UNIVERSIDADE TRAS OS MONTES E ALTO DOURO

ENGENHARIA INFORMÁTICA

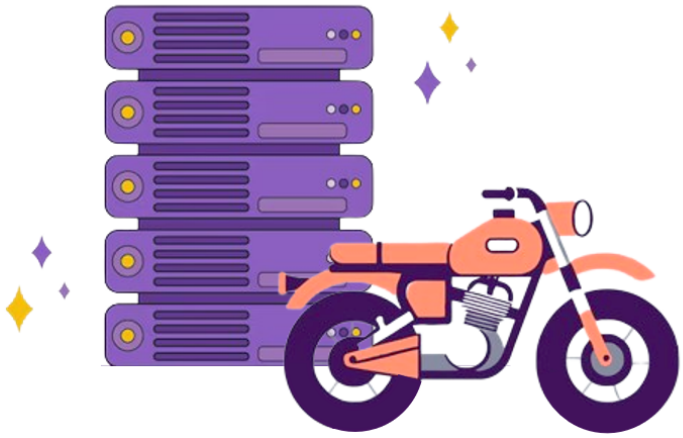
SISTEMAS DISTRIBUIDOS

TRABALHO PRÁCTICO 2

GESTÃO DE CLIENTES E SERVIÇOS DE MOBILIDADE

AUTOR
Raquel Ribeiro
al66766@utad.eu
Turma Pratica 1

DOCENTES
Prof. Hugo Paredes
Prof. Tiago Pinto



20 de junho de 2024

Conteúdo

1 // INTRODUÇÃO

Pretendendo-se dar continuidade ao sistema cliente/servidor do trabalho practico 1, capaz de efetuar a gestão de serviços e respetivas tarefas, assegurando a continuidade das funcionalidades já implementadas, neste trabalho introduz-se uma nova entidade no sistema: o Administrador.

O Administrador é capaz de gerir um serviço inteiro, podendo adicionar novas tarefas e editar as tarefas de qualquer serviço, visto que neste trabalho as motas foram consideradas os clientes, ou que cada cliente tinha a sua própria mota (ex: Uber).

Na perspetiva da arquitetura do sistema cliente/servidor, o Administrador será um cliente, consumindo as funcionalidades disponíveis pelo servidor para este tipo específico de cliente.

Também serão tratados métodos síncronos e assíncronos de comunicação, bem como um sistema de subscrição/notificações .

2 // PROTOCOLO

O protocolo de comunicação utilizado entre o cliente e o servidor é baseado no RabbitMQ, uma plataforma de mensagens que implementa Advanced Message Queuing Protocol. RabbitMQ facilita a troca de mensagens de forma assíncrona entre o servidor e os clientes através de queues (onde mensagens esperam em filas para ser entregues) e exchanges.
O RabbitMQ faz uso de Remote Procedure Calls, ou RPC, e da rpc queue utilizada para procedure calls onde o cliente envia um pedido, ou call, e o servidor responde.

Para o serviço de subscrição foram usadas Service Notifications Exchanges, ou simplesmente service notifications para publicar notificações para os clientes subscritos a um dado serviço.

Manteve-se o uso de mutexes e threads para garantir a segurança das threads ao consultar e modificar dados partilhados em ficheiros, como as tarefas e serviços nos respectivos ficheiros CSV.

3 // ATENDIMENTO DE CLIENTES E COMUNICAÇÃO COM CADA CLIENTE

No servidor é criada uma rpc queue que escuta continuamente por pedidos vindos de clientes. Quando uma mensagem é recebida, o servidor processa o pedido e envia a resposta apropriada, se possivel, de volta.

No excerto abaixo, podemos ver a queue a ser inicializada e que quando quer responder a um pedido chama o método HandleRequest, que é o método que contem outros métodos apropriados a responder a diferentes pedidos.

```
static void Main(string[] args) // Main method, entry point of the application.
{
InitRabbitMQ(); // Initialize RabbitMQ connection and channel.
```

```
PrintWorkingDirectory(); // Print the current working directory.
LoadServiceAllocationsFromCSV(); // Load service allocations from a CSV file.
LoadDataFromCSVForAllServices(); // Load tasks data from CSV files for all services.

// Create a new consumer for RabbitMQ.
var consumer = new EventingBasicConsumer(rabbitChannel);
// Define the event handler for received messages.
consumer.Received += (model, ea) =>
{
var body = ea.Body.ToArray(); // Get the body of the message.
var props = ea.BasicProperties; // Get the properties of the message.
// Create properties for the reply message.
var replyProps = rabbitChannel.CreateBasicProperties();
replyProps.CorrelationId = props.CorrelationId; // Set the correlation ID for
//the reply.

string response = null; // Initialize response to null.
try
{
var message = Encoding.UTF8.GetString(body); // Convert message body to string.
Console.WriteLine($"Received message: {message}"); // Print the received message.
response = HandleRequest(message); // Handle the request and get the response.
}
catch (Exception ex) // Catch any exceptions.
{
// Print the error message.
Console.WriteLine($"Error handling request: {ex.Message}");
response = "500 INTERNAL SERVER ERROR"; // Set response to internal server error.
}
}
// ...VER ANEXOS...
```

