




EP2 - MAC0352

Jogo da Velha

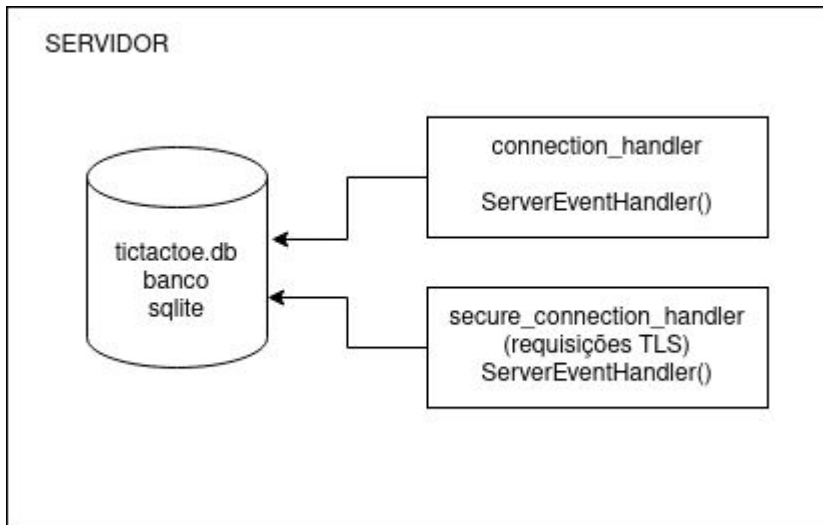
Online

Washington Luiz Meireles de Lima - 10737157

Ygor Tavela Alves - 10687642

- 
- **Visão Geral Arquitetura - Servidor**
 - **Detalhes de Implementação - Servidor**
 - **Visão Geral Arquitetura - Cliente**
 - **Detalhes de Implementação - Cliente**
 - **Desempenho**

Visão Geral Arquitetura - Servidor



- Servidor se baseia na execução de duas instâncias de um *ServerEventHandler*, um para requisições padrões TCP e outro para conexões que exigem criptografia através do TLS
- Há uma instância de um banco de dados SQLite para armazenar os dados referentes aos usuários e logs.



Visão Geral Arquitetura - Servidor

Tabelas - banco de dados servidor

users	
PK	<u>id</u>
	username
	password
	win_count
	lose_count
	tie_count

logs	
PK	<u>id</u>
	created_at
	type
	log



Detalhes de implementação - Servidor

Server:

- Executa duas instâncias da classe *ServerEventHandler* em threads, uma para tratar conexões padrões TCP e outra para tratar conexões criptografadas através de TLS.
- Inscreve as instâncias em eventos, efetuando ações correspondentes aos eventos que chegam nas requisições e retornando uma resposta para a mesma. As requisições e as respostas possuem um corpo padrão JSON:

```
{  
    packet_type: request | response,  
    packet_name: <event_name>,  
    ...  
    <body_data1> : <body_value1>,  
    ...  
}
```



Eventos:

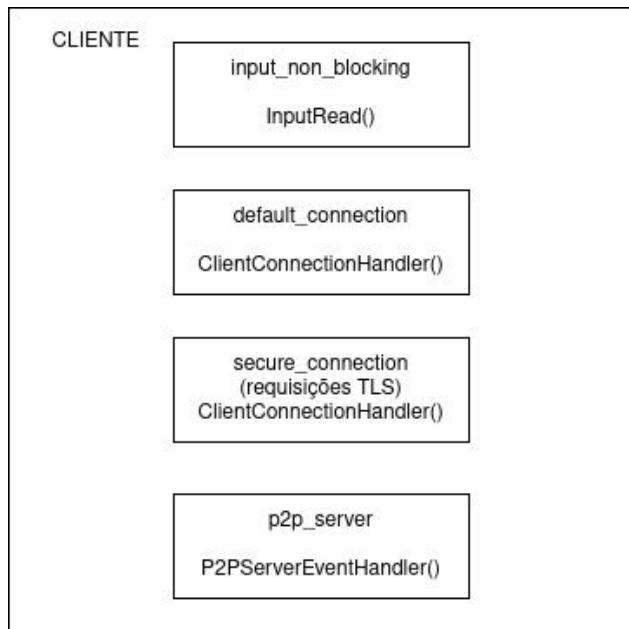
- No servidor, lidamos com os seguintes eventos:
 - **adduser** (TLS): insere um novo usuário no banco de dados, armazenando o novo nome de usuário e o hash de sua senha obtido com a biblioteca *bcrypt* (arquivo */server/src/auth.py*).
 - **login** (TLS): verifica se usuário existe e a senha está correta para então responder ao cliente que ele pode realizar o login.
 - **password_change** (TLS): valida entrada de usuário e, por fim, tenta atualizar a senha do usuário.
 - **new_user_connection**: caso o cliente tenha recebido como resposta um *OK* ao enviar uma requisição de *login*, o servidor trata a nova conexão, adicionando o usuário na lista de usuários ativos.
 - **list_players**: retorna como resposta ao cliente a lista de usuários ativos.
 - **leaderboard**: retorna como resposta ao cliente o ranking de usuários.
 - **init_game_permission**: verifica se é possível iniciar um novo jogo entre dois clientes.
 - **init_game**: atualiza os estados dos clientes que começaram um novo jogo.
 - **finish_game**: atualiza os estados dos clientes ao final de uma partida, além dos seus dados de partidas ganhas, perdidas ou empates no banco
 - **logout**: realiza ação de *logout* do cliente.



