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, change the messages that will
appear on screen 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 two 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 directly without a central server. Players make decisions that affect the outcome of the story, with multiple branching paths and endings. The game emphasizes collaboration, decision-making, and synchronization between players in real-time. Based on this, we will learn how to manage a distributed system, how to implement P2P, and how to make learning more enjoyable and fun through a game.

Concept

This project is a Peer-to-Peer multiplayer game where two players must collaborate in a survival scenario. The game uses a command-line interface (CLI), where players connect directly to each other without relying on a centralized server.

- Use case collection
    - **Where**: The players are at different locations, possibly across the internet.

- **When and How Often**: Players interact when the game prompts 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, with the game responding to the decisions made by each player.
- **Devices**: The game is designed for desktop systems running Python.
- **Data Storage:** The system does not store user data persistently. All game state is maintained in memory during the session.
- **Roles**: There are two roles: Player 1 and Player 2. Both players have equal responsibilities and decision-making power.

Requirements

- Functional Requirements
  - Peer-to-Peer Connection: The game must allow two players to connect directly over a network using IP addresses and ports.
  - Real-Time Decision Synchronization: Players must be able to make decisions simultaneously, and the game must synchronize these decisions in real-time.
  - 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 game follows a peer-to-peer architectural style. Each player runs an instance of the game, and the two instances communicate directly over a TCP connection. This architecture was chosen for its simplicity and lack of reliance on a central server.

Infrastructure

- ***Components****: Two game instances (one per player) connected over a TCP socket.*
- **Network Distribution:** The game instances can run on the same machine (localhost) or different machines on the same network.
- **Component Discovery:** Players manually enter the IP address and port of the peer they wish to connect to.

Modelling

- ***Domain Entities:***
  - Game: Manages the current scenario, decisions, and game state.
  - Player: Represents a user making decisions.
  - Scenario: A specific situation with a description and choices.
- **Domain Events:**
  - Player makes a decision.
  - Game state is updated.
  - Game ends or restarts.
- **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, sending JSON-encoded messages.
- **Interaction Patterns**: The game uses a request-response pattern for decision synchronization and state updates.

Behaviour

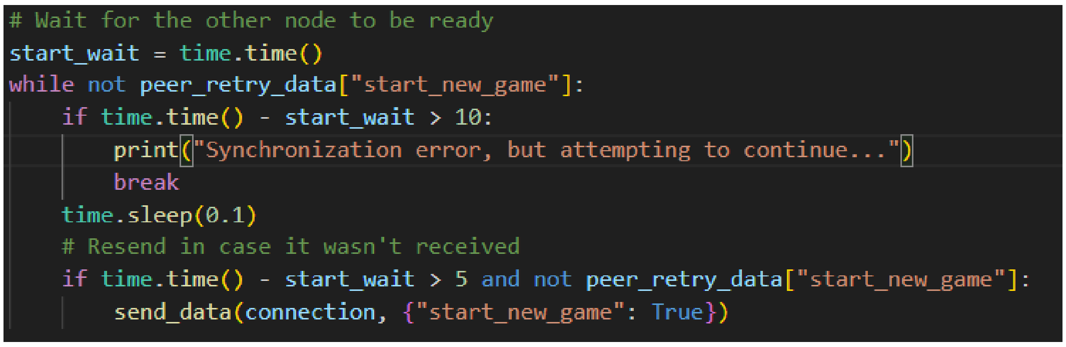
- **Game Instance:**
    - Listens for incoming connections.
    - Synchronizes decisions with the peer.
    - Updates the game state based on decisions.
- **Player:**
    - Inputs decisions (1 or 2).
    - Waits for the peer's decision.

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 way data is shared in the code is by sharing the decisions of both players, so that each node can continue the game normally based on the other's decisions.
- Is there any **heart-beating**, **timeout**, **retry mechanism**?
    - There is a timeout every time a response is expected from the other node to prevent deadlocks, and there are 2 retries:



```
# Wait for the other node to be ready
start_wait = time.time()
while not peer_retry_data["start_new_game"]:
    if time.time() - start_wait > 10:
        print("Synchronization error, but attempting to continue...")
        break
    time.sleep(0.1)
    # Resend in case it wasn't received
    if time.time() - start_wait > 5 and not peer_retry_data["start_new_game"]:
        send_data(connection, {"start_new_game": True})
```

- Is there any form of **error handling**?
    - When synchronization or connection fails, the system is notified, either at the time of connection or by timeout if a failure occurs on the other node. what happens when a component fails? why? how?

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:** Both players have equal access and control.
- **Cryptography:** No encryption is used.

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).

