UNIVERSIDADE TRAS OS MONTES E ALTO DOURO

ENGENHARIA INFORMÁTICA

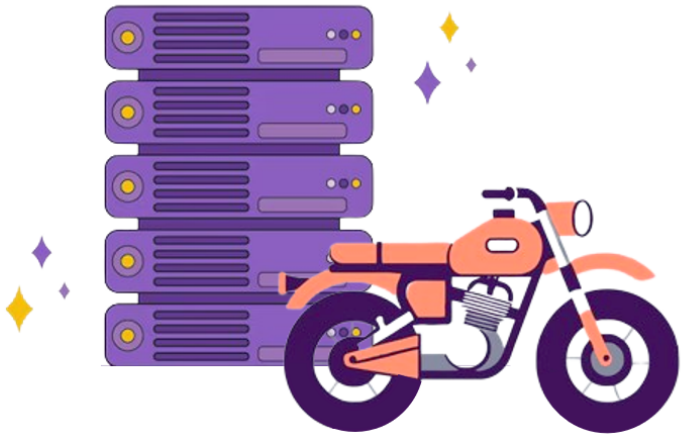
SISTEMAS DISTRIBUIDOS

TRABALHO PRACTICO 1

SERVIDOR DE GESTÃO DE SERVIÇOS DE MOBILIDADE

AUTOR
Raquel Ribeiro
al66766@utad.eu
Turma Pratica 1

DOCENTES
Prof. Hugo Paredes
Prof. Tiago Pinto



11 de junho de 2024

Conteúdo

1 // INTRODUÇÃO

O cliente e o servidor implementam um sistema de atribuição de tarefas, onde os clientes podem solicitar tarefas ao servidor, marcar tarefas como concluídas e encerrar a comunicação quando desejado. O servidor é responsável por gerir as solicitações dos clientes, alocar e distribuir tarefas, além de manter o controle sobre o estado das tarefas.

O cliente e o servidor que comunicam através do protocolo TCP usando sockets. A comunicação ocorre de forma assíncrona e permite o atendimento simultâneo de vários clientes.

2 // PROTOCOLO DE COMUNICAÇÃO

O protocolo de comunicação entre o cliente e o servidor é definido por uma série de comandos trocados através de uma conexão TCP.

As principais mensagens e os seus propósitos do lado do cliente sao:

- CONNECT: Inicia a ligação do cliente com o servidor.
- CLIENT ID:[id]: O cliente envia esta mensagem para se identificar ao servidor. so-called bullet.
- REQUEST TASK CLIENT ID:[id]: O cliente solicita uma nova tarefa.
- TASK COMPLETED: [descrição da tarefa]: O cliente informa o servidor que uma tarefa foi concluída.
- REQUEST SERVICE CLIENT ID:[id]: O cliente solicita um serviço.
- SAIR: O cliente informa o servidor que deseja desconectar-se.

No lado do servidor, as mensagens de resposta ao cliente sao:

- 100 OK: Ligação estabelecida.
- 400 BYE: O servidor reconhece o pedido de desconexão do cliente.
- 500 ERROR: [mensagem de erro]: O servidor encontrou um erro.
- ID CONFIRMED:[id]: O servidor confirma o ID do cliente.
- SERVICE ALLOCATED:[serviço]: O servidor atribui um serviço ao cliente.
- TASK ALLOCATED:[descrição da tarefa]: O servidor atribui uma tarefa ao cliente.
- NO SERVICE AVAILABLE: Não há serviços disponíveis para o cliente.
- NO TASKS AVAILABLE: Não há tarefas disponíveis.
- TASK MARKED COMPLETED: Tarefa marcada como concluída.

3 // ATENDIMENTO E COMUNICAÇÃO COM OS CLIENTES

O servidor aceita conexões de clientes utilizando um TcpListener na porta 1234. Cada conexão de cliente é tratada numa thread separada, permitindo que o servidor gere múltiplos clientes simultaneamente.

Cada cliente envia comandos ao servidor utilizando um StreamWriter e lê respostas utilizando um StreamReader. O servidor depois processa as mensagens de cada cliente e responde ou chama outros métodos de acordo com o pedido.