ServerEventHandler:

- Camada de baixo nível para tratar as conexões realizadas com o servidor
- Pode realizar tanto conexões padrões TCP como também criptografadas usando TLS
- Cria threads de execução para cada nova conexão e trata uma a uma, através de referências de *callbacks* passadas para cada tipo de evento inscrito no servidor.
- Pode enviar mensagens de *broadcast* para todas as conexões ativas no momento, possibilitando o envio de uma requisição de *heartbeat* a cada 60 segundos.

Visão Geral Arquitetura - Cliente



- Cliente possui 4 threads sendo executadas inicialmente.
 1. Leitura de entrada do usuário
 2. Controlador de conexões padrões TCP
 3. Controlador de conexões seguras usando TLS
 4. Servidor para atender requisições P2P



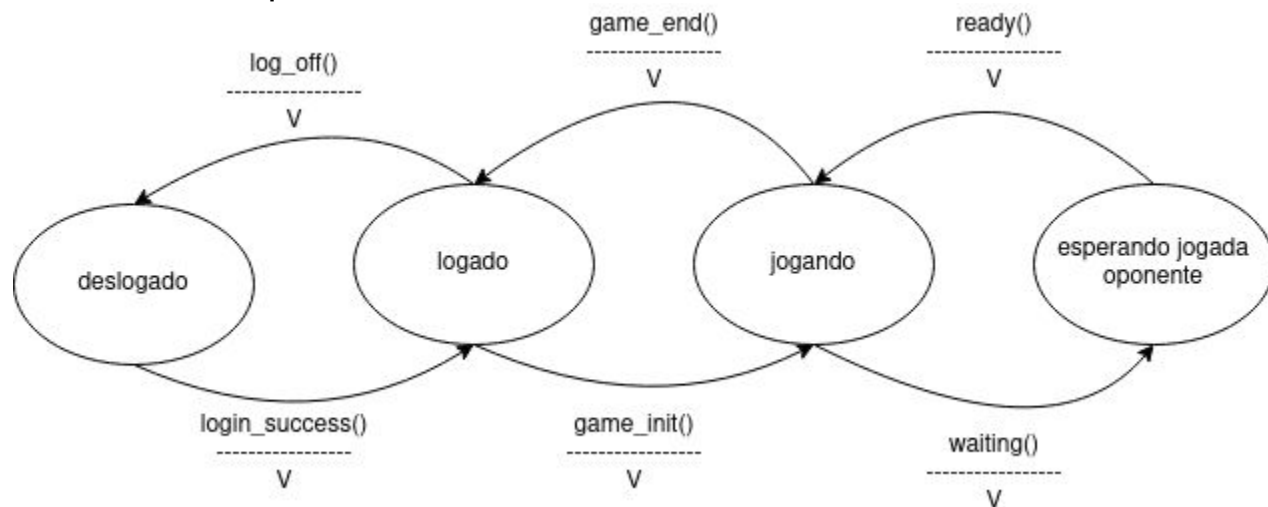
Detalhes de implementação - Cliente

Client:

- Executa uma instância da classe *InputRead*, responsável por ficar lendo as entradas dos usuários em uma thread e executando *callbacks* referentes ao comando solicitado. A leitura é *non-block*, o que possibilita o recebimento de convites para partidas e o posterior envio de uma resposta sem empecilhos.
- Executa em duas threads distintas, instâncias da classe *ClientConnectionHandler*. Tal classe implementa uma camada de baixo nível para as conexões realizadas com o servidor, facilitando a comunicação cliente/servidor.
- Executa em uma thread uma instância da classe *P2PServerEventHandler*, que funciona de forma análoga ao servidor, no entanto, ela trata especificamente a comunicação entre clientes.