Validation

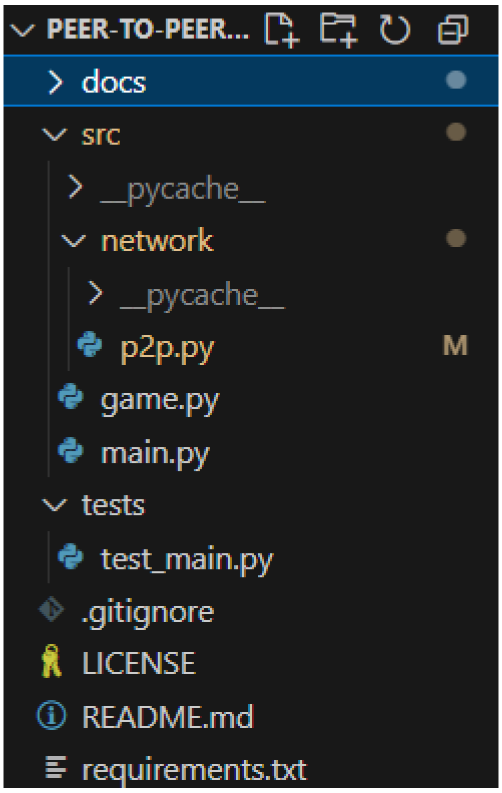
Automatic Testing

- 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

- 

```
PEER-TO-PEER...
> docs
v src
  > __pycache__
  v network
    > __pycache__
    p2p.py                M
  game.py
  main.py
v tests
  test_main.py
.gitignore
LICENSE
README.md
requirements.txt
```

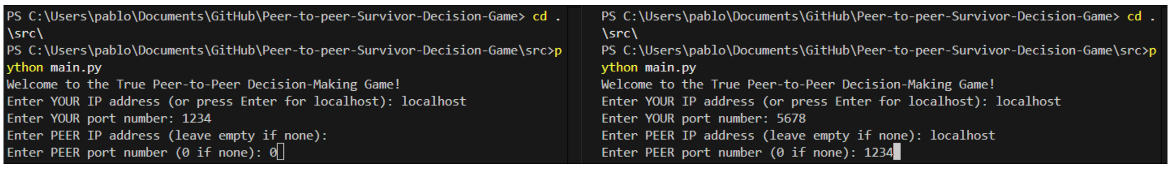
- **Versioning**: The game is versioned using Git tags.
- **Distribution**: The game is distributed as a GitHub repository.

Deployment

- Clone the repository: git clone [repository URL].
- Install the dependencies with:
  - pip install -r requirements.txt
- Navigate to the src directory: cd src.
- Run the game: python main.py.
- Follow the on-screen instructions to connect to a peer and start playing.
- Or you can read the README document for follow the steps.

User Guide

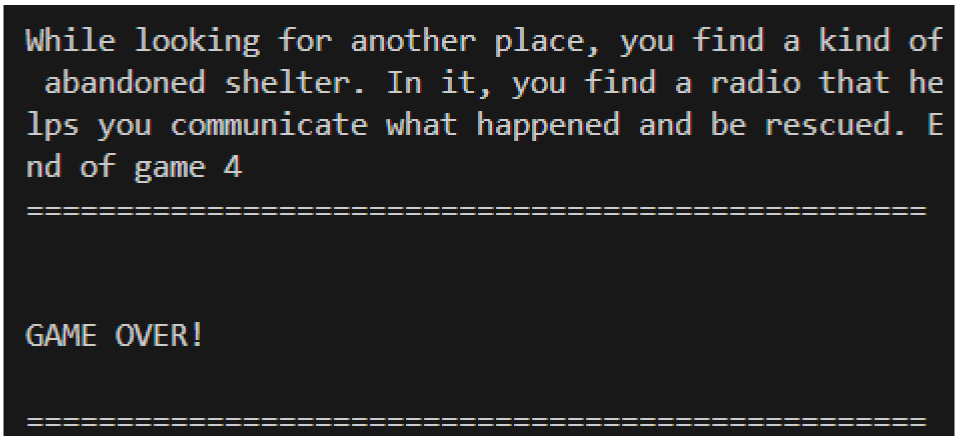
- how to use your software?
  - Game-Flow



```
PS C:\Users\pablo\Documents\GitHub\Peer-to-peer-Survivor-Decision-Game> cd .
\src\
PS C:\Users\pablo\Documents\GitHub\Peer-to-peer-Survivor-Decision-Game\src>p
ython main.py
Welcome to the True Peer-to-Peer Decision-Making Game!
Enter YOUR IP address (or press Enter for localhost): localhost
Enter YOUR port number: 1234
Enter PEER IP address (leave empty if none):
Enter PEER port number (0 if none): 0
```

```
PS C:\Users\pablo\Documents\GitHub\Peer-to-peer-Survivor-Decision-Game> cd .
\src\
PS C:\Users\pablo\Documents\GitHub\Peer-to-peer-Survivor-Decision-Game\src>p
ython main.py
Welcome to the True Peer-to-Peer Decision-Making Game!
Enter YOUR IP address (or press Enter for localhost): localhost
Enter YOUR port number: 5678
Enter PEER IP address (leave empty if none): localhost
Enter PEER port number (0 if none): 1234
```

Possible Final:



```
While looking for another place, you find a kind of
 abandoned shelter. In it, you find a radio that he
lps you communicate what happened and be rescued. E
nd of game 4
==================================================



GAME OVER!


==================================================
```

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.

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.