O cliente conecta-se ao servidor pelo channel criado pelo RabbitMQ e envia as calls para a rpc queue, aguardando respostas usando tambem uma queue de resposta exclusiva e correlaciona as respostas aos pedidos usando IDs de correlação.

```
 InitRabbitMQ(enderecoServidor); // Initializing RabbitMQ with the server IP address

// Printing a message indicating connection to the server

Console.WriteLine("Conectado ao servidor. Aguardando resposta...");
// Sending "CONNECT" message to the server and waiting for a response
 string response = Call("CONNECT");
// Printing the server's response
Console.WriteLine("Resposta do servidor: " + response);
```

```
if (response == "100 OK") // Checking if the response is "100 OK"

//... VER ANEXOS...


private static string Call(string message)
{
// Converting the message to a byte array
var messageBytes = Encoding.UTF8.GetBytes(message);
rabbitChannel.BasicPublish(exchange: "", routingKey: "rpc_queue", basicProperties:
props, body: messageBytes); // Publishing the message to the RabbitMQ queue

var responseReceived = false; // Flag to check if the response is received
string response = null; // Variable to store the response

consumer.Received += (model, ea) => // Event handler for received messages
{
// Checking if the correlation ID matches
if (ea.BasicProperties.CorrelationId == correlationId)
{
// Decoding the response body
response = Encoding.UTF8.GetString(ea.Body.ToArray());
responseReceived = true; // Setting the flag to true
}
};

// Consuming messages from the reply queue
rabbitChannel.BasicConsume(queue: replyQueueName, autoAck: true, consumer: consumer);

while (!responseReceived) // Waiting for the response
{
// Busy-wait for the response
}

return response; // Returning the response
}
```

4 // PROCEDIMENTOS REMOTOS EXPOSTOS

Os seguintes procedimentos remotos são expostos pelo servidor e podem ser chamados pelo cliente:

- CONNECT: Estabelece a conexão inicial;
- CLIENT ID: e PASSWORD: Autentica a identidade do cliente;

- ADMIN SERVICE ID: Aloca o Administrador a um serviço;
- ADD TASK: Adiciona uma tarefa ao ficheiro CSV de um serviço;
- CONSULT TASKS: Consulta todas as tarefas no ficheiro CSV de um serviço.
- CHANGE TASK STATUS: Altera o estado de uma tarefa;
- REQUEST TASK: Aloca a um cliente uma tarefa do seu respectivo serviço;
- TASK COMPLETED: Marca uma tarefa como concluída;.
- SUBSCRIBE / UNSUBSCRIBE: Subscreve / cancela a subscrição de um cliente a um serviço, de modo a receber ou nao notificações sobre actualizações feitas ao serviço.

5 // COMUNICAÇÃO ASSÍNCRONA

As mensagens trocadas entre o cliente e o servidor seguem um formato específico, dependendo do pedido efectuado. Alguns exemplos de mensagens são:

- CONNECT: Solicitação para estabelecer a conexão.
- CLIENT ID:<clientId>: Autenticação do cliente pelo ID.
- PASSWORD:<clientId>,<password>: Autenticação do cliente pela password.
- ADD TASK:<serviceId>|<taskDescription>: Call para adicionar uma tarefa.O path para o ficheiro utiliza o serviceId, visto que o nome do ficheiro CSV de qualquer serviço é serviceId.csv. A taskDescription, esta dividida em dois campos que são appended - taskId (no formato SXTN, onde X é a letra do serviço e N o numero da tarefa) e a sua descrição, separadas por uma virgula.
- CONSULT TASKS:<serviceId>: Call para consultar tarefas de um ficheiro. O path para o ficheiro é construido da mesma maneira que na call ADD TASK.
- CHANGE TASK STATUS:<serviceId>|<taskDescription>,<newStatus>,<additionalField>: Call para mudar o estado de progresso de uma tarefa. O path para encontrar o ficheiro é construido como anteriormente, e os outros campos sao separados, o método sabendo que os campos de uma tarefa sao separados por virgulas, e editados separadamente pelo cliente, neste caso, o Administrador.
- REQUEST TASK:<clientId>: Call para pedir uma nova tarefa.
- TASK COMPLETED:<clientId>|<taskDescription>: Call para marcar uma tarefa como concluída. o clientId neste caso serve de facto para encontrar o servico, comparando o primeiro campo do ficheiro serviceallocations.csv, correspondente ao dado clientId, com o segundo campo, que denota a que serviço aquele cliente pertence, adquirindo assim o serviceId e construindo então o path para o ficheiro CSV do serviço, serviceId.csv.
- SUBSCRIBE:<clientId>|<serviceId>: Pedido de subscrição a um dado serviço.
- UNSUBSCRIBE:<clientId>|<serviceId>: Pedido para cancelar a subscrição a um serviço.

6 // GRUPOS EXTERNOS PARA A TROCA DE INFORMAÇÃO

Os serviços e clientes que interagem através do RabbitMQ são os mecanismos para troca de informação externa. Cada cliente pode estar inscrito num ou mais serviços, a realizar varias tarefas, identificados nos respectivos ficheiros, e o servidor gere as inscrições e notificações para esses serviços.

