  string game_id = 1;
  repeated AgentState agents = 2;  // All agents (Pacman and ghosts)
  repeated Position food = 3;      // Remaining food pellets
  repeated Position capsules = 4;  // Power pellets
  repeated Position walls = 5;     // Wall positions
```

```
  int32 score = 6;                    // Current score
  GameStatus status = 7;              // Game status
  string winner_id = 8;               // ID of winning player (if game is
      finished)
  optional Position food_eaten = 9;    // Position of food eaten this tick
      (if any)
  optional Position capsule_eaten = 10; // Position of capsule eaten this
      tick (if any)
  GameMode game_mode = 11;            // Game mode (PVP or AI_PACMAN)
  AIDifficulty ai_difficulty = 12; // AI difficulty level for AI_PACMAN
      mode
}

// Other messages like Position, SessionInfo, CreateRequest, etc.
// Enums like Direction, AgentType, GameStatus
```

Listing 1: Key Messages from `pacman.proto`

3.2 Server Cluster & Raft Consensus (PySyncObj)

To achieve fault tolerance, the server logic is deployed as a cluster of identical replicas coordinated by the Raft consensus algorithm, facilitated by the `pysyncobj` library.

- **Replication Model:** The server runs as multiple instances (typically 3 or 5 for 1 or 2-fault tolerance) of the `PacmanServicer` class. This class inherits from `pysyncobj.SyncObj`, providing the Raft machinery.
- **Replicated State Machine:** The core game state (active sessions, player assignments, positions, scores, food layout) is the state machine managed by Raft. Operations that modify this state (e.g., creating a game, player joining/leaving, processing player actions, updating game state) are implemented as methods within `PacmanServicer` or `GameSession` that are invoked via methods marked with the `@replicated` decorator in `PacmanServicer`. `PySyncObj` ensures these decorated methods are executed only after the corresponding command is committed in the Raft log across a majority of replicas.
- **Leader Role:** Raft elects a single leader among the replicas. Only the leader directly handles client requests that modify state (like `CreateGame` or receiving `PlayerActions` within the `PlayGame` stream). The leader is also responsible for running the main game loop for active sessions.
- **Follower Role:** Follower replicas receive log entries from the leader via `pysyncobj`'s internal mechanisms and apply the committed commands to their local state machine, ensuring they stay synchronized. They typically do not interact directly with clients for write operations or game streaming.
- **State Persistence:** `PySyncObj` handles log persistence and snapshotting (configured via `SyncObjConf`) to allow replicas to recover their state after restarts.

3.3 Server Game Loop and State Management

Each active game session on the server is managed by a `GameSession` object, which runs its own update loop in a separate thread.

- **Tick Rate:** The game loop aims for a 60Hz tick rate, controlled by calculating the elapsed time for each update cycle and sleeping for the remaining duration of the target interval (1/60th of a second).
- **State Updates:** Within the loop (`GameSession.update_loop`), the server performs the following actions for each active game, but only if it is the current Raft leader:
    1. **Process Actions:** Retrieves queued player actions (directions) that have been received via the `PlayGame` RPC and committed via Raft (using `_replicated_player_action`).
    2. **Update Agent Positions:** Calculates the next position for each agent (Pacman and Ghosts) based on their current position, direction, and the layout's walls using logic from `helpers.game.Actions`.
    3. **Handle Collisions/Events:** Detects collisions with walls, food pellets, power capsules, and between Pacman and Ghosts. Updates the score, removes eaten food/capsules, manages scared ghost timers, and determines win/loss conditions.
    4. **Generate State Proto:** Creates a `pacman_pb2.GameState` protobuf message representing the new state.
    5. **Replicate State:** Uses a `@replicated` method (`_replicate_game_state`) to ensure the updated game state (score, agent positions, food, capsules, status) is consistently stored across the Raft cluster. This method includes the logical timestamp (`state_version`).
    6. **Broadcast State:** Sends the newly generated `GameState` message to all connected clients in that session via their respective `PlayGame` streams.

3.4 AI Pacman Implementation

As an extension to the standard PVP gameplay, our system implements an "AI Pacman" mode where human players control ghosts while a server-side AI agent controls Pacman. This mode adds significant strategic depth and variety to the gameplay experience.

- **AI Agent Architecture:** The implementation follows a class hierarchy with a base `AIPacmanAgent` class and three difficulty-specific implementations:
    - **Easy:** Uses randomized movement that avoids walls and slightly prefers continuing in the same direction for more natural movement.
    - **Medium:** Employs a modified greedy approach that targets nearby food while evaluating ghost proximity to avoid dangerous situations.
    - **Difficult:** Implements a sophisticated strategy balancing food collection, ghost avoidance, and strategic power capsule usage, with different evaluation weights based on the game state.