TCP listener, parte do Loop Principal do Servidor:

```
TcpListener servidor = null;
try
{
    // Initialize the TCP listener on port 1234
    servidor = new TcpListener(IPAddress.Any, 1234);
    servidor.Start();
    Console.WriteLine("Servidor iniciado. Aguardando conexões...");

    // Infinite loop to accept client connections
    while (true)
    {
        // Accept an incoming client connection
        TcpClient cliente = servidor.AcceptTcpClient();
        Console.WriteLine("Cliente conectado!");

        // Handle the client connection in a separate thread using the thread pool
        ThreadPool.QueueUserWorkItem(HandleClient, cliente);
    }
}
catch (SocketException ex)
{
    // Log any socket exceptions
    Console.WriteLine("Erro de Socket: " + ex.ToString());
}
finally
{
    // Stop the TCP listener if it was initialized
    if (servidor != null)
    {
        servidor.Stop();
    }
}
```

Aqui, o método HandleClient processa as mensagens de cada cliente, passando-as de seguida para o método ProcessMessage, que e realmente onde elas sao tratadas.

```
 private static void HandleClient(object obj)
 {
     TcpClient cliente = (TcpClient)obj;
     try
     {
         // Create network stream, reader, and writer for the client connection
         using (NetworkStream stream = cliente.GetStream())
         using (StreamReader leitor = new StreamReader(stream))
         using (StreamWriter escritor = new StreamWriter(stream) { AutoFlush = true })
         {
             string mensagem;
             // Read messages from the client and process them
             while ((mensagem = leitor.ReadLine()) != null)
             {
                 Console.WriteLine("Mensagem recebida: " + mensagem);
                 string resposta = ProcessMessage(mensagem);
                 // Send the response back to the client
                 escritor.WriteLine(resposta);
             }
         }
     }
     catch (IOException ex)
     {
         // Log any I/O exceptions
         Console.WriteLine("Erro de E/S: " + ex.ToString());
     }
     catch (Exception ex)
     {
         // Log any unexpected exceptions
         Console.WriteLine("Erro inesperado: " + ex.ToString());
     }
     finally
     {
         // Close the client connection
         if (cliente != null)
         {
             cliente.Close();
         }
     }
 }

 // Method to process incoming client messages
private static string ProcessMessage(string message)
{
    try
    {
        // Check if the message starts with "CONNECT"
```

```
        if (message.StartsWith("CONNECT", StringComparison.OrdinalIgnoreCase))
        {
            // Respond with a confirmation code
            return "100 OK";
        }

        //... VER ANEXOS PARA O RESTO DA LOGICA...

                // Check if the message is "SAIR" (exit)
        else (message.Equals("SAIR", StringComparison.OrdinalIgnoreCase))
        {
            // Respond with a disconnection code
            return "400 BYE";
        }
    }
    catch (Exception ex)
    {
        // Log any errors that occur during message processing
        Console.WriteLine($"Error processing message: {ex}");
        // Respond with a generic internal server error message
        return "500 ERROR: Internal server error";
    }
}
```

4 // PARTILHA FICHEIROS E ATENDIMENTO SIMULTÂNEO DE CLIENTES

O servidor carrega informações sobre a que serviço cada cliente pertence, através do ficheiro Alocacao Cliente Servico.csv, e dinamicamente encontra o local path para um ficheiro separado contendo as tarefas referentes a cada serviço, assim sabendo que tarefas alocar a um dado cliente, o estado de cada tarefa, e quem as completou ou esta a realizar.

```
private static void LoadServiceAllocationsFromCSV()
{
    // Define the file path for the service allocation CSV
    string baseDir = AppDomain.CurrentDomain.BaseDirectory;
    string serviceAllocationFilePath =
    Path.Combine(baseDir, "Alocacao_Cliente_Servico.csv");

    try
    {
        // Check if the CSV file exists
        if (File.Exists(serviceAllocationFilePath))
        {
            // Clear the service dictionary to ensure a clean start
```

```
            serviceDict.Clear();

            // Read each line from the CSV file, skipping the header
            foreach (var line in File.ReadLines(serviceAllocationFilePath).Skip(1))
            {
                // Split the line into parts based on comma separator
                var parts = line.Split(',');
                if (parts.Length >= 2)
                {
                    // Extract client ID and service ID from the parts
                    var clientId = parts[0].Trim();
                    var serviceId = parts[1].Trim();

                    // Populate the dictionary with client-service mappings
                    serviceDict[clientId] = serviceId;
                }
            }
            // Log successful loading of services from the CSV file
            Console.WriteLine("Serviços carregados com sucesso.");
        }

//... VER ANEXOS PARA O RESTO DO CODIGO...
```

Como este design de cliente/servidor opera num contexto de recursos partilhados, o uso de mutexes e threads assegura que dois clientes não possam aceder ou modificar o mesmo ficheiro concorrentemente. Este mecanismo de exclusão mútua é crucial para manter a integridade dos dados e evitar condições de corrida, o que pode levar a resultados imprevisíveis e inconsistências nos dados.