Os serviços e clientes comunicam principalmente através de pedidos ou mensagens publicadas nas queues e por exchanges no RabbitMQ.

6.1 EX: SUBSCRIÇÕES E NOTIFICAÇÕES

O cliente pode querer notificações de serviços específicos e receber atualizações sobre as tarefas desses serviços ao subscrever-se aos mesmos. O processo ocorre da seguinte forma:

1. O cliente subscreve ao serviço X para receber notificações de actualizações nesse serviço. O cliente passa a estar binded a um notification channel, que contem uma queue por onde serão passadas as notificações.

```
private static void BindNotificationQueue(string clientId, string serviceId)
{
// Constructing the queue name
var queueName = $"{clientId}_{serviceId}";
notificationChannel.QueueDeclare(queue: queueName, durable: true, exclusive:
false, autoDelete: false, arguments: null); // Declaring the queue
// Binding the queue to the exchange
notificationChannel.QueueBind(queue: queueName, exchange:
"service_notifications", routingKey: $"NOTIFICATION.{serviceId}");

// Creating a new EventingBasicConsumer for notifications
var consumer = new EventingBasicConsumer(notificationChannel);
consumer.Received += (model, ea) => // Event handler for received notifications
{
var body = ea.Body.ToArray(); // Getting the body of the message
var message = Encoding.UTF8.GetString(body); // Decoding the message
// Checking if the message does not start with "UNSUBSCRIBE
if (!message.StartsWith("UNSUBSCRIBE:")) "
{
// Printing the received notification
Console.WriteLine($"\n");
Console.WriteLine($"\nReceived notification for {serviceId}: {message}");
}
else
{
var parts = message.Split('-'); // Splitting the message by "-"
// Checking if the message format matches
if (parts.Length == 3 && parts[1] == clientId && parts[2] == serviceId)
```

```
{
// Unbinding the queue from the exchange
notificationChannel.QueueUnbind(queue: queueName, exchange:
"service_notifications", routingKey: $"NOTIFICATION.{serviceId}");
// Printing the unbound message
Console.WriteLine($"Unbound from notifications for service: {serviceId}");
}
}
};
// Consuming messages from the queue
notificationChannel.BasicConsume(queue: queueName, autoAck: true, consumer:
consumer);
// Printing the subscribed message
Console.WriteLine($"Subscribed to notifications for service: {serviceId}");
}
```

2. O servidor, depois passa a publicar as notificações para o cliente, através do notification channel, quando ocorrem eventos relevantes (como a adição de uma nova tarefa), mesmo que o cliente não esteja online no momento em que ocorrem. A queue permite que assim que volte a aceder ao sistema receba as notificações.

```
// Method to publish a notification.
private static void PublishNotification(string message, bool isAdminChange)
{
var body = Encoding.UTF8.GetBytes($"{message}"); // Convert the message to bytes.
string exchange = "service_notifications"; // Define the exchange name.
// Define the routing key using the service ID.
string routingKey = $"NOTIFICATION.{message.Split(':')[1]}";
rabbitChannel.BasicPublish(exchange: exchange, routingKey: routingKey,
basicProperties: null, body: body); // Publish the message.
// Print the notification message.
Console.WriteLine($"Sent notification: {message} with routing key {routingKey}");
}
```

