PEER-TO-PEER Survivor Decision Game

- Pablo Sánchez Gómez

AI Disclaimer

"During the preparation of this work, the author(s) used DeepSeek to add comments and clarify the code, translate on-screen messages from Spanish to English, to review potential structural errors that are not easily visible, and to assist with mechanical and unimportant tasks such as applying the same format to all available scenarios. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the final report/artifact."

Abstract

The Peer-to-Peer Survivor Decision Game is a multiplayer, text-based adventure game where three players collaborate to survive on a deserted island. The game is designed to run over a peer-to-peer (P2P) network, allowing players to connect through a central server, eliminating the need for them to determine and share their own IP addresses. Once connected by the central server, players will continue with their peer-to-peer connection without the server's intervention. Players make decisions that affect the outcome of the story, with multiple branching paths and endings. The game emphasizes collaboration, decision-making, and real-time synchronization between players. Through this project, we aim to gain insights into distributed system management, P2P implementation, and how to enhance learning through a game-based approach.

Concept

This project is a Peer-to-Peer multiplayer game where three players must collaborate in a survival scenario. The game uses a command-line interface (CLI), where players initially connect through a central server, which identifies their IP addresses using Google's DNS server. This allows the system to determine their public IPs for internet connectivity, without requiring them to perform this action manually.

- Use case collection
    - When and How Often: Players interact when the game presents them with scenarios that require decisions. Interactions occur as needed, typically in real-time.
    - How: Players communicate via peer-to-peer networking protocols over local or internet networks, and the game responds to the decisions made by each player.
    - Devices: The game is designed for desktop systems running Python.
    - Data Storage: User data is not stored persistently; all game states are maintained in memory during the session. All game state is maintained in memory during the session.
    - Roles: There are several entities: 3 different players, the server, and the bots. Later, we will explain in more detail how they interact with each other.

Requirements

- Functional Requirements
    - Peer-to-Peer Connection: The game must allow three players to connect directly over a network using IP addresses and ports.
    - Real-Time Decision Synchronization: Players must be able to make decisions simultaneously, with real-time synchronization ensuring consistency across all participants..
    - Branching Scenarios: The game must provide multiple branching scenarios based on player decisions, leading to different outcomes and endings.
    - Game State Management: The game must maintain and synchronize the state of the game between both players.
    - Restart Option: Players must be able to restart the game after completion.
- Non-Functional Requirements
    - Low Latency: The game must ensure minimal latency in decision synchronization to provide a smooth user experience.
    - Fault Tolerance: The game must handle network disconnections gracefully and notify players of any issues.
    - Cross-Platform Compatibility: The game must run on any system with Python 3.x installed.
- Implementation Requirements

- Programming Language: Python 3.8.
- Libraries: The game uses the socket library for networking and threading for concurrent operations.
- No External Dependencies: The game should not require additional libraries or frameworks beyond Python's standard library.

Design

Architecture

The system follows a hybrid model that combines:

- **Central Matchmaking Server**: Coordinates the formation of 3-player groups
- **P2P Mesh Communication**: Direct connections between the 3 players once matched
- **Resilience Mechanisms**: Replacement bots and asynchronous synchronization

Infrastructure

- ***Components**:*
  - **Matchmaking Server** (server.py):
    - Manages player registration
    - Forms 3-player groups
    - Distributes connection information between peers
  - **P2P Clients** (main.py + p2p.py):
    - 3 interconnected game instances
    - Automatic reconnection mechanism
    - Bot system for replacement in case of disconnections
  - **Bot Manager**:
    - Autonomous entities that take control in case of failures
    - Random decision-making logic with a time limit

- **Network Distribution:** Players on the same network (LAN) using local IPs
- **Component Discovery:**

```
sequenceDiagram
    Player->>Server: Registration (IP, Port, User)
    Server->>Player: Wait for 2 more
    Server->>Group: Notify IPs upon completing the trio
    Player1->>Player2: Direct connection
    Player1->>Player3: Direct connection
    Player2->>Player3: Direct connection
```

Modelling

- ***Domain Entities:***
  - **Matchmaking Server**:
    - Maintains player queue
    - Manages 3-player groups
    - Provides initial connection information
  - **Game Session**:
    - Coordinates shared state among the 3 or 2 players
    - Synchronizes decisions through consensus
    - Manages transitions between scenarios
  - **Bot**:
    - Autonomous reactive entity
    - Acts after 30 seconds of inactivity
    - Makes random decisions
  - Player:
    - Make decisions to continue the game
    - Use an username to represent himself
- **Domain Events:**
  - Server registration
  - Formation of the 3-player group
  - Initial P2P synchronization
  - Collaborative decision-making
  - Bot activation/deactivation
  - Consensus-based session restart
- **Messages**: JSON-encoded messages containing game state and decisions.
- **State**: The current scenario, player decisions, and game outcome.