Em sistemas distribuídos, onde os recursos são acedidos por múltiplos nós da rede, os mutexes podem ser ferramentas cruciais na coordenação do acesso a esses recursos, garantindo que as operações de leitura e escrita sejam realizadas de forma ordenada e segura.
Encontram-se mutexes em vários locais no código onde alteração ou consulta de ficheiros e necessária, mas um bom exemplo e no inicio do método AllocateTask, onde e requerido consultas a vários ficheiros de modo a atribuir um serviço a um clientId, e depois consulta, leitura e possível modificação das tarefas referentes a esse serviço.

```
 private static string AllocateTask(string clientId)
 {
     // Ensure thread safety using a mutex
     mutex.WaitOne();
     try
     {
         // Check if the client has a service allocated
         if (!serviceDict.ContainsKey(clientId))
         {
             // If the client has no allocated service, return a message
             //indicating no service is available
             return "NO_SERVICE_AVAILABLE";
```

```
}

// Get the service ID allocated to the client
string service = serviceDict[clientId];
Console.WriteLine($"Client {clientId} belongs to service {service}");

// Define the file path for the service's tasks CSV
string serviceFilePath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
$"{service}.csv");
Console.WriteLine($"Loading tasks from {serviceFilePath}");

//... VER ANEXOS PARA O RESTO DO CODIGO...
```

5 // SERVIÇO DE GESTÃO

Os principais métodos do código colaboram para criar o serviço de gestão:

- PrintWorkingDirectory: Imprime o diretório de trabalho atual, ajudando na verificação dos processos que ocorrem do lado do servidor.
- LoadDataFromCSV: Carrega tarefas a partir de um ficheiro CSV, inicializando a lista de tarefas disponíveis no servidor.
- HandleClient: Gere a comunicação com o cliente por threads, garantindo que múltiplos clientes possam ser atendidos simultaneamente.
- ProcessMessage: Processa mensagens recebidas do cliente e gera respostas apropriadas ou interpreta os pedidos dos clientes, depois executando as ações necessárias.
- AllocateService: Atribui um serviço a um cliente, pela consulta de um ficheiro CSV designado.
- AllocateTask: Atribui uma tarefa a um cliente, consulta o ficheiro CSV do serviço referente ao cliente, vê que tarefas não estão alocadas e por ordem atribui-as ao cliente que fez o pedido.
- MarkTaskAsCompleted: Marca uma tarefa como concluída no ficheiro CSV correspondente, atualizando o estado da tarefa após a confirmação de que o cliente a completou.
- IsTaskAllocated: Verifica se uma tarefa está atribuída a algum cliente, consultando a lista de tarefas num dado ficheiro CSV para determinar o estado atual.
- IsTaskCompleted: Verifica se uma tarefa está marcada como concluída, ajudando a evitar a retribuição de tarefas já terminadas.
- UpdateTaskCSV: Atualiza o estado da tarefa no ficheiro CSV, garantindo que as alterações no estado das tarefas sejam persistidas de forma correta no armazenamento permanente.

6 // ANEXOS

6.1 // CLONE DO GITHUB

Todo o trabalho pratico pode ser clonado do github para facilidade de acesso através do link:

REPOSITORIO

HTTPS://GITHUB.COM/DXCCCII/DISTRIBUTEDSYSTEMSTP1

6.2 // CÓDIGO DO CLIENTE

```
using System;                  // Provides basic functionalities like console input and out
using System.Diagnostics;      // Provides classes for interacting with system processes
using System.IO;               // Provides classes for reading and writing to files
using System.Net.Sockets;      // Provides classes for creating TCP/IP client and server ap
using System.Threading;        // Provides classes for threading, including Thread.Sleep

class Cliente
{
static void Main(string[] args)
{
Console.WriteLine("Bem-vindo à ServiMoto!"); // Welcome message

// Prompt for the server's IP address
Console.Write("Por favor, insira o endereço IP do servidor: ");
string enderecoServidor = Console.ReadLine();

try
{
while (true) // Keep the client running indefinitely
{
// Connect to the server
using (TcpClient cliente = new TcpClient(enderecoServidor, 1234))
using (NetworkStream stream = cliente.GetStream())
using (StreamReader leitor = new StreamReader(stream))
using (StreamWriter escritor = new StreamWriter(stream) { AutoFlush = true })
{
Console.WriteLine("Conectado ao servidor. Aguardando resposta..."); // Connected to server

// Send CONNECT message to initiate communication
escritor.WriteLine("CONNECT");
string resposta = leitor.ReadLine();
Console.WriteLine("Resposta do servidor: " + resposta); // Server response
Thread.Sleep(1000); // Add a delay of 1 second

// If the connection was successfully established, request and send the client ID
```

```
if (resposta == "100 OK")
{
Console.Write("Por favor, insira o seu ID de cliente: ");
string idCliente = Console.ReadLine();

// Send the client ID to the server
escritor.WriteLine("CLIENT_ID:" + idCliente);

// Receive confirmation from the server
resposta = leitor.ReadLine();
Console.WriteLine("Resposta " + resposta); // Server response
Thread.Sleep(1000); // Add a delay of 1 second

if (resposta.StartsWith("ID_CONFIRMED"))
{
while (true)
{
// Present options to the user
Console.WriteLine("1. Solicitar tarefa");
Console.WriteLine("2. Marcar tarefa como concluída");
Console.WriteLine("3. Sair");
Console.Write("Escolha uma opção: ");
string opcao = Console.ReadLine();

if (opcao == "1")
{
// Request a new task
escritor.WriteLine("REQUEST_TASK CLIENT_ID:" + idCliente);
resposta = leitor.ReadLine();
Console.WriteLine("Resposta do servidor: " + resposta); // Server response
Thread.Sleep(1000); // Add a delay of 1 second

if (resposta.StartsWith("TASK_ALLOCATED"))
{
string descricaoTarefa = resposta.Substring("TASK_ALLOCATED:".Length).Trim();
Console.WriteLine("Tarefa alocada: " + descricaoTarefa); // Allocated task
}
else
{
Console.WriteLine("Não há tarefas disponíveis no momento."); // No available tasks message
}
}
else if (opcao == "2")
{
// Mark task as completed
Console.Write("Por favor, insira a descrição da tarefa concluída: ");
string descricaoTarefa = Console.ReadLine();
```

```
escritor.WriteLine("TASK_COMPLETED: " + descricaoTarefa);
resposta = leitor.ReadLine();
Console.WriteLine("Resposta do servidor: " + resposta); // Server response
Thread.Sleep(1000); // Add a delay of 1 second
}
else if (opcao == "3")
{
// End communication
escritor.WriteLine("SAIR");
resposta = leitor.ReadLine();
Console.WriteLine("Resposta do servidor: " + resposta); // Server response
Thread.Sleep(1000); // Add a delay of 1 second
break; // Break out of the loop after receiving server response
}
else
{
// Invalid option message
Console.WriteLine("Opção inválida. Por favor, tente novamente.");
}
}
}
}
}
// Communication with server ended message
Console.WriteLine("Comunicação com o servidor encerrada.");
break; // Exit the while loop to end the client
}
}
catch (IOException ex)
{
// Input/output error occurred message
Console.WriteLine("Ocorreu um erro de E/S: " + ex.ToString());
}
catch (Exception ex)
{
Console.WriteLine("Ocorreu um erro: " + ex.Message); // An error occurred message
}
finally
{
// Ensure the console window closes when execution is complete
Environment.Exit(0);
}
}
}
```

