

ATIVIDADE 1: IMPLEMENTANDO UM SOCKET TCP

Disciplina: Plataformas de Distribuição.

Professor: Nelson Souto Rosa.

Integrantes: Maria Gabriela Alves Zuppardo e Victor Brunno Dantas de Souza Rosas.

Enunciado:

Implemente uma aplicação cliente-servidor usando socket TCP para coletar e exibir, em tempo quase real, métricas de desempenho de vários computadores. Cada cliente é instalado em uma máquina a ser monitorada e atua como um agente, reunindo periodicamente dados como (escolha apenas um deles para monitorar): uso de CPU (percentual de ocupação por núcleo e média geral), memória (total, utilizada, livre), disco (uso de espaço, taxa de leitura/escrita), e rede (taxa de upload/download, pacotes perdidos). Essas informações são enviadas em intervalos configuráveis (por exemplo, a cada 5 segundos) para o servidor, que mantém conexões persistentes com todos os agentes. Por fim, o servidor mantém uma lista de clientes conectados e armazena os dados em memória.

Implementação:

Foram criados dois arquivos sendo: [client.py](#) e [server.py](#). O servidor implementa um socket TCP capaz de lidar com múltiplas conexões simultaneamente usando threads, exemplo bem prático de uma aplicação de sistemas distribuídos.

server.py

A construção do servidor foi feita em python e utilizando as bibliotecas Socket, utilizado para criar a conexão de rede TCP e o Threading foi utilizado para o servidor lidar com os múltiplos clientes sem travar.

Para armazenar as métricas recebidas, foi criado o dicionário “**dados_clientes = {}**” e a chave do dicionário é o endereço de cada cliente (**addr**), e o valor é uma lista vazia, que será preenchida com as mensagens de métricas que ele envia.

Para a realização do tratamento individual de cada cliente, o servidor possui a função “**def handle_client(conn, addr)**” para cada nova conexão estabelecida e possui a **conn** (o objeto de conexão que permite enviar/receber dados) e o **addr** (o endereço IP e porta do cliente). Com a conexão estabelecida, o servidor cria uma nova entrada no dicionário **dados_clientes**, Enquanto o cliente está conectado, os dados são recebido (com capacidade de até 1024 bytes de clientes) em um ciclo **while** até que o **conn.recv()** retornar um objeto vazio, isso significa que a conexão foi encerrada pelo cliente. O **break** encerra o loop, e o servidor passa para o bloco **finally**. Enquanto os dados são enviados, a variável “**mensagem = data.decode()**” converte os dados recebidos (que são em bytes) para uma string legível

(**Unmarshalling**) e cada mensagem recebida é adicionada à lista de métricas do cliente correspondente no dicionário. Quando a conexão é finalizada, os dados são apagados.