Interaction

- **Communication:**
  Players communicate via TCP sockets, exchanging JSON-encoded messages.
  Each message contains:

- Type (decision, retry, bot_update).
- Sending player (name or identifier).
- Content (numerical decision, restart confirmation, or bot actions).

The message format ensures interoperability between game instances, allowing real-time synchronization of the game state.

- **Interaction Patterns**:
  - Decision Synchronization (P2P Consensus)
    Each player sends their decision (1 or 2) to the others via broadcast. The system waits until it receives 2/3 of the decisions (in the case of 3 players) before processing the result. If a player does not respond within 30 seconds, they are replaced by a bot that makes a random decision.
  - Coordinated Restart
    When a game ends, players vote (y/n) to restart. Unanimity is required among active players (humans).
  - Bot Handling
    Bots are activated after a timeout (30s) to maintain game dynamics and prevent network disconnection issues. They replace human players, sending random decisions to keep the game flow.

Behaviour

- Game Instance
  - Incoming Connections:
    Each instance opens a server socket on a random port (5000-6000) to accept connections from other players.
  - State Synchronization:
    The received decisions are stored in a dictionary (received_decisions). When enough votes are received, the result is calculated based on the decisions made, and the scenario is updated.
  - Game Update:
    Scenario transitions are determined using a decision matrix (transitions in game.py). The state (current scenario, decisions, active players) is maintained in memory and reset upon restart.
- Player
  - Decision Making:
    User input (1 or 2) is validated locally to decide which scenario to continue. User input (y or n) is validated locally to decide whether to restart the game or not. The decision is sent to all peers via send_data().
  - Synchronization Waiting:

If decisions are missing, the game enters a waiting loop with a timeout (30s). If the timeout is reached and the expected decisions have not been received, the system checks how many decisions are missing from the expected ones. The missing players are added to a bot list to account for them in future iterations, and their decisions are replaced with random decisions. Once this point is reached, players will continue waiting only for players they know are still active, and stop waiting for disconnected players. Once the game ends, it can be replayed with bots only to check different scenarios.

- Server
  - **Listening for Client Connections:**
    The server begins listening on a specific port (5000) and accepts incoming connections from clients (players) who want to join the game.
  - **Client Handling:**
    Each time a player connects, a thread is created (using `threading.Thread`) to handle that client's connection concurrently without blocking the main server.
  - **Player Registration:**
    The server receives player information (such as username, IP address, and port) in JSON format via a socket. This information is decoded and then added to a waiting list of players.
  - **Formation of Player Groups:**
    When a player connects, the server adds them to the waiting queue (`players_waiting`). Once there are at least three players in the queue, the server forms a group of three players.
  - **Distribution of Connection Information:**
    Once the three players are grouped, the server distributes the connection information of the other two players to the corresponding player. This means each player in the group of three will receive the IP address and port of the other two players to connect with each other (via P2P).
  - **Closing the Connection with the Player:**
    After the server sends the peer information to a player, the player's socket is closed, terminating the connection with the matchmaking server.
  - **Concurrent Access Synchronization:**
    The server uses a threading lock to ensure that access to the waiting list of players is safe, preventing race conditions when modifying the queue of waiting players.

Data and Consistency Issues

- Data Storage: No persistent storage is used. All data is stored in memory.
- Consistency: The game ensures consistency by synchronizing the game state after each decision.

Fault-Tolerance

- Is there any form of data **replication** / federation / sharing?
    - The game implements **asymmetric data replication** through JSON message broadcasting. Key mechanisms:
        - **Decision Broadcast Protocol**

```python
# p2p.py - Broadcast to all peers
def send_data(sock, message):
    data = json.dumps(message).encode()
    sock.sendall(len(data).to_bytes(4, 'big') + data)

# When player makes choice:
for conn in connections:
    send_data(conn, {"decision": choice, "player": username})
```

Each node sends its decision to all others, creating eventual consistency.

        - **State Synchronization**

```python
# game.py - Shared decision registry
class Game:
    def __init__(self):
        self.received_decisions = {}  # Stores others' choices

    def register_decision(self, username, decision):
        self.received_decisions[username] = decision
```

All nodes maintain identical decision maps for scenario resolution.

