GROUP 5

HOME SERVICE ABOUT US CONTACT US

NETWORK PROGRAMMING

TOPIC : B3

NETWORKED WORD CHAIN GAME

22BA13102	Nguyễn Tiến Duy
23BI14030	Trần Thục Anh
23BI14032	Nguyễn Thị Vàng Anh
22BA13020	Nguyễn Phương Anh
22BA13001	Bùi Trường An
23BI14356	Lương Quỳnh Nhi
22BA13032	Trần Thượng Nam Anh

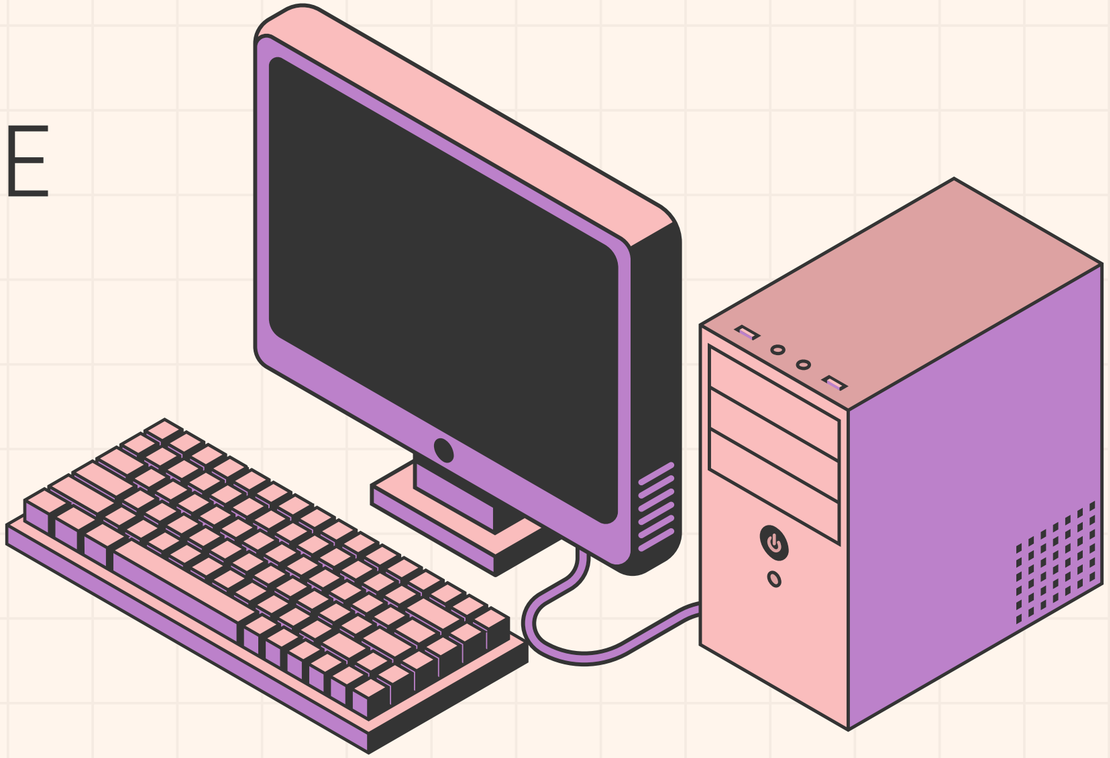
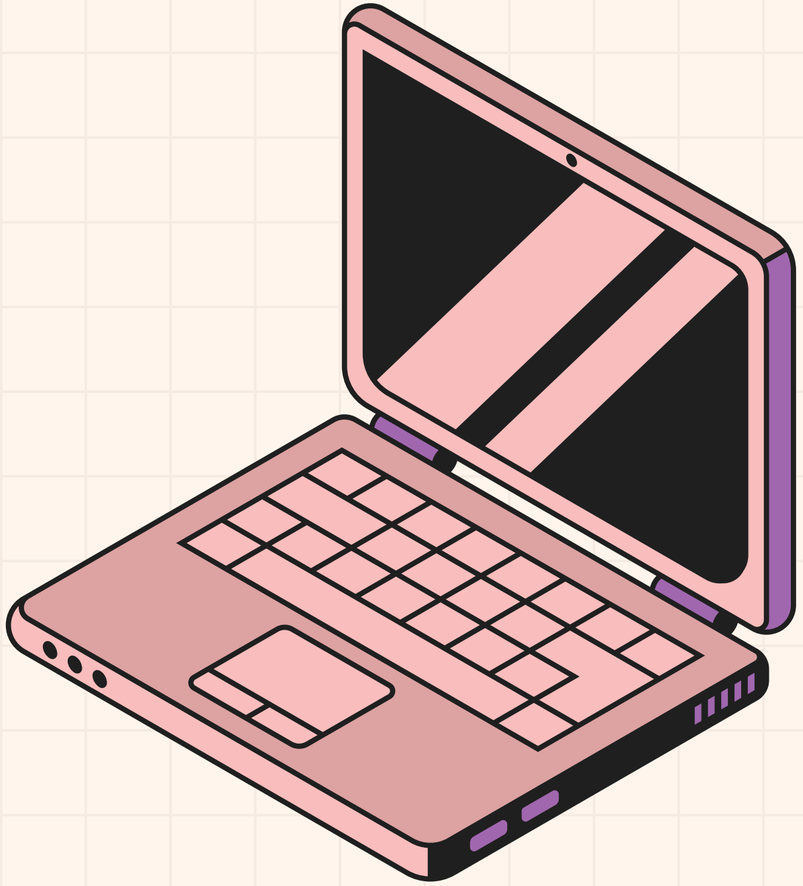