6.3 // CÓDIGO DO SERVIDOR

```
using System; // Provides basic functionalities like console input and output
using System.Collections.Generic; // Collections.Generic namespace provides classes that de
using System.IO; // Provides classes for reading and writing to files
using System.Linq; // Provides classes and interfaces that support queries that use Languag
using System.Net; // Provides a simple programming interface for many of the protocols used
using System.Net.Sockets; // Provides classes for creating TCP/IP client and server applica
using System.Threading; // Provides classes and interfaces that enable multithreaded progra

class Servidor
{
// Dictionary to store the mapping between client IDs and their allocated services
private static Dictionary<string, string> serviceDict =
new Dictionary<string, string>();

// Dictionary to store the tasks for each service, mapping task IDs to their descriptions
private static Dictionary<string, List<string>> taskDict =
new Dictionary<string, List<string>>();

// Mutex to ensure thread safety when accessing shared resources
private static Mutex mutex = new Mutex();

static void Main(string[] args)
{
// Print the current working directory for debugging purposes
PrintWorkingDirectory();

// Load service allocations from a CSV file
LoadServiceAllocationsFromCSV();

// Load tasks for all services from their respective CSV files
LoadDataFromCSVForAllServices();

TcpListener servidor = null;
try
{
// Initialize the TCP listener on port 1234
servidor = new TcpListener(IPAddress.Any, 1234);
servidor.Start();
Console.WriteLine("Servidor iniciado. Aguardando conexões...");

// Infinite loop to accept client connections
while (true)
{
// Accept an incoming client connection
TcpClient cliente = servidor.AcceptTcpClient();
```

```
Console.WriteLine("Cliente conectado!");

// Handle the client connection in a separate thread using the thread pool
ThreadPool.QueueUserWorkItem(HandleClient, cliente);
}
}
catch (SocketException ex)
{
// Log any socket exceptions
Console.WriteLine("Erro de Socket: " + ex.ToString());
}
finally
{
// Stop the TCP listener if it was initialized
if (servidor != null)
{
servidor.Stop();
}
}
}

private static void HandleClient(object obj)
{
TcpClient cliente = (TcpClient)obj;
try
{
// Create network stream, reader, and writer for the client connection
using (NetworkStream stream = cliente.GetStream())
using (StreamReader leitor = new StreamReader(stream))
using (StreamWriter escritor = new StreamWriter(stream) { AutoFlush = true })
{
string mensagem;
// Read messages from the client and process them
while ((mensagem = leitor.ReadLine()) != null)
{
Console.WriteLine("Mensagem recebida: " + mensagem);
string resposta = ProcessMessage(mensagem);
// Send the response back to the client
escritor.WriteLine(resposta);
}
}
}
catch (IOException ex)
{
// Log any I/O exceptions
Console.WriteLine("Erro de E/S: " + ex.ToString());
}
```

```
catch (Exception ex)
{
// Log any unexpected exceptions
Console.WriteLine("Erro inesperado: " + ex.ToString());
}
finally
{
// Close the client connection
if (cliente != null)
{
cliente.Close();
}
}
}

// Method to process incoming client messages
private static string ProcessMessage(string message)
{
try
{
// Check if the message starts with "CONNECT"
if (message.StartsWith("CONNECT", StringComparison.OrdinalIgnoreCase))
{
// Respond with a confirmation code
return "100 OK";
}
// Check if the message starts with "CLIENT_ID:"
else if (message.StartsWith("CLIENT_ID:", StringComparison.OrdinalIgnoreCase))
{
// Extract the client ID from the message
string clientId = message.Substring("CLIENT_ID:".Length).Trim();
Console.WriteLine($"Received CLIENT_ID: {clientId}");
// Respond with confirmation of the client ID
return $"ID_CONFIRMED:{clientId}";
}
// Check if the message starts with "TASK_COMPLETED:"
else if (message.StartsWith("TASK_COMPLETED:", StringComparison.OrdinalIgnoreCase))
{
// Extract the task description from the message
string taskDescription = message.Substring("TASK_COMPLETED:".Length).Trim();

// Get the client ID associated with the current connection
string clientId = GetClientIdFromMessage(message);

// Mark the task as completed and return the response
return MarkTaskAsCompleted(clientId, taskDescription);
}
```

```
// Check if the message starts with "REQUEST_SERVICE CLIENT_ID:"
else if
(message.StartsWith("REQUEST_SERVICE CLIENT_ID:", StringComparison.OrdinalIgnoreCase))