7 // ANEXOS

7.1 // CLONE DO GITHUB

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

HTTPS://GITHUB.COM/DXCCCII/DISTRIBUITEDSYSTEMSTP2

7.2 // CÓDIGO DO CLIENTE

```
using System; // Importing System namespace for basic functionalities
using System.Collections.Generic; // Importing System.Collections.Generic namespace for using collections like Dictionary
using System.Text; // Importing System.Text namespace for text encoding
using RabbitMQ.Client; // Importing RabbitMQ.Client namespace for RabbitMQ functionalities
using RabbitMQ.Client.Events; // Importing RabbitMQ.Client.Events namespace for RabbitMQ event-based consumer

class Cliente // Declaring a class named Cliente
{
private static IConnection rabbitConnection; // Declaring a static variable for RabbitMQ connection
private static IModel rabbitChannel; // Declaring a static variable for RabbitMQ channel
private static string replyQueueName; // Declaring a static variable for reply queue name
private static EventingBasicConsumer consumer; // Declaring a static variable for RabbitMQ consumer
private static string correlationId; // Declaring a static variable for correlation ID
private static IBasicProperties props; // Declaring a static variable for basic properties of RabbitMQ messages

private static Dictionary<string, string> subscribedServices = new Dictionary<string, string>(); // Declaring a static dictionary to store subscribed services
private static IConnection notificationConnection; // Declaring a static variable for notification connection
private static IModel notificationChannel; // Declaring a static variable for notification channel

static void Main(string[] args) // Main method, entry point of the program
{
Console.WriteLine("Bem-vindo à ServiMoto!"); // Printing welcome message
Console.WriteLine($"\n"); // Printing a new line
Console.Write("Por favor, insira o endereço IP do servidor: "); // Prompting the user to enter the server IP address
string enderecoServidor = Console.ReadLine(); // Reading the server IP address from the user input

try
{
InitRabbitMQ(enderecoServidor); // Initializing RabbitMQ with the server IP address

Console.WriteLine("Conectado ao servidor. Aguardando resposta..."); // Printing a message indicating connection to the server

string response = Call("CONNECT"); // Sending "CONNECT" message to the server and waiting for a response
Console.WriteLine("Resposta do servidor: " + response); // Printing the server's response

if (response == "100 OK") // Checking if the response is "100 OK"
{
Console.WriteLine($"\n"); // Printing a new line
Console.Write("Por favor, insira o seu ID: "); // Prompting the user to enter their ID
string idCliente = Console.ReadLine(); // Reading the client ID from the user input
response = Call($"CLIENT_ID:{idCliente}"); // Sending "CLIENT_ID" message to the server and waiting for a response
Console.WriteLine("Resposta: " + response); // Printing the server's response

if (response.StartsWith("ID_CONFIRMED")) // Checking if the response starts with "ID_CONFIRMED"
{
Console.WriteLine($"\n"); // Printing a new line
Console.Write("Por favor, insira a sua password: "); // Prompting the user to enter their password
string senha = Console.ReadLine(); // Reading the password from the user input
response = Call($"PASSWORD:{idCliente},{senha}"); // Sending "PASSWORD" message to the server and waiting for a response
Console.WriteLine("Resposta: " + response); // Printing the server's response

if (response == "PASSWORD_CONFIRMED") // Checking if the response is "PASSWORD_CONFIRMED"
{
InitNotificationListener(); // Initializing the notification listener
StartNotificationListener(idCliente); // Starting the notification listener with the client ID

if (idCliente.StartsWith("Adm")) // Checking if the client ID starts with "Adm"
{
while (true)
{
Console.WriteLine($"\n"); // Printing a new line
Console.Write("Por favor, insira o ID do serviço que quer gerir (e.g., Servico_X): "); // Prompting the user to enter the service ID to manage
string adminServiceId = Console.ReadLine(); // Reading the admin service ID from the user input
response = Call($"ADMIN_SERVICE_ID:{adminServiceId}"); // Sending "ADMIN_SERVICE_ID" message to the server and waiting for a response
Console.WriteLine("Resposta do servidor: " + response); // Printing the server's response

if (response == "SERVICE_CONFIRMED") // Checking if the response is "SERVICE_CONFIRMED"
{
AdministradorMenu(idCliente, adminServiceId); // Calling the administrator menu with the client ID and admin service ID
break; // Breaking the loop
}
else if (response == "SERVICE_NOT_FOUND") // Checking if the response is "SERVICE_NOT_FOUND"
{
Console.WriteLine(response); // Printing the response
}
else
{
Console.WriteLine("Resposta do servidor desconhecida: " + response); // Printing an unknown server response
break; // Breaking the loop
}
}
}
else
{
ClienteMenu(idCliente); // Calling the client menu with the client ID
}
}
else
{
Console.WriteLine("Autenticação falhou."); // Printing authentication failed message
}
}
}
```

```
Console.WriteLine("Comunicação com o servidor encerrada."); // Printing communication with server closed message
}
catch (Exception ex) // Catching any exceptions
{
Console.WriteLine("Ocorreu um erro: " + ex.Message); // Printing the error message