TABLE OF CONTENTS



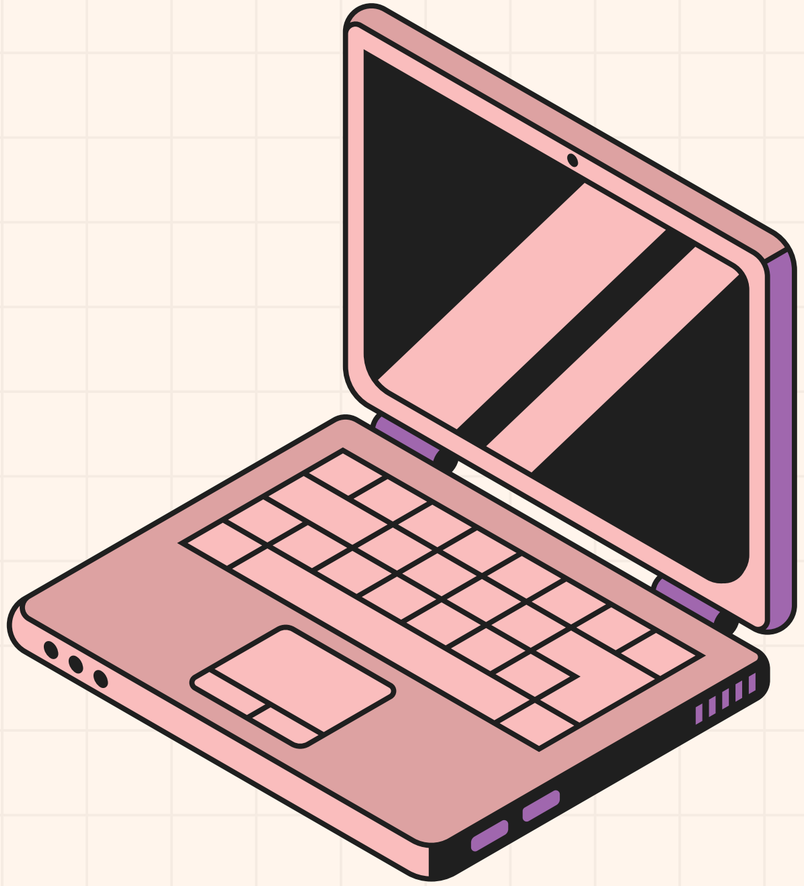
REQUIREMENT

Window 11, Unix system

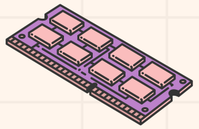
Python 3.13

Library:

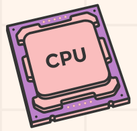
- socket
- threading
- queue
- json
- random
- string
- datetime
- tk



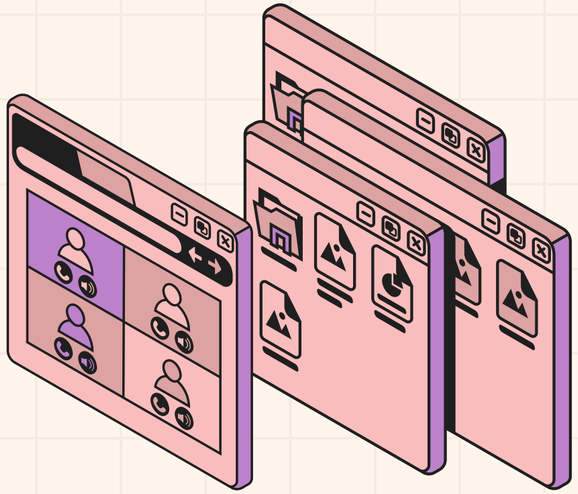
GAME OVERVIEW



- Multiplayer word chain game for **2-5 players** per session
- Players provide word **starting with the last letter** of previous word
- **Dictionary** validation ensures legitimate words
- **30-second turn** timer with timeout penalties



HOW TO PLAY ?

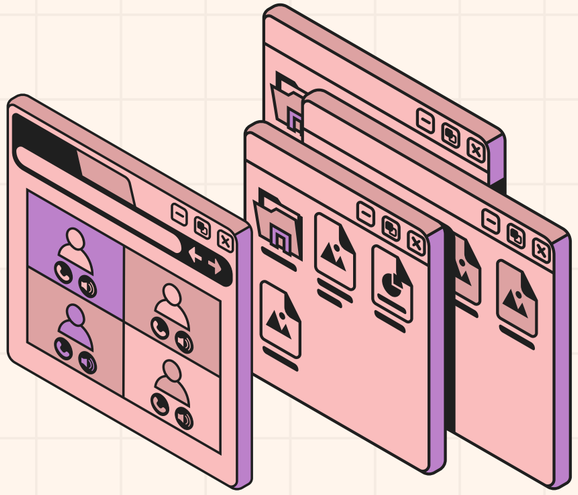


The **first** player to join becomes the **host** and **starts** the game.

🧠 Word Rules:

- The word must start with the last letter of the previous one.
- The word must be **valid** (checked with the dictionary).
- **No repeated words** are allowed during the game.

HOW TO PLAY ?