{
// Extract the client ID from the message
string clientId = message.Substring("REQUEST_SERVICE CLIENT_ID:".Length).Trim();
// Allocate a service to the client and return the response
return AllocateService(clientId);
}
// Check if the message starts with "REQUEST_TASK CLIENT_ID:"
else if
(message.StartsWith("REQUEST_TASK CLIENT_ID:", StringComparison.OrdinalIgnoreCase))
{
// Extract the client ID from the message
string clientId = message.Substring("REQUEST_TASK CLIENT_ID:".Length).Trim();
// Allocate a task to the client and return the response
return AllocateTask(clientId);
}
// Check if the message is "SAIR" (exit)
else if (message.Equals("SAIR", StringComparison.OrdinalIgnoreCase))
{
// Respond with a disconnection code
return "400 BYE";
}
else
{
// Respond with an error if the command is not recognized
return "500 ERROR: Comando não reconhecido";
}
}
catch (Exception ex)
{
// Log any errors that occur during message processing
Console.WriteLine($"Error processing message: {ex}");
// Respond with a generic internal server error message
return "500 ERROR: Internal server error";
}
}

// Method to extract the service ID from the message
private static string GetServiceIdFromMessage(string message)
{
// Extract the service ID from the message
// Here you need to implement the logic to extract the service ID from the message
// For example, if the message format is "TASK_COMPLETED: <serviceId> <taskDescription>"
// You can split the message and get the service ID from the second part
// Update this logic according to your message format
```

```
string[] parts = message.Split(' ');
if (parts.Length >= 3)
{
return parts[1].Trim();
}
else
{
// Return null or throw an exception if the service ID cannot be extracted
throw new ArgumentException("Service ID not found in the message.");
}
}

private static string GetClientIdFromMessage(string message)
{
string[] parts = message.Split(':');
if (parts.Length >= 2)
{
return parts[1].Trim();
}
return string.Empty;
}


private static void PrintWorkingDirectory()
{
// Print the current working directory to the console
string workingDirectory = Environment.CurrentDirectory;
Console.WriteLine("Current Working Directory: " + workingDirectory);
}

// Method to load service allocations from a CSV file
private static void LoadServiceAllocationsFromCSV()
{
// Define the file path for the service allocation CSV
string baseDir = AppDomain.CurrentDomain.BaseDirectory;
string serviceAllocationFilePath = Path.Combine(baseDir, "Alocacao_Cliente_Servico.csv");

try
{
// Check if the CSV file exists
if (File.Exists(serviceAllocationFilePath))
{
// Clear the service dictionary to ensure a clean start
serviceDict.Clear();

// Read each line from the CSV file, skipping the header
foreach (var line in File.ReadLines(serviceAllocationFilePath).Skip(1))
```

```
{
// Split the line into parts based on comma separator
var parts = line.Split(',');
if (parts.Length >= 2)
{
// Extract client ID and service ID from the parts
var clientId = parts[0].Trim();
var serviceId = parts[1].Trim();

// Populate the dictionary with client-service mappings
serviceDict[clientId] = serviceId;
}
}
// Log successful loading of services from the CSV file
Console.WriteLine("Serviços carregados com sucesso.");
}
else
{
// Log an error if the CSV file doesn't exist
Console.WriteLine($"Erro: Arquivo {serviceAllocationFilePath} não encontrado.");
}
}
catch (Exception ex)
{
// Log any errors encountered during CSV file loading
Console.WriteLine($"Erro ao carregar dados dos arquivos CSV: {ex.Message}");
}
}

// Method to allocate a service to a client based on their client ID
private static string AllocateService(string clientId)
{
// Check if the client has a service allocated
if (serviceDict.ContainsKey(clientId))
{
// Get the service ID allocated to the client and log the allocation
string service = serviceDict[clientId];
Console.WriteLine($"Alocando serviço '{service}' para o cliente {clientId}");
return "SERVICE_ALLOCATED:" + service;
}
else
{
// If the client has no allocated service, return a message indicating no service
//is available
return "NO_SERVICE_AVAILABLE";
}
}
```

```
// Method to load data from CSV files for all services
private static void LoadDataFromCSVForAllServices()
{
// Iterate through each service in the service dictionary for debugging purposes
foreach (var serviceId in serviceDict.Values)
{
// Define the file path for the service's tasks CSV