}
finally
{
rabbitChannel.Close(); // Closing the RabbitMQ channel
rabbitConnection.Close(); // Closing the RabbitMQ connection
Environment.Exit(0); // Exiting the application
}
}

private static void InitRabbitMQ(string enderecoServidor)
{
var factory = new ConnectionFactory() { HostName = enderecoServidor }; // Creating a connection factory with the server IP address
rabbitConnection = factory.CreateConnection(); // Creating a RabbitMQ connection
rabbitChannel = rabbitConnection.CreateModel(); // Creating a RabbitMQ channel
replyQueueName = rabbitChannel.QueueDeclare().QueueName; // Declaring a reply queue and getting its name
consumer = new EventingBasicConsumer(rabbitChannel); // Creating a new EventingBasicConsumer for the channel
correlationId = Guid.NewGuid().ToString(); // Generating a new unique correlation ID
props = rabbitChannel.CreateBasicProperties(); // Creating basic properties for the RabbitMQ message
props.CorrelationId = correlationId; // Setting the correlation ID in the properties
props.ReplyTo = replyQueueName; // Setting the reply queue name in the properties
}

private static string Call(string message)
{
var messageBytes = Encoding.UTF8.GetBytes(message); // Converting the message to a byte array
rabbitChannel.BasicPublish(exchange: "", routingKey: "rpc_queue", basicProperties: props, body: messageBytes); // Publishing the message to the RabbitMQ queue

var responseReceived = false; // Flag to check if the response is received
string response = null; // Variable to store the response

consumer.Received += (model, ea) => // Event handler for received messages
{
if (ea.BasicProperties.CorrelationId == correlationId) // Checking if the correlation ID matches
{
response = Encoding.UTF8.GetString(ea.Body.ToArray()); // Decoding the response body
responseReceived = true; // Setting the flag to true
}
};

rabbitChannel.BasicConsume(queue: replyQueueName, autoAck: true, consumer: consumer); // Consuming messages from the reply queue

while (!responseReceived) // Waiting for the response
{
// Busy-wait for the response
}

return response; // Returning the response
}

private static void ClienteMenu(string idCliente)
{
while (true)
{
Console.WriteLine("\n");
Console.WriteLine("1. Solicitar tarefa");
Console.WriteLine("2. Marcar tarefa como concluída");
Console.WriteLine("3. Subscrever a um servico");
Console.WriteLine("4. Cancelar a subscricao as notificacoes de um servico");
Console.WriteLine("5. Sair");
Console.Write("Escolha uma opção: ");
string opcao = Console.ReadLine();

string response;

switch (opcao)
{
case "1":
response = Call($"REQUEST_TASK|{idCliente}"); // Sending a request task message to the server
Console.WriteLine("Resposta do servidor: " + response); // Printing the server's response
break;

case "2":
Console.Write("Por favor, insira a descrição da tarefa concluída: "); // Prompting the user to enter the task description
string descricaoTarefa = Console.ReadLine(); // Reading the task description from the user input
response = Call($"TASK_COMPLETED|{idCliente}|{descricaoTarefa}"); // Sending a task completed message to the server
Console.WriteLine("Resposta do servidor: " + response); // Printing the server's response
break;

case "3":
Console.Write("Por favor, insira o ID do serviço para subscrever: "); // Prompting the user to enter the service ID to subscribe
string serviceIdSubscribe = Console.ReadLine(); // Reading the service ID from the user input
response = Call($"SUBSCRIBE|{idCliente}|{serviceIdSubscribe}"); // Sending a subscribe message to the server
if (response == "SUBSCRIBED") // Checking if the response is "SUBSCRIBED"
{
subscribedServices[serviceIdSubscribe] = $"{idCliente}_{serviceIdSubscribe}"; // Adding the service to subscribed services
BindNotificationQueue(idCliente, serviceIdSubscribe); // Binding the notification queue
Console.WriteLine($"Subscribed to service: {serviceIdSubscribe}"); // Printing the subscription message
}
Console.WriteLine("Resposta do servidor: " + response); // Printing the server's response
break;
```

```
case "4":
Console.Write("Por favor, insira o ID do serviço para cancelar a subscricao: "); // Prompting the user to enter the service ID to unsubscribe
string serviceIdUnsubscribe = Console.ReadLine(); // Reading the service ID from the user input
response = Call($"UNSUBSCRIBE|{idCliente}|{serviceIdUnsubscribe}"); // Sending an unsubscribe message to the server
if (response == "UNSUBSCRIBED") // Checking if the response is "UNSUBSCRIBED"
{
UnbindNotificationQueue(serviceIdUnsubscribe); // Unbinding the notification queue
subscribedServices.Remove(serviceIdUnsubscribe); // Removing the service from subscribed services
Console.WriteLine($"Unsubscribed from service: {serviceIdUnsubscribe}"); // Printing the unsubscription message
}
Console.WriteLine("Resposta do servidor: " + response); // Printing the server's response
break;

case "5":
return; // Returning from the method to exit the menu

default:
Console.WriteLine("Opção inválida. Por favor, tente novamente."); // Printing invalid option message
break;
}
}
}

private static void AdministradorMenu(string idCliente, string adminServiceId)
{
while (true)
{
Console.WriteLine("\n");
Console.WriteLine("1. Criar nova tarefa");