🎯 Goal:

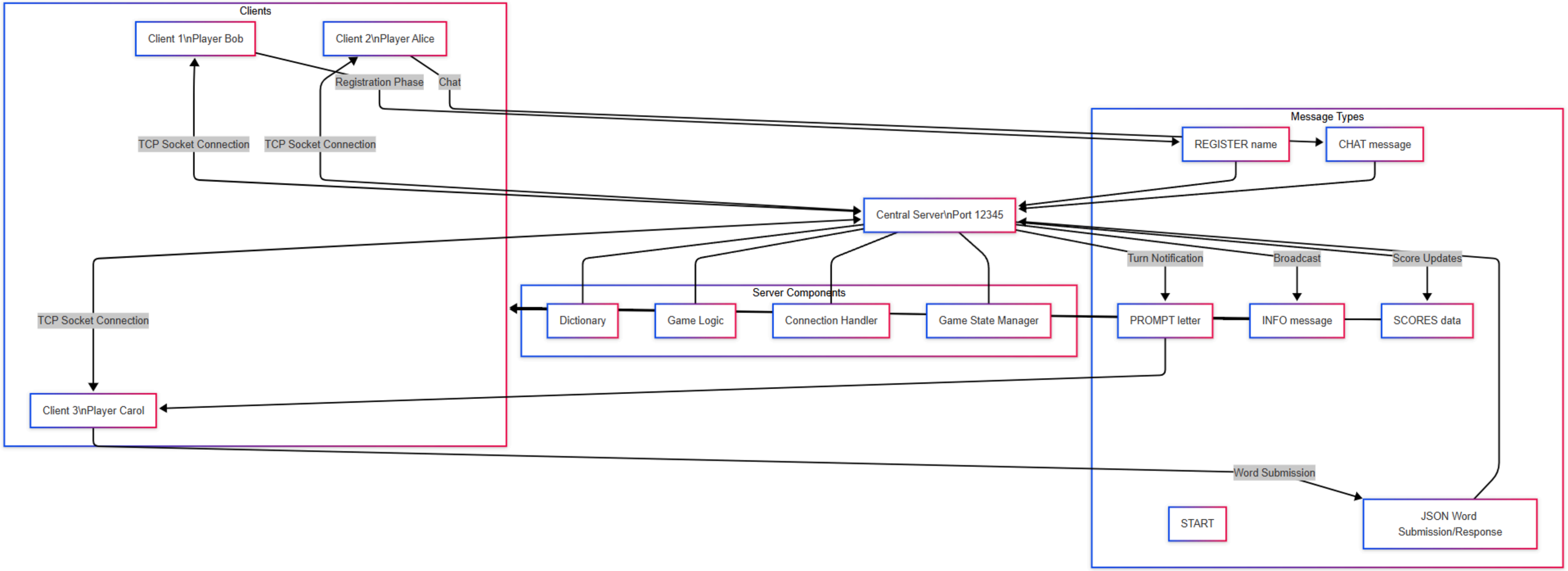
The first player to reach **50 points** wins the game.

🏆 Scoring System:

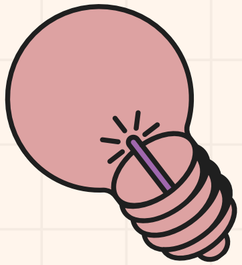
- ✅ Correct word → **+1 point** per character
  Example: elephant → **+8 points**
- ⚡ Fast answer (<5s) → **+2 bonus points**
- ❌ Invalid word → **-1 point**
- ⏰ Timeout (no answer) → **-2 points**