string serviceFilePath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, $"{serviceId}
Console.WriteLine
($"Loading tasks for service '{serviceId}' from {serviceFilePath}");

// Load tasks for the current service from its respective CSV file
//for debugging purposes
LoadDataFromCSV(serviceFilePath);
}
}

// Method to load data from a CSV file for a specific service
private static void LoadDataFromCSV(string serviceFilePath)
{
try
{
// Check if the CSV file exists
if (File.Exists(serviceFilePath))
{
// Clear the task dictionary to ensure a clean start
taskDict.Clear();

// Read each line from the CSV file, skipping the header
foreach (var line in File.ReadLines(serviceFilePath).Skip(1))
{
// Log the processing of each line for debugging purposes
Console.WriteLine($"Processing line: {line}");

// Split the line into parts based on comma separator
var parts = line.Split(',');
if (parts.Length >= 3)
{
// Extract task details from the parts
var taskId = parts[0].Trim();
var taskDescription = parts[1].Trim();
var taskStatus = parts[2].Trim();

// Extract optional client ID if available
var clientId = parts.Length > 3 ? parts[3].Trim() : null;
```

```
// Log task details for debugging purposes
Console.WriteLine
($"Task ID: {taskId}, Description: {taskDescription}, Status: {taskStatus},
Client ID: {clientId}");

// Check if the task is unallocated
if (taskStatus.Equals("Nao alocada", StringComparison.OrdinalIgnoreCase))
{
// If the task is unallocated, add it to the task dictionary
if (!taskDict.ContainsKey(taskId))
{
taskDict[taskId] = new List<string>();
Console.WriteLine($"Created new task entry for ID: {taskId}");
}
taskDict[taskId].Add(taskDescription);
Console.WriteLine($"Added task '{taskDescription}' to taskDict under ID: {taskId}");
}
else
{
// If the task is already allocated, skip it
Console.WriteLine($"Task {taskId} is already allocated to client {clientId}. Skipping.");
}
}
}
// Log successful loading of tasks from the CSV file
Console.WriteLine($"Tarefas carregadas com sucesso de {serviceFilePath}.");
}
else
{
// Log an error if the CSV file doesn't exist
Console.WriteLine($"Erro: Arquivo {serviceFilePath} não encontrado.");
}
}
catch (Exception ex)
{
// Log any errors encountered during CSV file processing
Console.WriteLine
($"Erro ao carregar dados do arquivo CSV {serviceFilePath}: {ex.Message}");
}
}

// Method to allocate a task to a client based on their client ID
private static string AllocateTask(string clientId)
{
// Ensure thread safety using a mutex
mutex.WaitOne();
try
```

```
{
// Check if the client has a service allocated
if (!serviceDict.ContainsKey(clientId))
{
// If the client has no allocated service, return a message indicating no service
//is available
return "NO_SERVICE_AVAILABLE";
}

// Get the service ID allocated to the client
string service = serviceDict[clientId];
Console.WriteLine($"Client {clientId} belongs to service {service}");

// Define the file path for the service's tasks CSV
string serviceFilePath =
 Path.Combine(AppDomain.CurrentDomain.BaseDirectory, $"{service}.csv");
Console.WriteLine($"Loading tasks from {serviceFilePath}");

// Reload the tasks from the CSV file to get the latest state
LoadDataFromCSV(serviceFilePath);

// Log the number of tasks loaded from the CSV file for debugging purposes
Console.WriteLine($"Found {taskDict.Count} tasks loaded from {serviceFilePath}");
Console.WriteLine($"Verifying unallocated tasks for service '{service}'");

// Iterate through the tasks to find an unallocated one
foreach (var kvp in taskDict)
{
foreach (var taskDescription in kvp.Value)
{
if (!IsTaskAllocated(serviceFilePath, kvp.Key, taskDescription))
{
// If the task is unallocated, allocate it to the client
Console.WriteLine($"Task '{taskDescription}'
is unallocated. Allocating to client {clientId}.");
// Update the task's status to "Em curso" (in progress) and assign it to the client
UpdateTaskCSV(serviceFilePath, kvp.Key, "Em curso", clientId);
return $"TASK_ALLOCATED:{taskDescription}";
}
else
{
// If the task is already allocated, skip it
Console.WriteLine($"Task '{taskDescription}' is already allocated. Skipping.");
}
}
}
```

```
// If no unallocated task is found, return a message indicating no task is available
return "NO_TASK_AVAILABLE";
}
finally
{
// Release the mutex to allow other threads to access shared resources
mutex.ReleaseMutex();
}
}

// Method to check if a task is already allocated in a CSV file