- Is there any **heart-beating**, **timeout**, **retry mechanism**?
    - The system uses **layered timeouts** with **exponential backoff**:

```python
# p2p.py - Peer connection with retries
def connect_to_peers():
    retries = 0
    while retries < 3:  # 3 attempts
        try:
            sock.connect((peer["ip"], peer["port"]))
            break
        except:
            retries += 1
            time.sleep(2 ** retries)  # Backoff: 2s, 4s, 8s
```

- Decision Synchronization

```python
# p2p.py - Timeout-driven bot activation
def wait_for_decisions(timeout=30):
    start_wait = time.time()
    while time.time() - start_wait < timeout:
        if decisions_received():
            break
    activate_bots()
```

- Network Operations

```python
# p2p.py - Socket-level timeouts
sock.settimeout(10)       # Connect timeout
sock.settimeout(2.0)      # Per-operation timeout
```

- Is there any form of **error handling**?
  - Connections Failure Handling

```python
# p2p.py - Graceful degradation
except (ConnectionResetError, BrokenPipeError):
    print(f"🤖 {player} disconnected")
    with lock:
        bot_players.add(player)
        game.generate_bot_decisions()
```

In the event that a player does not respond within the established time, it is considered that they may have experienced a disconnection or some kind of problem. Therefore, in order not to disrupt the game flow, they are added to the bot list as a player who is temporarily replaced by a bot. If 2 players continue the game, they can proceed without any issues, without the help of the bot. Only in the case that they decide to restart the game to explore more endings without leaving the game, they will be accompanied by 2 bots. If a player is left alone, due to 2 disconnections or network problems with the other 2 players, they will be directly accompanied by the 2 bots.

- Data validation

```python
# p2p.py - Malformed message handling
def recv_data(sock):
    try:
        return json.loads(data)
    except JSONDecodeError:
        print("⚠ Invalid message format")
        return None
```

A message received as erroneous will be ignored.

- Consensus Failure Recovery

```python
# p2p.py - Partial decision resolution
if len(received_decisions) < required:
    if num_humans >= 2:
        wait_for_decisions()  # Retry
    else:
        force_bot_consensus()
```

If the received decisions are not the expected number (3) but there are still 2 players, the game can continue with those 2, since the game is adapted to accommodate 2 or 3 players in principle.

Availability

- **Caching**: No caching is used.
- **Load Balancing:** Not applicable due to the P2P nature of the game.

Security

- **Authentication:** No authentication is required.
- **Authorization:** All players have equal access and control over the game.
- **Cryptography:** The game does not implement encryption.

Implementation

Technological details

- **Network Protocol:** TCP for reliable communication.
- **Data Representation:** JSON for encoding game state and decisions.
- **Frameworks**: Standard Python libraries (socket, threading, json).

- **Threds:** The system is composed of three types of threads:
  - **Connection Acceptance Thread (acceptor_thread)**
    - **Function:** Waits for incoming connections from other players.
    - **Relevant Code:**

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind((host, port))
server.listen(2)  # Queue for 2 pending connections

try:
    while len(connections) < 2 and not exit_flag.is_set():
        conn, addr = server.accept()  # Blocks until connection
        connections.append(conn)  # Thread-safe append
finally:
    server.close()  # Cleanup on exit
```

    - **Behavior:**
      - Runs in the background (daemon=True).
      - If a player connects, it saves their socket in the connections list.
  - **Message Handling Threads (handle_peer_connection)**
    - **Function:** Receives messages (decisions, restart votes) from other players.
    - **Relevant Code:**

```
sock.settimeout(2.0)  # Enables keep-alive check

while not exit_flag.is_set():
    try:
        data = recv_data(sock)  # Non-blocking due to timeout
        if data:
            if "decision" in data:
                with lock:  # Thread-safe state update
                    game.register_decision(data["player"], data["de
cision"])
            elif "retry" in data:
                with lock:
                    peer_retry_data[data["player"]] = data["retry"]
                retry_event.set()  # Signals main thread

    except socket.timeout:
        continue  # Normal timeout, check again
    except (ConnectionResetError, BrokenPipeError):
        break  # Peer disconnected
```

    - **Behavior:**
      - Each connected player has their own reception thread.

        - If a player disconnects, the thread ends and a bot is activated.
    - **Main Thread (main game loop)**
        - **Function:**
            - Displays the scenarios and asks for input from the local player.
            - Processes the decisions and advances the game.
        - **Relevant Code:**

```
    while not exit_flag.is_set():
        # Display current scenario
        scenario = game.get_scenario()
        print(f"\nCurrent scenario: {scenario['scenario']}")

        # Blocking input (could be improved with async I/O)
        choice = int(input("\nYour choice (1/2): "))

        # Thread-safe decision processing
        with lock:
            game.player_decision = choice
            for conn in connections:  # Broadcast to all peers
                send_data(conn, {"decision": choice, "player": usernam
e})