A partir da função “**def start_server(..)**” o servidor é iniciado e gerenciado a chegada de novos clientes. A função citada recebe como parâmetro o host (0.0.0.0) para aceitar conexões de qualquer endereço de rede e a porta 5000 caso no futuro precisasse implementar um servidor em flask para fazer uma interface (<http://localhost:5000> porta padrão). Em seguida, o objeto socket do servidor é criado define o uso de endereços IPv4 e define o tipo de Socket que será TCP:

```
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

Posteriormente, o servidor é associado ao endereço e as portas especificados, tornando acessível (**server.bind((host, port))**), e o servidor é colocado em modo de escuta (**server.listen()**) pronto para receber requisições e as informações do novo cliente é impressa no terminal.

Dentro do loop do **while true**, o comando **conn, addr = server.accept()** é o mais importante do loop. Nele, o servidor pausa a sua execução e aguarda por uma nova conexão de cliente. Assim que um cliente se conecta, o método **accept()** retorna dois objetos: **conn**, que é a conexão em si e permite a comunicação bidirecional com o cliente, e **addr**, que é uma tupla contendo o endereço IP e a porta do cliente.

Após aceitar uma conexão, o servidor não a trata diretamente. Em vez disso, ele cria uma nova **thread** (uma linha de execução separada) para chamar a função **handle_client**. Isso é o que permite que o servidor lide com múltiplos clientes ao mesmo tempo, pois cada cliente terá sua própria thread dedicada. Assim que isso ocorre, o número atual de clientes que estão sendo monitorados é impressa no terminal (**print(f"[STATUS] Clientes conectados: {len(dados_clientes)}")**).

O comando “**if __name__ == "__main__"**” é uma prática comum em Python que garante que a função **start_server()** só seja executada quando o script for rodado diretamente.

client.py

No lado do cliente, ele se conecta a um servidor, coleta métricas de uso da CPU (parâmetro escolhido por nós) e as envia periodicamente.

Assim como o server, no cliente também é necessário a importação das bibliotecas necessárias. **socket** para a comunicação em rede, **psutil** para a coleta de dados do sistema, e **time** para controlar o intervalo entre os envios.

Para coletar os dados de uso do processador, cria-se a função **def coletar_metricas_cpu()**, que com o comando **psutil.cpu_percent(percpu=True)**, coleta o uso de CPU para cada núcleo individualmente. O argumento **percpu=True** faz com que a função retorne uma lista de porcentagens, uma para cada núcleo. Já **psutil.cpu_percent()** coleta a média geral de uso da CPU em todos os núcleos. A ausência do argumento **percpu** resulta em

um valor único. A função retorna um dicionário com os dois valores coletados, organizados com as chaves "por_nucleo" e "media".

Após os dados do processador coletados pelo cliente, é necessário estabelecer a conexão com o servidor por meio de **client = socket.socket(...)** em que cria um objeto socket do tipo TCP (**socket.SOCK_STREAM**) para se conectar ao servidor, enquanto o **client.connect((server_host, server_port))** estabelece a conexão com o servidor. O cliente tenta se conectar ao endereço e porta especificados (**127.0.0.1:5000** por padrão). Se o servidor não estiver rodando, a conexão irá falhar.

Após a conexão ser estabelecida, o cliente pode enviar os dados coletados para o servidor por meio da função de coleta de métricas chamada para obter os dados mais recentes (**metricas = coletar_metricas_cpu()**). Uma String formatada com os dados coletados (**mensagem = f"CPU -> Núcleos: {metricas['por_nucleo']} | Média: {metricas['media']}%"**) e depois enviados para o servidor, no entanto convertendo string para bytes usando **.encode()**, pois o socket só pode transmitir dados binários (**client.send(mensagem.encode())**) e espera um período de 5 segundo para realizar um novo envio (**time.sleep(intervalo)**). O processo de conversão da String para dados binários é conhecido como **Marshalling**.

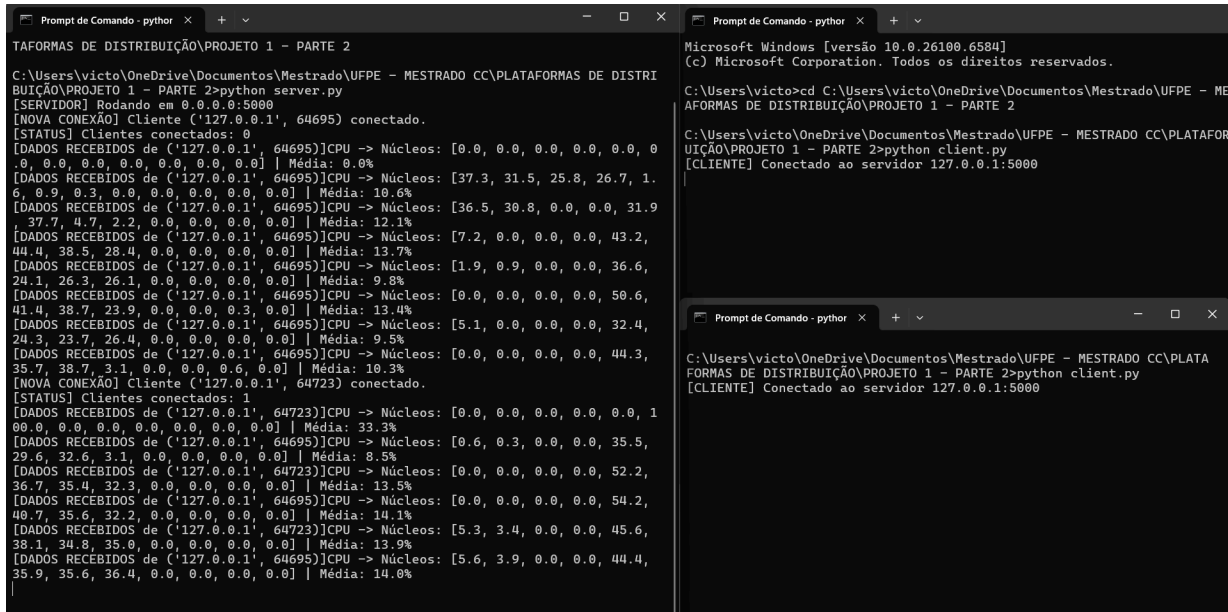
Caso o usuário pressione CTRL+C no terminal, o programa executa o **except KeyboardInterrupt**, exibe uma mensagem de encerramento manual e passa para o bloco **finally**. O bloco **finally** sempre é executado, seja qual for o motivo da saída do loop (seja por um erro ou por um encerramento manual). Por fim, **client.close()** fecha a conexão com o servidor de forma limpa, liberando os recursos da rede.

Por fim, assim como o servidor, o cliente também executa **if __name__ == "__main__":** por padrão em scripts Python garantindo que a função **start_client()** seja chamada apenas quando o arquivo for executado diretamente, e não quando ele for importado como um módulo por outro arquivo

Assim, finalizado a aplicação cliente-servidor usando um Socket TCP.

Anexo:

Figura 1: Aplicação Socket TCP com dois clientes e um servidor.