Comandos do usuário:

- No cliente, aceitamos como entrada comandos específicos, que são executados de acordo com o estado do cliente. Um cliente pode ter os seguintes estados que variam de acordo com os comandos que são executados.





Comandos do usuário:

- Cada comando é representado em um dicionário, em que, as chaves são os nomes dos comandos e o seu valor é outro dicionário que possui uma referência para a *callback* a ser executada e o estado em que o cliente deve estar para executar tal comando.

```
self.commands = {
    "adduser": {
        "callback": self.__add_user,
        "state": [self.user_state.logged_out],
    },
    "login": {"callback": self.__login, "state": [self.user_state.logged_out]},
    "passwd": {
        "callback": self.__passwd,
        "state": [self.user_state.logged_out, self.user_state.logged],
    },
    "list": {"callback": self.__players, "state": [self.user_state.logged]},
    "leaders": {"callback": self.__leaders, "state": [self.user_state.logged]},
    "begin": {"callback": self.__new_game, "state": [self.user_state.logged]},
    "send": {"callback": self.__send, "state": [self.user_state.playing_game]},
    "end": {
        "callback": self.__end_game,
        "state": [
            self.user_state.playing_game,
            self.user_state.waiting_game_instruction,
        ],
    },
    "logout": {"callback": self.__logout, "state": [self.user_state.logged]},
    "exit": {
        "callback": self.__exit_client,
        "state": [self.user_state.logged_out, self.user_state.logged],
    },
}
```



Jogo da Velha:

- Existe uma classe *TicTacToe* (*game.py*) responsável pela lógica do jogo. Ao iniciar um jogo entre dois clientes, cada um deles iniciam uma instância da mesma e vão atualizando o jogo conforme os movimentos do jogador e o do seu oponente.



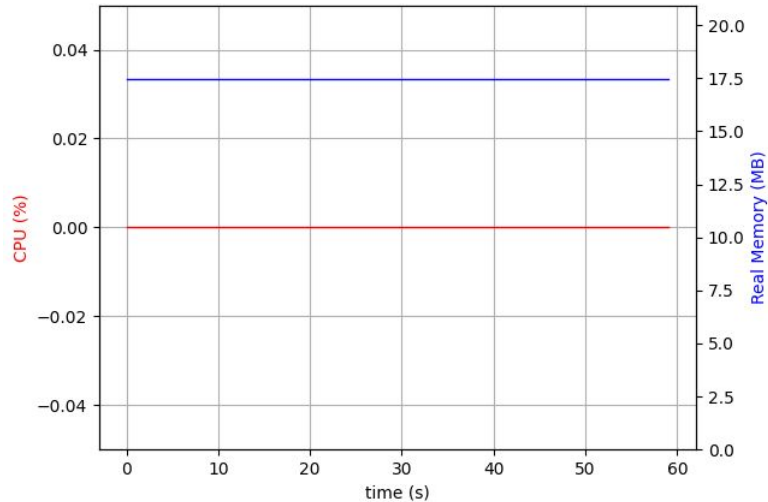
Desempenho

Cenário de Testes:

- Inicializa coleta de métricas de para processo com o PID do servidor e clientes inicializados;
 - Para CPU/Memória foi utilizado o programa psrecord;
 - Para Rede foi utilizado o Wireshark (Statistics -> I/O Graphs)
- Os testes que exigiam clientes foram feitos utilizando a maioria dos comandos disponíveis, e no caso para 2 clientes, foi feito alguns jogos no período de coleta de métricas.

Desempenho

Caso 1 - Apenas servidor:

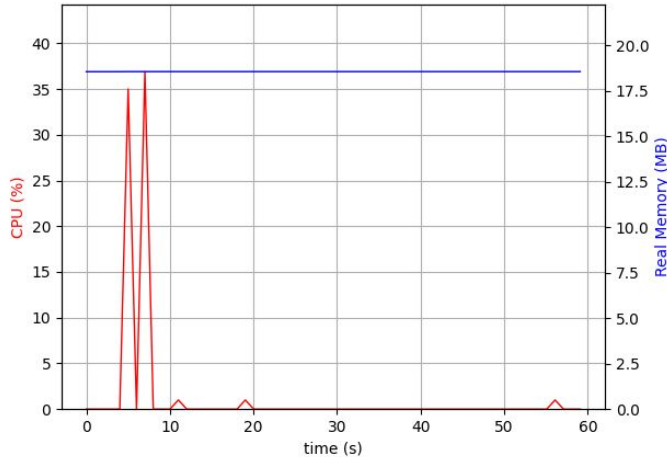


- Não houve consumo de CPU/rede para este caso
- Consumo de memória constante, apenas para inicializar servidor