DESIGN

Network Achitechture



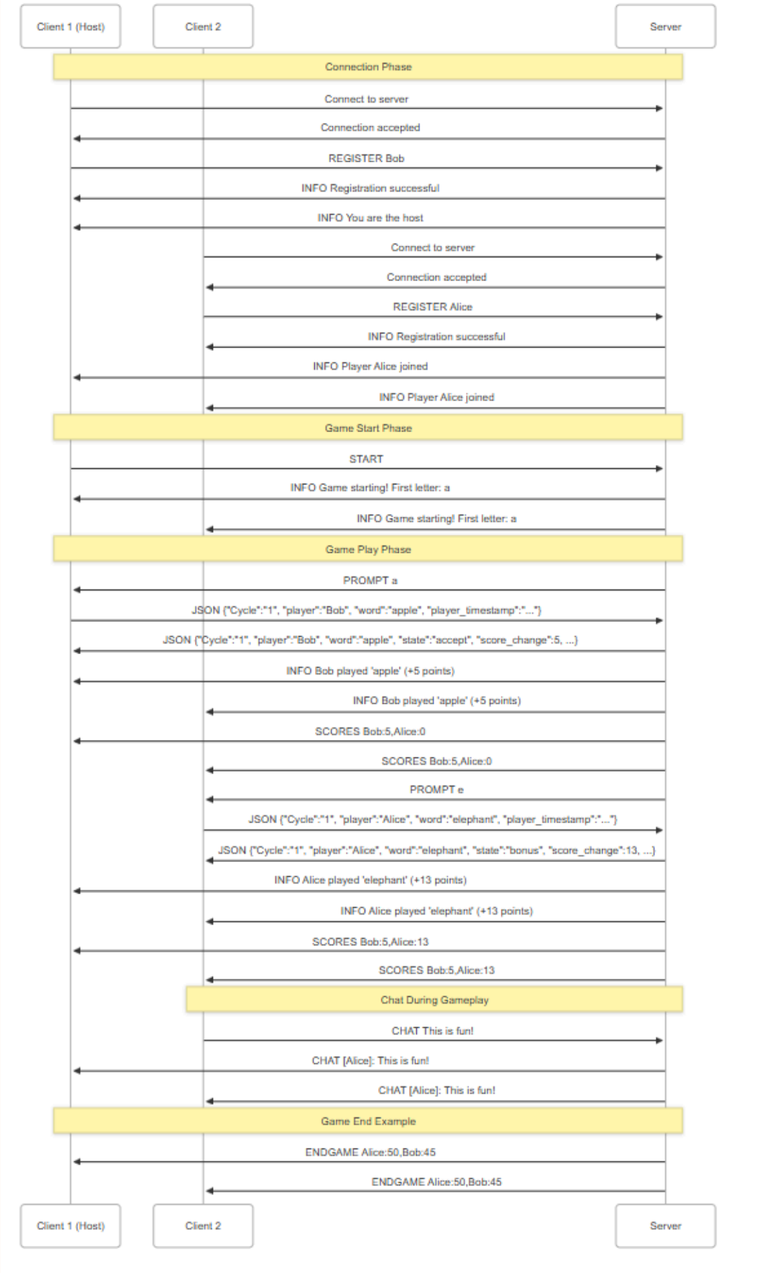
DESIGN



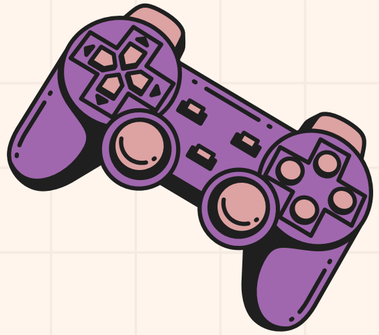
Network Protocol

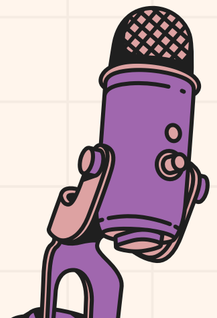
The communication is based on **TCP sockets** using **plain text commands** and **JSON** for structured data.

Network Protocol diagram



IMPLEMENTATION





Connection Establishment

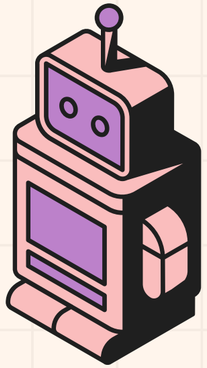
Server Setup

- Creates **TCP** socket bound to port 12345
- Listens for up to 5 concurrent connections
- Uses **accept_loop()** to constantly listen for new clients

```
def accept_loop():
    load_dictionary()
    srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    srv.bind(('', PORT))
    srv.listen(MAX_PLAYERS)
    print(f"Server listening on port {PORT}...")
```

Connection Establishment

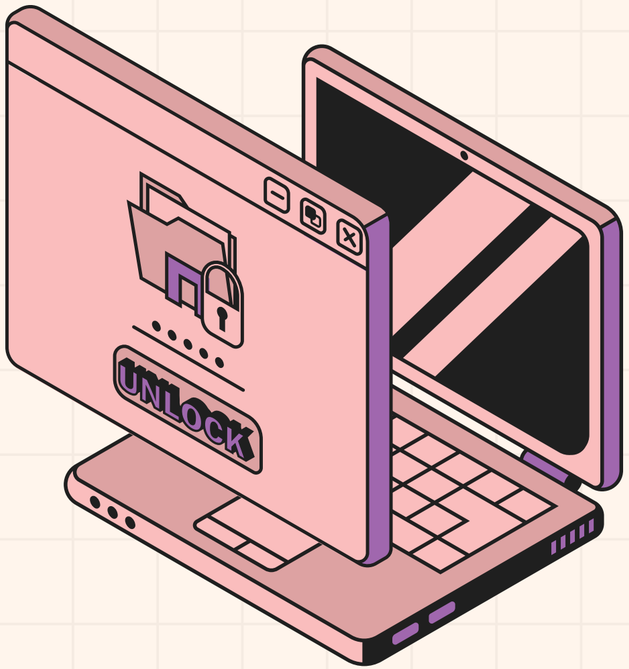
Client Connection



- Makes up to 3 connection attempts
- Each attempt has a 5-second timeout
- Provides error handling for failed connections

```python
def connect_to_server(self):
    for attempt in range(RETRY_ATTEMPTS):
        try:
            self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.client_socket.settimeout(5)
            self.client_socket.connect((HOST, PORT))
            self.client_socket.settimeout(None)
            self.add_message_to_chat("System", "Connected to server")
            return
        except Exception as e:
            self.add_message_to_chat("System", f"Connection attempt {attempt + 1} failed: {e}")
            if attempt < RETRY_ATTEMPTS - 1:
                time.sleep(RETRY_DELAY)
    messagebox.showerror("Error", "Failed to connect to server after multiple attempts")
    self.root.quit()
```

Message Protocol



Simple Text Commands

- Registration: **REGISTER** [name]
- Game Control: **START**, **PROMPT** [letter]
- Chat: **CHAT** [message]
- Information: **INFO** [message], **SCORES** player1:score1,...