private static bool IsTaskAllocated
(string serviceFilePath, string taskId, string taskDescription)
{
try
{
// Check if the CSV file exists
if (File.Exists(serviceFilePath))
{
// Read each line from the CSV file, skipping the header
foreach (var line in File.ReadLines(serviceFilePath).Skip(1))
{
// Split the line into parts based on comma separator
var parts = line.Split(',');
if (parts.Length >= 3)
{
// Extract task ID, task description, and task status from the parts
var loadedTaskId = parts[0].Trim();
var loadedTaskDescription = parts[1].Trim();
var loadedTaskStatus = parts[2].Trim();

// Check if the loaded task matches the specified task ID and task description
if (loadedTaskId == taskId && loadedTaskDescription == taskDescription)
{
// Task is considered allocated if its status is not "Nao alocada"
return !loadedTaskStatus.Equals("Nao alocada", StringComparison.OrdinalIgnoreCase);
}
}
}
}
else
{
// Log an error if the CSV file doesn't exist
Console.WriteLine($"Erro: Arquivo {serviceFilePath} não encontrado.");
}
}
catch (Exception ex)
```

```
{
// Log any errors encountered while checking task allocation
Console.WriteLine($"Erro ao verificar se a tarefa está alocada: {ex.Message}");
}

// If the task is not found in the file, assume it is unallocated
return false;
}

// Method to update the status of a task in the CSV file
private static void UpdateTaskCSV(string serviceFilePath, string taskId, string newStatus,
{
try
{
// Check if the CSV file exists
if (File.Exists(serviceFilePath))
{
// Read all lines from the CSV file and store them in a list
List<string> lines = File.ReadAllLines(serviceFilePath).ToList();

// Iterate through each line starting from the second line (skipping the header)
for (int i = 1; i < lines.Count; i++)
{
// Split the line into parts based on comma separator
string[] parts = lines[i].Split(',');
if (parts.Length >= 3 && parts[0].Trim() == taskId)
{
// If the task ID matches, update the task's status and assigned client ID
lines[i] = $"{taskId},{parts[1]},{newStatus},{clientId}";

// Write the modified lines back to the CSV file
File.WriteAllLines(serviceFilePath, lines);
break;
}
}
}
else
{
// If the CSV file doesn't exist, log an error message
Console.WriteLine($"Erro: Arquivo {serviceFilePath} não encontrado.");
}
}
catch (Exception ex)
{
// Log any errors encountered during CSV file updating
Console.WriteLine($"Erro ao atualizar o arquivo CSV: {ex.Message}");
}
```

```
}

private static string ProcessMarkTaskCompleted(string message, string clientId)
{
// Extrair a descrição da tarefa da mensagem
string[] parts = message.Split(':');
if (parts.Length >= 2)
{
string descricaoTarefa = parts[1].Trim();

// Chamar o método MarkTaskAsCompleted com o clientId
string resposta = MarkTaskAsCompleted(clientId, descricaoTarefa);

return resposta;
}
else
{
return "500 ERROR: Descrição da tarefa ausente.";
}
}

private static string MarkTaskAsCompleted(string clientId, string taskDescription)
{
// Iterate through all services
foreach (var serviceId in serviceDict.Values)
{
// Construct the file path for the service
string serviceFilePath =
 Path.Combine(AppDomain.CurrentDomain.BaseDirectory, $"{serviceId}.csv");

try
{
if (File.Exists(serviceFilePath))
{
// Read all lines from the service file
List<string> lines = File.ReadAllLines(serviceFilePath).ToList();

// Iterate through each line
foreach (string line in lines)
{
// Loop through all the lines in the file
for (int i = 1; i < lines.Count; i++)
{
string[] parts = lines[i].Split(',');
if (parts.Length >= 4 && parts[1].Trim() == taskDescription)
{
// Mark the task as completed
```

```
parts[2] = "Concluido";

// Update the line in the list of lines
lines[i] = string.Join(",", parts);

// Write the updated lines back to the file
File.WriteAllLines(serviceFilePath, lines);

// Return success message
return "TASK_MARKED_COMPLETED";
}
}
}
}
else
{
// Service file not found
return $"ERROR_FILE_NOT_FOUND:{serviceFilePath}";
}
}
catch (Exception ex)
{
// Error occurred while marking task as completed
Console.WriteLine($"Error marking task as completed: {ex.Message}");
return "ERROR_MARKING_TASK_COMPLETED";
}
}

// Task description not found
return "ERROR_TASK_NOT_FOUND";
}
}
```

6.4 // AVISO

Para ver os resultados das alterações feitas as tarefas nos ficheiros CSV, por favor abri-los na pasta de debug dentro do projeto depois de encerrar a comunicação com o servido. Obrigado.