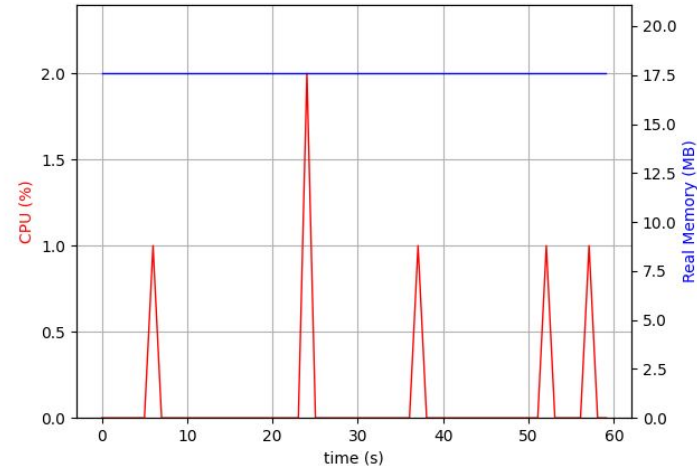
Desempenho

Caso 2 - Servidor + Cliente

Servidor



Cliente

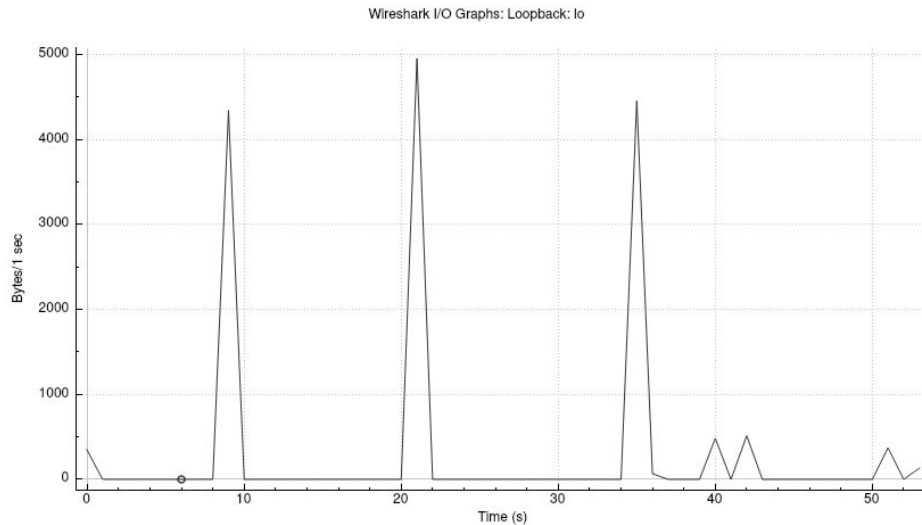


- Houve algumas anomalias na curva da CPU do servidor após um dado instante de tempo, em que, o consumo de CPU diminui abruptamente, apesar de ainda haver requisições para o servidor. Um dos motivos pode ser a falta de precisão do *psrecord*.
- Assim como no caso anterior, o consumo de memória foi constante também.

Desempenho

Caso 2 - Servidor + Cliente

- O consumo de rede, assim como o da CPU, apresentou picos durante as requisições, no entanto, aparentemente o consumo foi mais estável.