Console.WriteLine("2. Consultar tarefas");
Console.WriteLine("3. Alterar status da tarefa");
Console.WriteLine("4. Sair");
Console.Write("Escolha uma opção: ");
string opcao = Console.ReadLine();

string response;

switch (opcao)
{
case "1":
Console.Write("Por favor, insira a descrição da nova tarefa com o formato SX_TNUMERO DA TAREFA,DESCRICAO DA TAREFA: "); // Prompting the user to enter the task description
string descricaoTarefa = Console.ReadLine(); // Reading the task description from the user input
response = Call($"ADD_TASK|{adminServiceId}|{descricaoTarefa}"); // Sending an add task message to the server
Console.WriteLine("Resposta do servidor: " + response); // Printing the server's response
break;

case "2":
response = Call($"CONSULT_TASKS|{adminServiceId}"); // Sending a consult tasks message to the server
Console.WriteLine($"\nTarefas no {adminServiceId}:\n{response}"); // Printing the server's response
break;

case "3":
Console.Write("Por favor, insira a descrição da tarefa a ser alterada: "); // Prompting the user to enter the task description
string taskDescription = Console.ReadLine(); // Reading the task description from the user input

Console.Write("Insira o novo status da tarefa: "); // Prompting the user to enter the new status
string newStatus = Console.ReadLine(); // Reading the new status from the user input

Console.Write("Insira o campo adicional: "); // Prompting the user to enter the additional field
string additionalField = Console.ReadLine(); // Reading the additional field from the user input

response = Call($"CHANGE_TASK_STATUS|{adminServiceId}|{taskDescription},{newStatus},{additionalField}"); // Sending a change task status message to the server
Console.WriteLine("Resposta do servidor: " + response); // Printing the server's response
break;

case "4":
return; // Returning from the method to exit the menu

default:
Console.WriteLine("Opção inválida. Por favor, tente novamente."); // Printing invalid option message
break;
}
}
}

private static void InitNotificationListener()
{
var factory = new ConnectionFactory() { HostName = "localhost" }; // Creating a connection factory with the host name "localhost"
notificationConnection = factory.CreateConnection(); // Creating a RabbitMQ connection for notifications
notificationChannel = notificationConnection.CreateModel(); // Creating a RabbitMQ channel for notifications
notificationChannel.ExchangeDeclare(exchange: "service_notifications", type: ExchangeType.Topic); // Declaring an exchange for service notifications
Console.WriteLine("Notification listener initialized."); // Printing notification listener initialized message
}

private static void StartNotificationListener(string clientId)
{
foreach (var serviceId in subscribedServices.Keys) // Iterating through subscribed services
{
BindNotificationQueue(clientId, serviceId); // Binding the notification queue for each service
}

Console.WriteLine("Client is waiting for notifications..."); // Printing waiting for notifications message
}

private static void BindNotificationQueue(string clientId, string serviceId)
{
```

```
var queueName = $"{clientId}_{serviceId}"; // Constructing the queue name
notificationChannel.QueueDeclare(queue: queueName, durable: true, exclusive: false, autoDelete: false, arguments: null); // Declaring the queue
notificationChannel.QueueBind(queue: queueName, exchange: "service_notifications", routingKey: $"NOTIFICATION.{serviceId}"); // Binding the queue to the exchange

var consumer = new EventingBasicConsumer(notificationChannel); // Creating a new EventingBasicConsumer for notifications
consumer.Received += (model, ea) => // Event handler for received notifications
{
var body = ea.Body.ToArray(); // Getting the body of the message
var message = Encoding.UTF8.GetString(body); // Decoding the message
if (!message.StartsWith("UNSUBSCRIBE:")) // Checking if the message does not start with "UNSUBSCRIBE:"
{
Console.WriteLine($"\n");
Console.WriteLine($"\nReceived notification for {serviceId}: {message}"); // Printing the received notification
}
else
{
var parts = message.Split('-'); // Splitting the message by "-"
if (parts.Length == 3 && parts[1] == clientId && parts[2] == serviceId) // Checking if the message format matches
{
notificationChannel.QueueUnbind(queue: queueName, exchange: "service_notifications", routingKey: $"NOTIFICATION.{serviceId}"); // Unbinding the queue from the exchange
Console.WriteLine($"Unbound from notifications for service: {serviceId}"); // Printing the unbound message
}
}
};

notificationChannel.BasicConsume(queue: queueName, autoAck: true, consumer: consumer); // Consuming messages from the queue
Console.WriteLine($"Subscribed to notifications for service: {serviceId}"); // Printing the subscribed message
}

private static void UnbindNotificationQueue(string serviceId)
{
if (subscribedServices.ContainsKey(serviceId)) // Checking if the service is in the subscribed services
{
var queueName = subscribedServices[serviceId]; // Getting the queue name
notificationChannel.QueueUnbind(queue: queueName, exchange: "service_notifications", routingKey: $"NOTIFICATION.{serviceId}"); // Unbinding the queue from the exchange
Console.WriteLine($"Unbound from notifications for service: {serviceId}"); // Printing the unbound message
}
}
}
```