Message Protocol

JSON Messages

- Used for word submissions and game state updates

```
{
  "Cycle": "1",
  "player": "2k18",
  "word": "paleolithic",
  "player_timestamp": "2025-05-21T09:45:32.594Z",
  "server_timestamp": "2025-05-21T09:45:33.002Z"
}
```

{"Cycle": "1", "player": "2k18", "word": "paleolithic", "server_timestamp": "2025-05-21T09:45:33.002Z", "state": "accept", "score_change": 11, "current_score": 11}

Message Handling

Server-Side Processing
- Receiving Messages

```
data = sock.recv(1024)
for line in data.decode().splitlines():
    if line.startswith("REGISTER "):
        # Registration handling
    elif line.strip() == "START":
        # Game start handling
    elif line.startswith("CHAT "):
        # Chat handling
    elif line.startswith("{"):
        # JSON word submission handling
```

- Server reads up to **1024** bytes at a time
- Processes each complete line of text separately
- Handles different message types based on prefixes or JSON format

Message Handling

```python
def broadcast(msg):
    data = (msg + "\n").encode()
    with lock:
        for p in players:
            try:
                p['socket'].sendall(data)
            except:
                pass
```

Server-Side Processing
- Sending Messages

- **sendall**() ensures the complete message is sent
- Messages are encoded to bytes before sending
- Broadcast function sends messages to all connected players
- Error handling catches disconnected clients

Message Handling

```python
def receive_messages(self):
    buffer = b""
    while True:
        try:
            chunk = self.client_socket.recv(BUFFER_SIZE)
            if not chunk:
                raise ConnectionError("Server disconnected")
            buffer += chunk

            while b'\n' in buffer:
                message, buffer = buffer.split(b'\n', 1)
                message = message.decode('utf-8', errors='ignore').strip()
                # Process message based on type
        except ConnectionError as e:
            # Handle connection errors
```

Client-Side Processing – Receiving Messages

- The client uses a separate thread for receiving messages
- A buffer accumulates incoming data
- Messages are **processed** when complete
- Different message types trigger different UI updates
- Partial messages remain in the buffer until complete

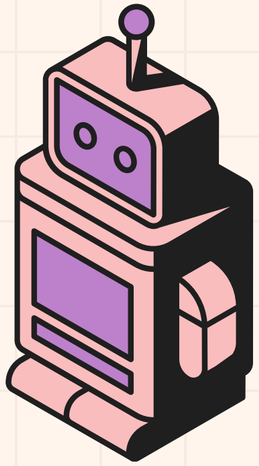
Message Handling

Client-Side Processing - Sending Messages

```python
def send_word(self):
    word = self.word_entry.get().strip()
    ts = datetime.datetime.utcnow().isoformat(timespec='milliseconds') + "Z"
    payload = {
        "Cycle": str(self.current_cycle),
        "player": self.name,
        "word": word,
        "player_timestamp": ts
    }
    try:
        self.client_socket.send((json.dumps(payload) + "\n").encode('utf-8'))
    except (ConnectionError, BrokenPipeError) as e:
        # Handle errors
```

- **JSON payloads** are serialized, encoded, and sent with newline
- **Timestamps** are added before sending
- Error handling catches connection problems

Synchronization Mechanisms



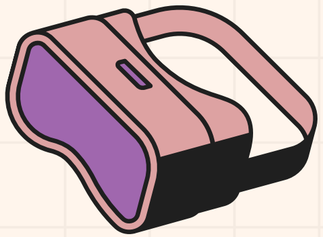
Message Queuing

- Each player has their own queue
- Allows server to wait for responses with **timeout**

```
q = queue.Queue()
player_queues[idx] = q


msg = player_queues[turn].get(timeout=TIMEOUT_SEC)
```

Synchronization Mechanisms

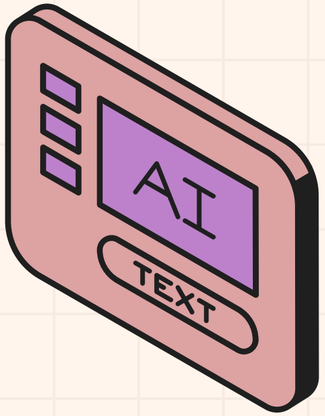


Threading and Locks

- Prevents race conditions with multiple client threads
- Protects shared data access

```
with lock:
    for p in players:
        try:
            p['socket'].sendall(data)
        except:
            pass
```

Synchronization Mechanisms



Game State Synchronization

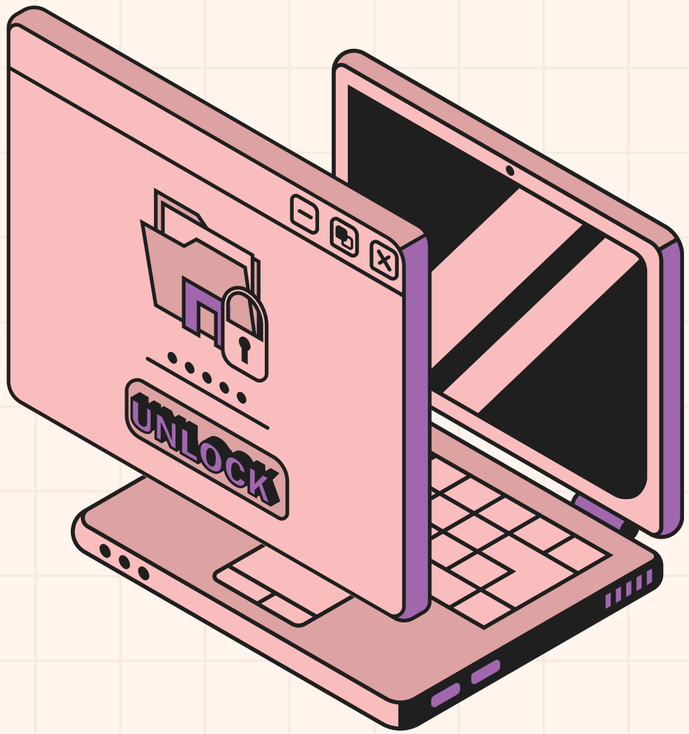
- Server broadcasts score updates to all players
- Ensures consistent game state across clients

```
def broadcast_scores():
    with lock:
        parts = [f"{p['name']}:{p['score']}" for p in players]
    broadcast("SCORES " + ",".join(parts))
```