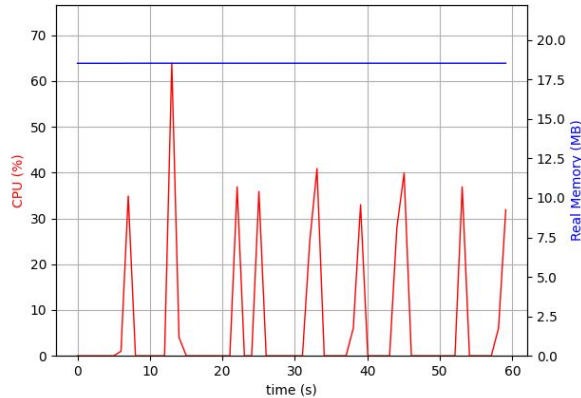


Desempenho

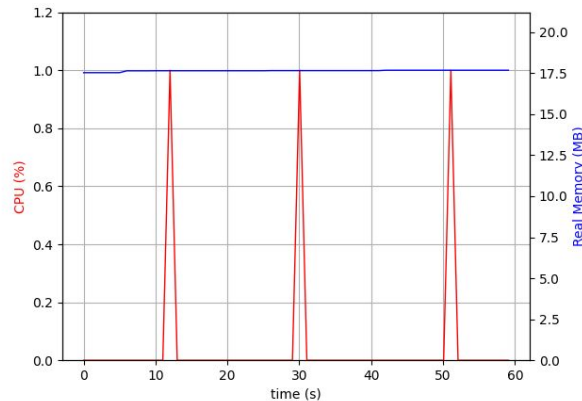
Caso 3 - Servidor + 2 Clientes

- Os picos de consumo de CPU provavelmente ocorrem durante as requisições do cliente. Além disso, há em algum momento uma requisição de heartbeat que o servidor envia para o cliente.
- De forma geral, os gráficos de consumo de CPU apresentaram comportamento análogo aos anteriores.

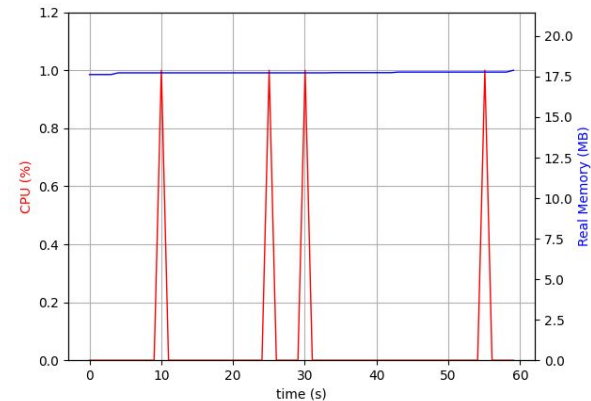
Servidor



Cliente 1



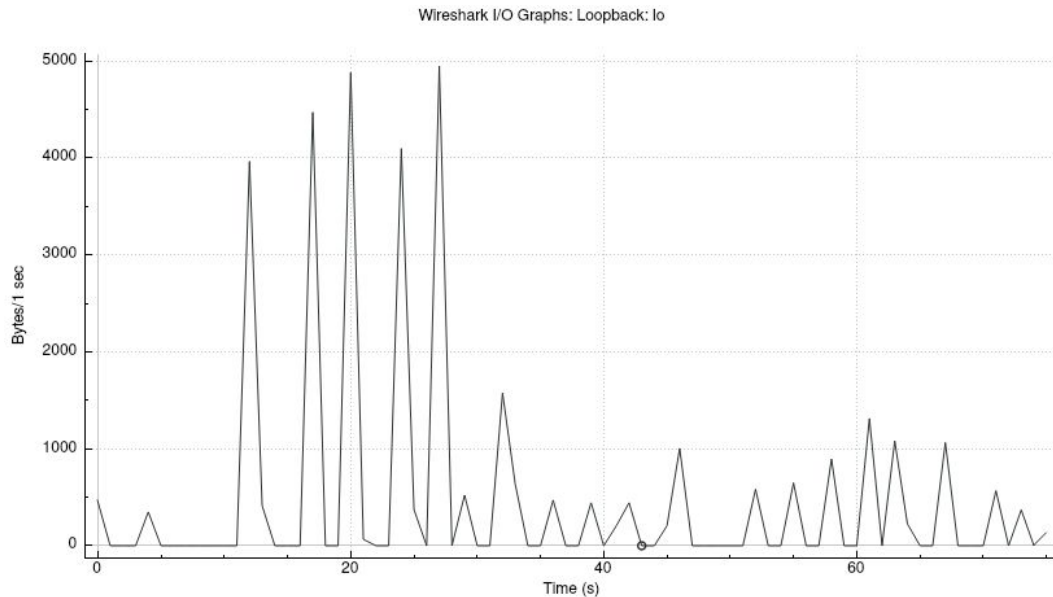
Cliente 2



Desempenho

Caso 3 - Servidor + 2 Clientes

- O consumo de rede neste caso foi claramente maior. Tal fato, pode ter ocorrido não só em virtude da maior interação entre clientes e o servidor, mas também, graças as interações de jogo realizadas entre clientes.





Desempenho

Ambiente de Testes:

- Arquitetura x86_64
- Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
- 8GB RAM
- Controlador Ethernet: Realtek Semiconductor Co., Ltd.
RTL8111/8168/8411 PCI Express Gigabit Ethernet
Controller