7.3 // CÓDIGO DO SERVIDOR

```
using System; // Import the System namespace.
using System.Collections.Generic; // Import the System.Collections.Generic namespace.
using System.IO; // Import the System.IO namespace.
using System.Linq; // Import the System.Linq namespace.
using System.Text; // Import the System.Text namespace.
using System.Threading; // Import the System.Threading namespace.
using RabbitMQ.Client; // Import the RabbitMQ.Client namespace.
using RabbitMQ.Client.Events; // Import the RabbitMQ.Client.Events namespace.

class Servidor // Define a class named Servidor.
{
public static Dictionary<string, (string ServiceId, string Password)> serviceDict = new Dictionary<string, (string ServiceId, string Password)>(); // Dictionary to store servi
public static Dictionary<string, List<string>> taskDict = new Dictionary<string, List<string>>(); // Dictionary to store tasks for each service.
private static Mutex mutex = new Mutex(); // Mutex for thread safety.

private static IConnection rabbitConnection; // Declare a variable for RabbitMQ connection.
private static IModel rabbitChannel; // Declare a variable for RabbitMQ channel.
private static string rpcQueueName = "rpc_queue"; // Define the RPC queue name.

static void Main(string[] args) // Main method, entry point of the application.
{
InitRabbitMQ(); // Initialize RabbitMQ connection and channel.
PrintWorkingDirectory(); // Print the current working directory.
LoadServiceAllocationsFromCSV(); // Load service allocations from a CSV file.
LoadDataFromCSVForAllServices(); // Load tasks data from CSV files for all services.

var consumer = new EventingBasicConsumer(rabbitChannel); // Create a new consumer for RabbitMQ.
consumer.Received += (model, ea) => // Define the event handler for received messages.
{
var body = ea.Body.ToArray(); // Get the body of the message.
var props = ea.BasicProperties; // Get the properties of the message.
var replyProps = rabbitChannel.CreateBasicProperties(); // Create properties for the reply message.
replyProps.CorrelationId = props.CorrelationId; // Set the correlation ID for the reply.

string response = null; // Initialize response to null.
try
{
var message = Encoding.UTF8.GetString(body); // Convert message body to string.
Console.WriteLine($"Received message: {message}"); // Print the received message.
response = HandleRequest(message); // Handle the request and get the response.
}
catch (Exception ex) // Catch any exceptions.
{
Console.WriteLine($"Error handling request: {ex.Message}"); // Print the error message.
response = "500 INTERNAL SERVER ERROR"; // Set response to internal server error.
}
finally
{
var responseBytes = Encoding.UTF8.GetBytes(response); // Convert response to bytes.
rabbitChannel.BasicPublish(exchange: "", routingKey: props.ReplyTo, basicProperties: replyProps, body: responseBytes); // Publish the response.
```

```
rabbitChannel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false); // Acknowledge the message.
}
};

rabbitChannel.BasicConsume(queue: rpcQueueName, autoAck: false, consumer: consumer); // Start consuming messages from the RPC queue.
Console.WriteLine("RPC Server is running. Waiting for requests..."); // Print server running message.
Console.ReadLine(); // Wait for user input to keep the server running.
}

private static void InitRabbitMQ() // Method to initialize RabbitMQ.
{
var factory = new ConnectionFactory() { HostName = "localhost" }; // Create a connection factory with the hostname.
rabbitConnection = factory.CreateConnection(); // Create a connection to RabbitMQ.
rabbitChannel = rabbitConnection.CreateModel(); // Create a channel.
rabbitChannel.QueueDeclare(queue: rpcQueueName, durable: false, exclusive: false, autoDelete: false, arguments: null); // Declare the RPC queue.
rabbitChannel.ExchangeDeclare(exchange: "service_notifications", type: ExchangeType.Topic); // Declare the exchange for notifications.
Console.WriteLine("RabbitMQ Initialized."); // Print RabbitMQ initialized message.
}

private static string HandleRequest(string message) // Method to handle incoming requests.
{
string response = null; // Initialize response to null.
if (message.StartsWith("CONNECT")) // Check if message is CONNECT.
{
response = "100 OK"; // Set response to OK.
}
else if (message.StartsWith("CLIENT_ID:")) // Check if message contains client ID.
{
string clientId = message.Substring("CLIENT_ID:".Length).Trim(); // Extract client ID from the message.
response = $"ID_CONFIRMED:{clientId}"; // Set response to ID confirmed.
}
else if (message.StartsWith("PASSWORD:")) // Check if message contains password.
{
string[] parts = message.Substring("PASSWORD:".Length).Trim().Split(','); // Split the message to get client ID and password.
string clientId = parts[0].Trim(); // Extract client ID.
string password = parts[1].Trim(); // Extract password.

if (serviceDict.ContainsKey(clientId) && serviceDict[clientId].Password == password) // Check if client ID and password match.
{
response = "PASSWORD_CONFIRMED"; // Set response to password confirmed.
}
else
{
response = "403 FORBIDDEN"; // Set response to forbidden.
}
}
else if (message.StartsWith("ADMIN_SERVICE_ID:")) // Check if message contains admin service ID.
{
string serviceId = message.Substring("ADMIN_SERVICE_ID:".Length).Trim(); // Extract service ID.

if (!serviceId.StartsWith("Servico_")) // Check if service ID is valid.
{
response = "500 BAD REQUEST"; // Set response to bad request.
}
else
{
string serviceFilePath = Path.Combine(serviceId + ".csv"); // Create file path for the service.
response = File.Exists(serviceFilePath) ? "SERVICE_CONFIRMED" : "SERVICE_NOT_FOUND"; // Check if file exists and set response accordingly.
}
}
else
{
string[] parts = message.Split('|'); // Split the message to get command and data.
string command = parts[0]; // Extract command.
string clientId = parts[1]; // Extract client ID.
