Protocolo da Camada de Aplicação: Jogo Multiplayer "Pedra, Papel e Tesoura"

Grupo: 14
Membros: Thiago Piccoli, Gabriel Weiand

1. Objetivo/Função da Aplicação e Protocolo Desenvolvido

O objetivo desta aplicação é implementar um jogo multiplayer de "Pedra, Papel e Tesoura" em uma arquitetura cliente-servidor. O protocolo de aplicação desenvolvido foi projetado para gerenciar todas as interações necessárias para a execução do jogo. Ele define como um cliente se registra no servidor, entra em uma fila de pareamento, participa de uma partida trocando jogadas com um oponente, e como os resultados são computados e comunicados, além de manter um ranking global de vitórias.

2. Características do Protocolo Desenvolvido

- **Arquitetura: Cliente-Servidor**
    - **Justificativa:** Foi escolhida uma arquitetura centralizada onde um servidor autoritativo gerencia a lógica do jogo, o estado das partidas e o pareamento de jogadores. Os clientes são interfaces leves responsáveis apenas por enviar as ações do usuário e renderizar o estado recebido do servidor. Essa abordagem simplifica o desenvolvimento do cliente e previne trapaças, já que toda a lógica de validação reside no servidor.
- **Modelo de Conexão: Com Estado (Stateful)**
    - **Justificativa:** O protocolo é inerentemente com estado, pois o servidor precisa manter informações cruciais sobre a sessão de cada jogador, como seu nome, se está em uma partida ou na fila de espera, e o estado da partida atual (rodada, pontuação). Além disso, o estado do ranking global é mantido durante toda a execução do servidor.
- **Persistência: Persistente**
    - **Justificativa:** A conexão TCP estabelecida pelo cliente é mantida ativa durante toda a sessão do jogo (da conexão inicial até o fim da partida). Um jogo com múltiplas rodadas e interações rápidas se beneficia de uma conexão persistente, evitando a sobrecarga de estabelecer uma nova conexão TCP para cada jogada enviada.

- **Modo de Comunicação: Híbrido (Push/Pull)**
    - **Justificativa:** O protocolo utiliza ambos os modelos para uma comunicação eficiente.
        - **Pull:** O cliente inicia a comunicação para realizar ações deliberadas, como enviar uma jogada (ROC, PAP, SCI) ou solicitar o ranking (RAN).
        - **Push:** O servidor inicia a comunicação para notificar o cliente sobre eventos assíncronos que ele não solicitou, como o início de uma partida (MAT) ou a solicitação de uma jogada (PLA).
- **Controle: Na Banda (In-band)**
    - **Justificativa:** Todas as mensagens, tanto de dados (as jogadas) quanto de controle (início de partida, solicitação de jogada), trafegam pelo mesmo socket TCP. Não há um canal de comunicação separado para o controle do protocolo, o que simplifica a implementação da rede.

3. Listagem dos Tipos de Mensagens (Cliente → Servidor)

Tipo de Mensagem	Descrição da Função
CON	**Conectar:** Enviada pelo cliente ao iniciar a conexão para se registrar no servidor com um nome de jogador.
ROC / PAP / SCI	**Jogada:** Enviada para submeter a jogada do jogador (Pedra, Papel ou Tesoura) durante uma rodada.
RAN	**Ranking:** Solicita ao servidor o ranking atual de vitórias.
QUI	**Sair (Quit):** Informa ao servidor que o cliente deseja se desconectar de forma voluntária.

4. Listagem e Descrição dos Tipos de Mensagens (Servidor → Cliente)

Tipo de Mensagem	Descrição da Função
MAT	**Partida (Match):** Notifica o cliente que um oponente foi encontrado e a partida irá começar.
PLA	**Jogar (Play):** Sinaliza o início de uma nova rodada e solicita que o cliente faça sua jogada.
WIN / LOS / TIE	**Resultado da Rodada:** Informa o resultado da rodada (Vitória, Derrota ou Empate) e qual foi a jogada do oponente.
RAN	**Ranking:** Enviada em resposta a uma solicitação do cliente, contendo a lista de

	jogadores e suas vitórias.
END	**Fim (End):** Sinaliza o fim da partida ou o desligamento do servidor, contendo uma mensagem final.