Error Handling

Connection Failures

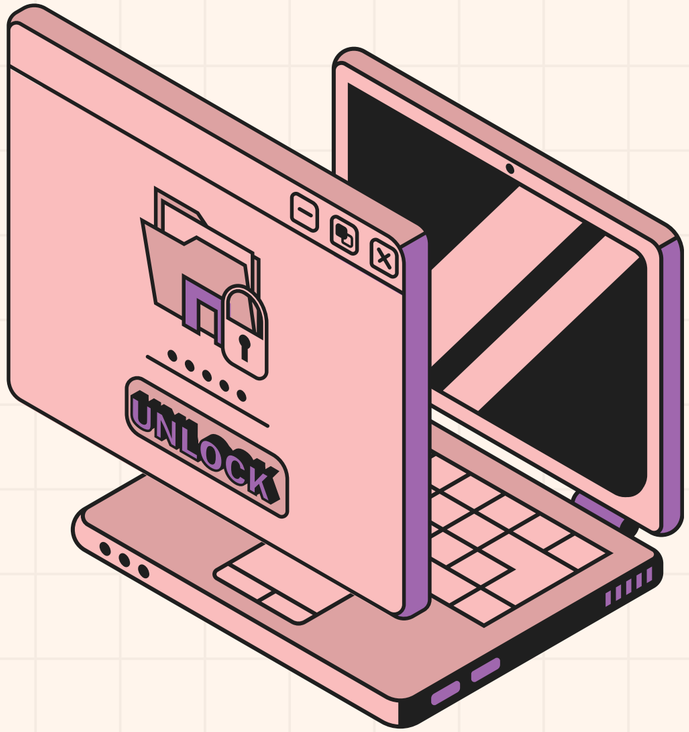
- Client reconnection attempts
- Graceful handling of disconnections
- User notifications



Error Handling

Message Parsing

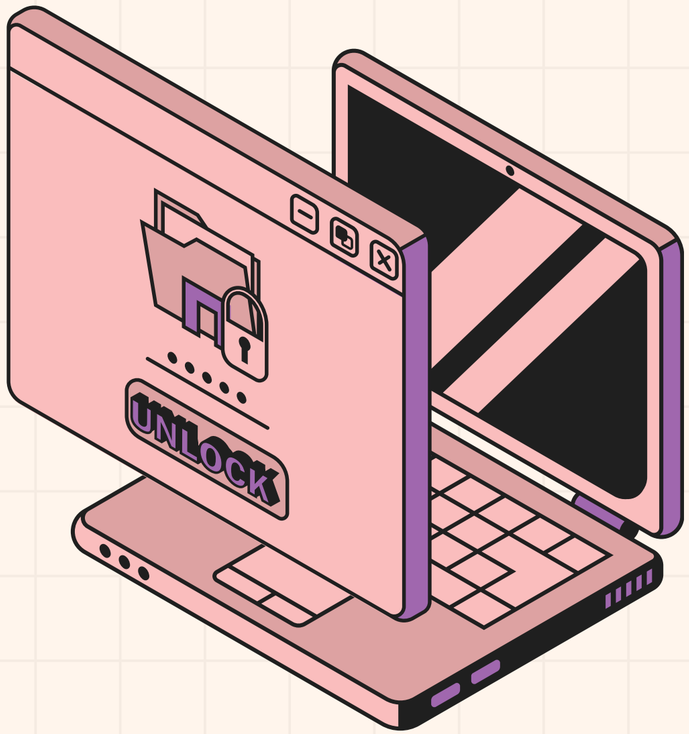
- Try/except blocks for JSON parsing
- Server validation before processing



Error Handling

Network Timeouts

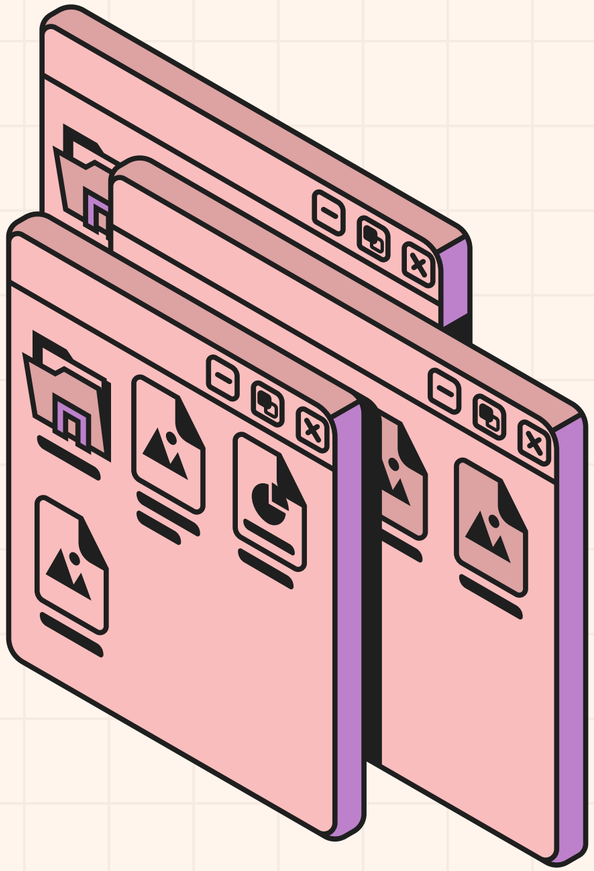
- 30-second game turn timeout
- 5-second connection timeouts