        # Wait for peers with timeout
        wait_for_decisions(game, timeout=30)
```

        - **Behavior:**
            - Is blocking at the input(), but the other threads remain active.
            - If the other players do not respond in 30 seconds, it activates bots.

Validation

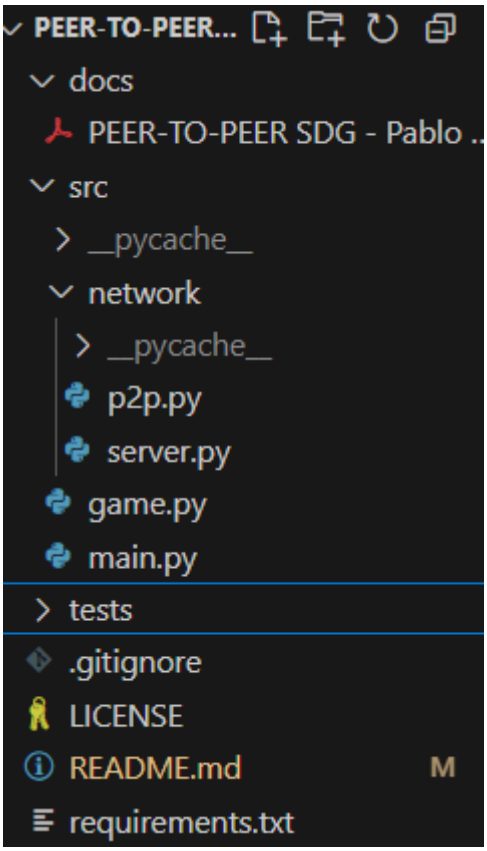
Automatic Testing

- The test that has been performed is a test based on P2P communication. Given the limited knowledge I had before the project, I worked on a test that could prove the P2P connections, and from there, the basic architecture of the project was formed.

Acceptance test

- I have manually tested the game's functionality, as the connection can be verified through tests, but testing the various branches of the game and their respective endings must be done by playing.

Release



- **Versioning**: The game is versioned using Git tags.
- **Distribution**: The game is distributed as a GitHub repository.

Deployment

- Clone the repository: git clone [repository URL].
- Install the dependencies with:
  - pip install -r requirements.txt
- Navigate to the network directory: cd src/network
- Run the server: python server.py.
- Navigate to the src directory in three differents windows: cd src
- Run the game: python main.py.
- Follow the on-screen instructions to connect to a peer and start playing.

- Alternatively, refer to the README document for detailed installation and usage instructions.

User Guide

- how to use your software?
    - All players will see the connection status:

      Welcome to the True Peer-to-Peer Decision-Making Game!
      Enter your username: x
      Your connection details:
      IP Address: x.x.x.x
      Port: x
      Connecting to matchmaking server...

    - Scenario Presentation

      When 3 players are connected to the server the game will display the current scenario:
      ==========
      Your ship has been wrecked by a storm...
      You must act fast: search for food (1) or look for shelter (2).
      ==========

    - Making Decisions

      Each player will be prompted:
        - Your decision (1/2):

      Enter your choice (1 or 2) and press Enter.

    - Waiting for Peer

      After submitting your choice, the game will wait for the other players:
        - Waiting for the other player's decision...
    - Outcome Display

      Once both players have made their choices, the result is shown:

==========
Your choice: 1
Other player's choice: 2
==========

The game will then display the outcome of the combined decisions.

- Progression

  The game will automatically advance to the next scenario based on the players' choices.

Self-evaluation

I worked on this project individually because, as an Erasmus student, I didn't have the opportunity to get to know my classmates well. Additionally, I believe I would have had difficulties coordinating due to the language barrier and scheduling differences. While Italian students in Cesena usually have similar timetables and courses, I had subjects from both the master's and bachelor's programs, which made collaboration even more challenging.

The biggest challenge was understanding how P2P architecture works, as well as implementing threads to ensure its proper functionality. This caused me many issues since I often encountered errors that were either unfamiliar to me or difficult to solve. To stay organized, I tried to document my progress using *issues*, but structuring everything on my own was still difficult, especially at the beginning, when I felt lost and overwhelmed by the subject matter. Initially, I built a client-server communication model to help myself understand the system step by step. Once I felt more confident, I gradually transformed it into a P2P architecture. The game initially supported a 2-player setup, but it was updated due to increased requirements, adding support for 3 players. This caused me problems managing synchronizations, sending and receiving data from the 3 players, as it scaled the difficulty that could exist between 2. The main problem here was the correct synchronization in normal situations, so that the game could be repeated infinitely and the players would not have problems with this. I also added a central server, whose only function was to facilitate the players' entry into the game, without having to know their different IPs, only having to execute the code and from then on continue normally with peer-to-peer. Finally, once all the code was working as I

wanted, I implemented fault tolerance, using timeouts and bots, so that players would not see their gaming experience ruined by a disconnection or error of any kind.

This project has helped me improve my ability to organize and document software development more professionally, learning how to structure a project using *README*, *issues*, and *branches* in GitHub. It has also given me a deeper understanding of how distributed systems work internally and the importance of proper code management.

Additionally, as I progressed, I got excited about making the game more diverse, which led me to implement more possible endings than I had originally planned.

I hope you enjoy playing the game and that you can see the effort I put into this project.