- **Server Integration:** The server manages the AI agent with a special player ID `"ai_pacman"`. During the main game loop in `update_loop()`, if the mode is `AI_PACMAN`, the server calls `update_ai_pacman()` which:
    - Creates a simplified game state representation (`SimpleGameState`) with essential data for the AI agent
    - Invokes the AI agent's `get_action()` method to determine the next move
    - Validates and applies the move, including food consumption checks

- **Raft Consistency Model:** The AI agent's actions are deterministically derived from the distributed game state and thus are also part of the consistent state machine managed by Raft. This ensures that all server replicas maintain consistent AI behavior even during leader changes.

3.5 Logical Clock for State Synchronization

The server uses a simple timestamp-based mechanism to help clients order incoming game states, especially important given network latency variations.

- **Timestamping:** The leader assigns a monotonically increasing timestamp (milliseconds since epoch, stored in `state_version`) to each `GameState` it generates before replication and broadcast.
- **Client-Side Use:** The client's `GameStateAdapter` receives these states. While the primary consistency guarantee comes from Raft on the server-side, this timestamp allows the client to potentially identify and handle or discard out-of-order updates received over the network, ensuring smoother rendering based on the latest server-acknowledged state.

4 Client Architecture

4.1 Initialization and Menus

- Upon launch, the client displays a main menu with options to create a new game, list existing games, or join an ongoing game session.
- The client uses gRPC stubs generated by `grpcio-tools` to call the `CreateGame` and `ListGames` RPCs.

4.2 Client Architecture

The client application is designed to be lightweight and responsive, focusing purely on rendering the game state received from the server cluster and transmitting player actions back. It connects to the server using gRPC, establishing a bidirectional stream for real-time gameplay communication via the 'GameStream' RPC defined in our '.proto' file.

Key aspects of the client architecture include:

- **Asynchronous Communication:** Leveraging Python's 'asyncio' library and 'grpc.aio', the client handles network communication asynchronously. This ensures that the user interface remains responsive even while waiting for server updates or sending actions.
- **Server- State Rendering:** The client does not perform any game logic simulation. It solely relies on the 'GameState' messages received from the server cluster via the bidirectional stream. Upon receiving a new state, it updates the local display (using Pygame) to reflect the authoritative positions of Pac-Man, ghosts, pellets, and scores.
- **Action Transmission:** Player inputs (keyboard presses for movement) are translated into simple 'PlayerAction' messages (e.g., specifying direction 'UP', 'DOWN', 'LEFT', 'RIGHT') and sent asynchronously to the server over the gRPC stream.

- **State Synchronization Awareness:** While the primary synchronization mechanism (Raft and logical clocks) resides on the server, the client implicitly benefits from it by receiving consistently ordered state updates.
- **Connection Management:** The client implements basic logic to handle connection establishment and graceful termination. It includes retry mechanisms with exponential backoff to manage transient network issues or temporary server unavailability during leader elections or node restarts, aiming to provide a smoother player experience.

4.3 AI Pacman Mode Support

The client architecture has been extended to support the AI Pacman mode with a seamless user experience:

- **Mode Selection Interface:** The GUI includes radio buttons that allow players to select between traditional PVP mode and AI Pacman mode when creating a game. When AI Pacman mode is selected, additional UI elements for difficulty selection dynamically appear through the `toggle_difficulty_display()` method.
- **Difficulty Settings:** Players can choose between three AI difficulty levels (EASY, MEDIUM, HARD), which are communicated to the server via the `GameConfig` protocol buffer message during game creation.
- **Role Assignment:** In AI Pacman mode, the client's `join_game()` method automatically assigns all human players to ghost roles, adapting the player experience accordingly. The UI clearly displays "AI" as the controller of the Pacman character.
- **Game Session Discovery:** The game listing interface shows the game mode and AI difficulty for each available game session, allowing players to quickly identify and join AI Pacman games that match their preferred challenge level.
- **Rendering Adaptation:** The `GameStateAdapter` translates the server-provided game state (including AI Pacman's position and actions) into the appropriate visualization without requiring mode-specific rendering logic, maintaining a clean separation between game logic and display.

4.4 Deployment and Network Configuration

Given the project's scope and focus on core distributed systems concepts, the deployment strategy centers on local network execution. The server cluster nodes and clients are intended to be run on separate machines within the same local network or on a single machine using different ports for simulation.

- **Server Cluster Setup:** Each server node is configured with the addresses of its peers for Raft communication. A discovery mechanism or static configuration file (‘config.json‘) is used to initialize the cluster.
- **Client Connection:** Clients connect to a known entry point (e.g., one of the server node addresses). In a more robust setup, a load balancer or discovery service could abstract the specific server nodes from the client.

- **Network Considerations:** Running on a local network minimizes latency, which is crucial for a real-time game like Pac-Man. However, the design incorporates fault tolerance (via Raft) and client-side retries to handle potential packet loss or node failures, simulating challenges found in wider network deployments. Firewall configurations must allow traffic on the gRPC and Raft communication ports between the participating machines.