5. Formato de Cada Tipo de Mensagem e Seus Respectivos Campos

O protocolo utiliza o formato
JSON para todas as mensagens, o que oferece estrutura e legibilidade. Cada mensagem é um objeto JSON enviado como uma única linha, delimitada por um caractere de nova linha (\n).

Formato Geral:

```
JSON:
{
  "type": "NOME_DO_COMANDO",
  "payload": { ... }
}
```

- type: Campo obrigatório que funciona como o cabeçalho principal, definindo a intenção da mensagem.
- payload: Um objeto JSON contendo os dados específicos associados ao comando.

Exemplos de Mensagens:

- **Cliente se conectando (CON):**

```
JSON
{
  "type": "CON",
  "payload": {
    "nome": "Goku"
  }
}
```

- **Servidor iniciando uma partida (MAT):**

```
JSON
{
  "type": "MAT",
  "payload": {
    "oponente": "Vegeta"
  }
}
```

- **Servidor informando vitória na rodada (WIN):**
  JSON
  ```
  {
    "type": "WIN",
    "payload": {
      "jogada_oponente": "sci"
    }
  }
  ```

6. Especificação dos Valores Possíveis dos Campos

Campo	Subcampo (em payload)	Tipo	Valores Possíveis
type	-	String	CON, MAT, PLA, ROC, PAP, SCI, WIN, LOS, TIE, RAN, QUI, END
payload	nome	String	Qualquer nome de jogador não vazio. Ex: "Goku".
	oponente	String	O nome do jogador oponente. Ex: "Vegeta".
	jogada_oponente	String	A jogada feita pelo oponente. Ex: "roc", "pap", "sci", "timeout".
	ranking	Array de Objetos	Lista de jogadores e vitórias. Ex: [{"nome": "Goku", "vitorias": 5}, {"nome": "Vegeta", "vitorias": 3}]
	mensagem	String	Texto informativo para o comando END. Ex: "O vencedor da partida foi Goku!".

7. Outras Informações Relevantes

- **Casos de Uso Típicos:**

1. O jogador inicia o cliente, se conecta ao servidor e se registra com um nome (CON).
2. O servidor coloca o jogador em uma fila de espera.
3. Quando um segundo jogador se conecta, o servidor os emparelha e envia uma notificação MAT para ambos.
4. O servidor gerencia a partida em um sistema de melhor de três rodadas, usando PLA para solicitar jogadas e WIN/LOS/TIE para comunicar os resultados.
5. Ao final das três rodadas, o servidor determina o vencedor geral da partida e envia END para ambos os clientes, que encerram a conexão.
6. A qualquer momento, um cliente pode enviar RAN para visualizar o ranking.

- **Protocolos e APIs Utilizados:**
  - **TCP:** O protocolo da camada de transporte, escolhido por sua confiabilidade e entrega ordenada de pacotes.
  - **Python socket:** API utilizada para a programação de rede de baixo nível, permitindo a criação e gerenciamento de sockets TCP.
  - **Python json:** API utilizada para a serialização (codificação) e desserialização (decodificação) das mensagens trocadas entre cliente e servidor.
  - **Python threading:** API utilizada para gerenciar a concorrência, permitindo que o servidor atenda a múltiplos clientes e que o cliente execute I/O de rede e de usuário simultaneamente.
- **Técnica de Segurança e Criptografia:**
  - O protocolo em sua versão atual **não implementa** nenhuma camada de segurança ou criptografia. Os dados são trafegados em texto plano. Para uma aplicação em produção, seria essencial implementar TLS (Transport Layer Security) sobre a conexão TCP para garantir a confidencialidade e integridade dos dados.
- **Dificuldades Encontradas (Potenciais):**
  - **Gerenciamento de Estado Concorrente:** A principal dificuldade foi garantir o acesso seguro às estruturas de dados globais (como jogadores_em_espera e ranking) por múltiplas threads de clientes. Isso foi solucionado com o uso de um
    threading.Lock() para serializar o acesso a essas seções críticas do código, prevenindo condições de corrida [9].

  - **Tratamento de Desconexões:** Lidar com a desconexão abrupta de um cliente no meio de uma partida é um desafio. O protocolo atual remove o jogador das listas ativas, e a lógica de jogo trata a ausência de jogada como TIMEOUT, permitindo que a partida continue e termine para o jogador restante.