```
Prompt de Comando - pythor x + v
TAFORMAS DE DISTRIBUIÇÃO\PROJETO 1 - PARTE 2
C:\Users\victo\OneDrive\Documentos\Mestrado\UFPE - MESTRADO CC\PLATAFORMAS DE DISTRIBUIÇÃO\PROJETO 1 - PARTE 2>python server.py
[SERVIDOR] Rodando em 0.0.0.0:5000
[NOVA CONEXÃO] Cliente ('127.0.0.1', 64695) conectado.
[STATUS] Clientes conectados: 0
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] | Média: 0.0%
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [37.3, 31.5, 25.8, 26.7, 1.6, 0.9, 0.3, 0.0, 0.0, 0.0, 0.0, 0.0] | Média: 10.6%
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [36.5, 30.8, 0.0, 0.0, 31.9, 37.7, 4.7, 2.2, 0.0, 0.0, 0.0, 0.0] | Média: 12.1%
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [7.2, 0.0, 0.0, 0.0, 43.2, 44.4, 38.5, 28.4, 0.0, 0.0, 0.0, 0.0] | Média: 13.7%
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [1.9, 0.9, 0.0, 0.0, 36.6, 24.1, 26.3, 26.1, 0.0, 0.0, 0.0, 0.0] | Média: 9.8%
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [0.0, 0.0, 0.0, 0.0, 50.6, 41.4, 38.7, 23.9, 0.0, 0.0, 0.3, 0.0] | Média: 13.4%
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [5.1, 0.0, 0.0, 0.0, 32.4, 24.3, 23.7, 26.4, 0.0, 0.0, 0.0, 0.0] | Média: 9.5%
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [0.0, 0.0, 0.0, 0.0, 44.3, 35.7, 38.7, 3.1, 0.0, 0.0, 0.6, 0.0] | Média: 10.3%
[NOVA CONEXÃO] Cliente ('127.0.0.1', 64723) conectado.
[STATUS] Clientes conectados: 1
[DADOS RECEBIDOS de ('127.0.0.1', 64723)]CPU -> Núcleos: [0.0, 0.0, 0.0, 0.0, 0.0, 1.00, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] | Média: 33.3%
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [0.6, 0.3, 0.0, 0.0, 35.5, 29.6, 32.6, 3.1, 0.0, 0.0, 0.0, 0.0] | Média: 8.5%
[DADOS RECEBIDOS de ('127.0.0.1', 64723)]CPU -> Núcleos: [0.0, 0.0, 0.0, 0.0, 52.2, 36.7, 35.4, 32.3, 0.0, 0.0, 0.0, 0.0] | Média: 13.5%
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [0.0, 0.0, 0.0, 0.0, 54.2, 40.7, 35.6, 32.2, 0.0, 0.0, 0.0, 0.0] | Média: 14.1%
[DADOS RECEBIDOS de ('127.0.0.1', 64723)]CPU -> Núcleos: [5.3, 3.4, 0.0, 0.0, 45.6, 38.1, 34.8, 35.0, 0.0, 0.0, 0.0, 0.0] | Média: 13.9%
[DADOS RECEBIDOS de ('127.0.0.1', 64695)]CPU -> Núcleos: [5.6, 3.9, 0.0, 0.0, 44.4, 35.9, 35.6, 36.4, 0.0, 0.0, 0.0, 0.0] | Média: 14.0%

Microsoft Windows [versão 10.0.26100.6584]
(c) Microsoft Corporation. Todos os direitos reservados.
C:\Users\victo>cd C:\Users\victo\OneDrive\Documentos\Mestrado\UFPE - MESTRADO CC\PLATAFORMAS DE DISTRIBUIÇÃO\PROJETO 1 - PARTE 2
C:\Users\victo\OneDrive\Documentos\Mestrado\UFPE - MESTRADO CC\PLATAFORMAS DE DISTRIBUIÇÃO\PROJETO 1 - PARTE 2>python client.py
[CLIENTE] Conectado ao servidor 127.0.0.1:5000

Prompt de Comando - pythor x + v
C:\Users\victo\OneDrive\Documentos\Mestrado\UFPE - MESTRADO CC\PLATAFORMAS DE DISTRIBUIÇÃO\PROJETO 1 - PARTE 2>python client.py
[CLIENTE] Conectado ao servidor 127.0.0.1:5000
```

Fonte: Autores, 2025.