Performance Considerations

Timestamping

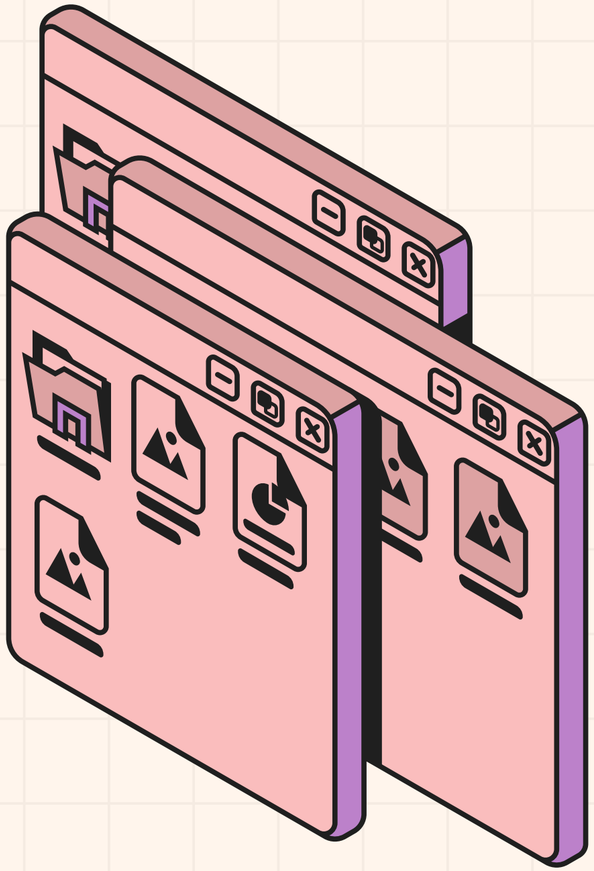
- Both client and server add timestamps
- Enables latency analysis and fair gameplay