A cloud deployment (e.g., using Docker containers managed by Kubernetes on a platform like GCP, AWS, or Azure) would be a natural next step for broader accessibility but was outside the primary scope of this academic project.

4.5 Testing Strategy

A multi-layered testing approach was adopted to ensure correctness and robustness:

- **Unit Tests:** Focused on isolated components:
  - **gRPC Interface:** Testing serialization/deserialization of Protobuf messages ('GameState', 'PlayerAction', etc.).
  - **Raft (pysyncobj):** Leveraging the library's own tests and adding specific checks for our state machine application logic (applying game states).
  - **Server Game Logic:** Testing core game functions (e.g., 'move$_p$*acman*', '*move*$_g$*host*', '*check*$_c$*ollisions*', *score*
- **Integration Tests:** Testing interactions between components:
  - Client-Server Communication: Simulating a client connecting, sending actions, and receiving game state updates via the gRPC stream.
  - Server Cluster Interaction: Launching a small cluster (3 nodes) and verifying leader election, state replication, and failover behavior under simulated node failures.
- **Manual Testing:** Playing the game under various conditions (different layouts, numbers of ghosts, simulated network delays/failures) to identify usability issues and edge cases missed by automated tests.
- **Static Analysis:** Using tools like Ruff/Flake8 for linting and MyPy for static type checking to catch potential errors early and maintain code quality.
- **CI/CD (Conceptual):** Although not fully implemented with a dedicated CI server for this project, the structure facilitates automation. A pipeline (e.g., GitHub Actions) would ideally be configured to automatically run linters, type checkers, and unit/integration tests on each commit or pull request.

5 Project Milestones

The project development followed these approximate phases:

1. **Foundation & Setup:** Initial project structure, environment setup (Python, Pygame, gRPC tools), basic Pac-Man game logic (single-player, non-distributed).
2. **gRPC API Definition:** Designing and implementing the '.proto' file for client-server communication (game creation, state streaming, actions). Generating initial Python stubs.

3. **Basic Client-Server:** Implementing a single-server version where a client connects via gRPC, sends actions, and receives game state updates.

4. **AI Pacman Mode:** Implementing an alternative gameplay mode where Pacman is autonomously controlled by an AI agent while human players control ghosts. This required designing a three-tiered difficulty system. The server architecture was extended to manage the AI agent's decision-making within the main game loop while maintaining the distributed consensus guarantees. The client UI was enhanced with mode selection, difficulty configuration, and appropriate role assignment for an intuitive player experience.

5. **Raft Integration:** Integrating 'pysyncobj' library, setting up the Raft cluster configuration, defining the replicated state machine to handle game state.

6. **Distributed State Management:** Modifying the server logic to run within the Raft framework, ensuring state changes are proposed and committed through the consensus protocol. Implementing logical clocks for ordering.

7. **Fault Tolerance & Refinement:** Testing leader election, node failure scenarios, client reconnection logic. Refining the game loop and synchronization mechanisms.

8. **Testing & Documentation:** Writing unit and integration tests, performing manual testing, and drafting this technical write-up.

5.1 Lessons Learned

This project provided valuable insights into building distributed, real-time applications:

- **Complexity of Consensus:** Implementing and debugging distributed consensus (even using a library like 'pysyncobj') is challenging. Understanding the nuances of Raft (leader election, log replication, commit index) is crucial for correct operation and troubleshooting. State machine design must be deterministic and idempotent.

- **gRPC for Real-time Communication:** gRPC's bidirectional streaming is well-suited for real-time applications like games, providing an efficient and structured way to handle continuous updates. However, careful management of the stream lifecycle and error handling on both client and server is necessary.

- **State Synchronization Challenges:** Ensuring consistent game state across clients when the authoritative state resides on a potentially changing set of server nodes requires careful design. Logical clocks helped in managing causality, but the core consistency relies heavily on the underlying consensus protocol.

- **Asynchronous Programming:** 'asyncio' is powerful for handling I/O-bound tasks like network communication but introduces its own complexities regarding task management, error propagation, and debugging compared to synchronous code.

- **Trade-offs in Fault Tolerance:** Achieving fault tolerance (e.g., 2-fault tolerance with a 5-node Raft cluster) comes at the cost of increased communication overhead and complexity compared to a single-server architecture. The choice depends heavily on the application's availability requirements.

- **Importance of Testing:** Thorough testing, especially integration testing that simulates network partitions or node failures, is indispensable for validating the correctness of distributed systems.

Future work could involve exploring alternative consensus protocols, implementing more sophisticated client-side prediction, deploying to a cloud environment, or enhancing the AI capabilities of the ghosts within the distributed framework.