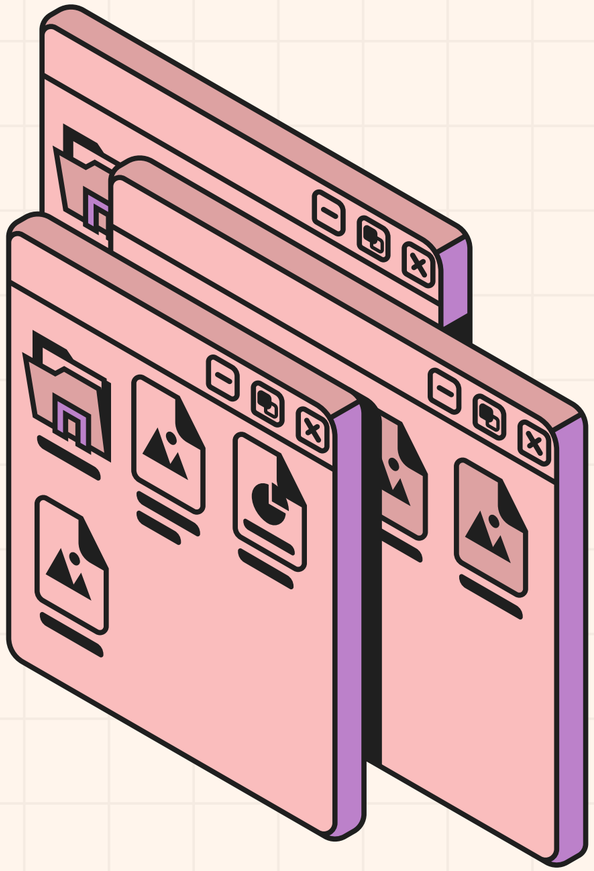
Performance Considerations

Buffered Reading

- Handles partial messages effectively
- Ensures complete message processing



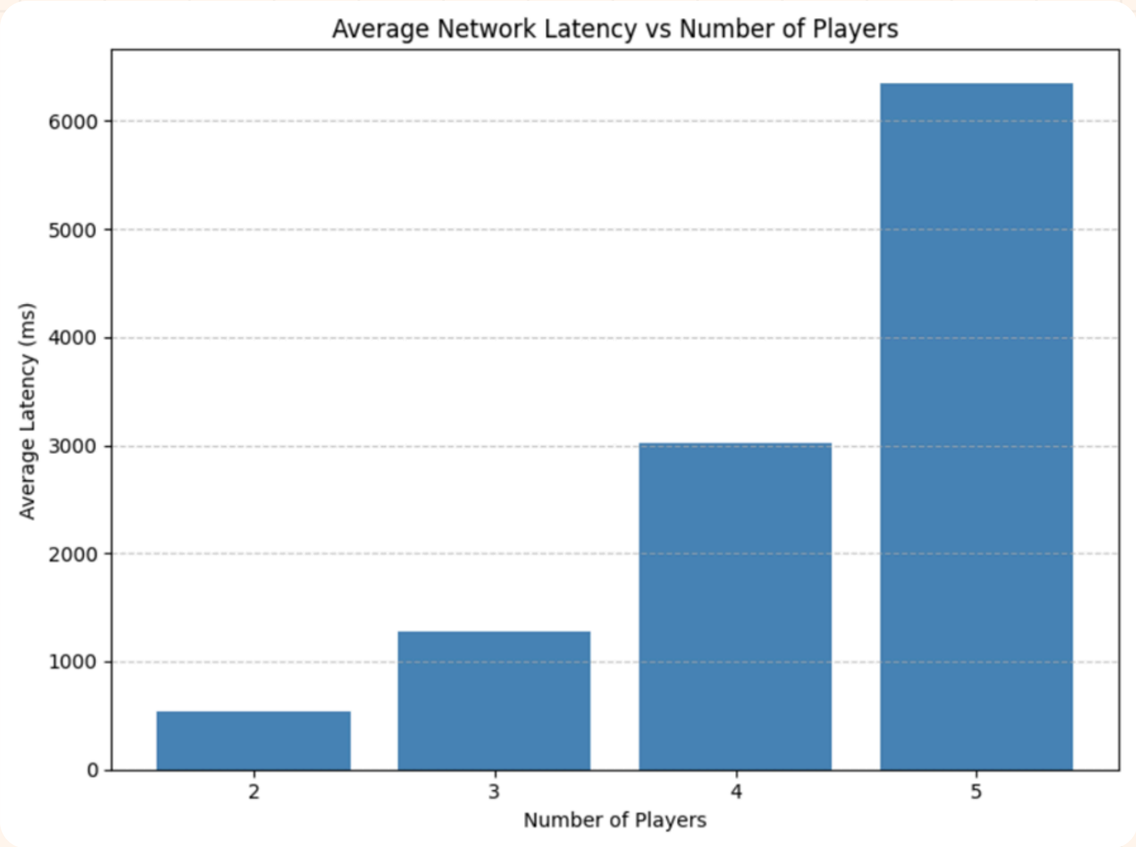
Performance Considerations

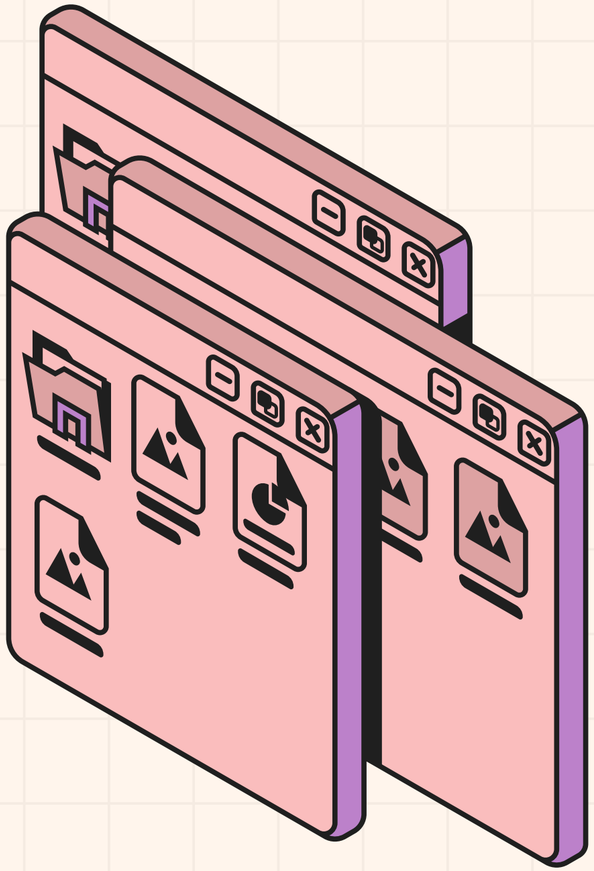


Thread Safety

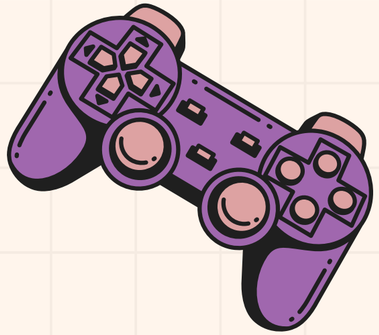
- Locks protect shared resources
- UI updates on main thread to avoid race conditions

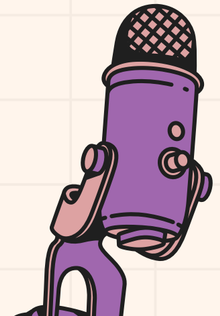
Network latency (suppose that no one timeout in their turn)



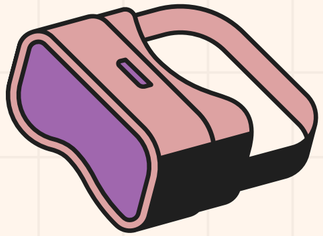


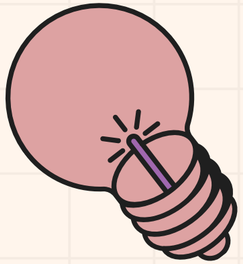
GAMEPLAY





GROUP 5





THANK YOU

QUESTION US IF WONDERING