string data = parts.Length > 2 ? parts[2] : null; // Extract data if present.

switch (command) // Switch based on the command.
{
case "ADD_TASK":
response = AddTask(clientId, data); // Call AddTask method and get the response.
break;
case "CONSULT_TASKS":
response = ConsultTasks(clientId); // Call ConsultTasks method and get the response.
break;
case "CHANGE_TASK_STATUS":
var statusParts = data.Split(','); // Split data to get task details.
response = ChangeTaskStatus(clientId, statusParts[0], statusParts[1], statusParts[2]); // Call ChangeTaskStatus method and get the response.
break;
case "REQUEST_TASK":
response = AllocateTask(clientId); // Call AllocateTask method and get the response.
break;
case "TASK_COMPLETED":
response = MarkTaskAsCompleted(clientId, data); // Call MarkTaskAsCompleted method and get the response.
break;
case "SUBSCRIBE":
SubscribeToService(clientId, data); // Call SubscribeToService method.
response = "SUBSCRIBED"; // Set response to subscribed.
break;
case "UNSUBSCRIBE":
UnsubscribeFromService(clientId, data); // Call UnsubscribeFromService method.
response = "UNSUBSCRIBED"; // Set response to unsubscribed.
break;
default:
response = "500 BAD REQUEST"; // Set response to bad request for unknown commands.
break;
```

```
}
}
return response; // Return the response.
}

private static void LoadServiceAllocationsFromCSV() // Method to load service allocations from a CSV file.
{
string csvFilePath = Path.Combine(Directory.GetCurrentDirectory(), "service_allocations.csv"); // Get the file path.

try
{
var lines = File.ReadAllLines(csvFilePath); // Read all lines from the CSV file.
foreach (var line in lines.Skip(1)) // Iterate through the lines, skipping the header.
{
var parts = line.Split(','); // Split each line to get parts.

if (parts.Length >= 3) // Check if there are enough parts.
{
string clientId = parts[0].Trim(); // Extract client ID.
string serviceId = parts[1].Trim(); // Extract service ID.
string password = parts[2].Trim(); // Extract password.

serviceDict[clientId] = (serviceId, password); // Add client ID, service ID, and password to the dictionary.
}
}
}
catch (Exception ex) // Catch any exceptions.
{
Console.WriteLine($"Error loading data from CSV file {csvFilePath}: {ex.Message}"); // Print error message.
}
}

private static string AddTask(string serviceId, string taskDescription) // Method to add a task.
{
string serviceFilePath = serviceId + ".csv"; // Create file path for the service.
try
{
string newTask = $"{taskDescription},nao alocada,"; // Create a new task string.
File.AppendAllLines(serviceFilePath, new string[] { newTask }); // Append the new task to the file.
PublishNotification($"\nTASK_ADDED:{serviceId}: {taskDescription}", isAdminChange: true); // Publish a notification for the new task.
return "201 CREATED"; // Return created response.
}
catch (Exception ex) // Catch any exceptions.
{
Console.WriteLine($"Error adding task: {ex.Message}"); // Print error message.
return "500 INTERNAL SERVER ERROR"; // Return internal server error response.
}
}

private static string ConsultTasks(string serviceFilePath) // Method to consult tasks.
{
serviceFilePath = serviceFilePath + ".csv"; // Create file path for the service.
try
{
string[] tasks = File.ReadAllLines(serviceFilePath); // Read all tasks from the file.
StringBuilder response = new StringBuilder(); // Create a StringBuilder for the response.
foreach (string task in tasks) // Iterate through the tasks.
{
response.AppendLine(task); // Append each task to the response.
}
response.AppendLine("END"); // Append end to the response.
return response.ToString(); // Return the response as a string.
}
catch (Exception ex) // Catch any exceptions.
{
Console.WriteLine($"Error consulting tasks: {ex.Message}"); // Print error message.
return "500 Internal Server Error"; // Return internal server error response.
}
}

private static string ChangeTaskStatus(string serviceId, string taskDescription, string newStatus, string additionalField) // Method to change the status of a task.
{
string serviceFilePath = serviceId + ".csv"; // Create file path for the service.
try
{
string[] lines = File.ReadAllLines(serviceFilePath); // Read all lines from the file.
bool taskFound = false; // Initialize task found flag.

for (int i = 0; i < lines.Length; i++) // Iterate through the lines.
{
string line = lines[i]; // Get the line.
string[] parts = line.Split(','); // Split the line to get parts.

if (parts.Length >= 3 && parts[1].Trim() == taskDescription) // Check if the line matches the task description.
{
if (!IsValidStatus(newStatus)) // Check if the new status is valid.
{
return "500 BAD REQUEST - Invalid newStatus"; // Return bad request response.
}

if (newStatus.ToLower() == "nao alocada") // Check if the new status is nao alocada.
{
additionalField = ""; // Set additional field to empty.
}
else if (!string.IsNullOrEmpty(additionalField) && !additionalField.StartsWith("Cl_")) // Check if the additional field is valid.
{
```

```
return "500 BAD REQUEST - Additional field must start with 'Cl_'"; // Return bad request response.
}

parts[2] = newStatus; // Set the new status.
if (parts.Length == 3) // Check if there are only 3 parts.
{
line = string.Join(",", parts[0], parts[1], parts[2], additionalField); // Create a new line with the additional field.
}
else
{
parts[3] = additionalField; // Set the additional field.
line = string.Join(",", parts); // Create a new line.
}

lines[i] = line; // Update the line.
taskFound = true; // Set task found flag to true.
break; // Break the loop.
}
}

if (taskFound) // Check if the task was found.
{
File.WriteAllLines(serviceFilePath, lines); // Write the lines to the file.
string notificationMessage = $"\nTASK_STATUS_CHANGED:{serviceId}:{taskDescription}:{newStatus}"; // Create a notification message.
PublishNotification(notificationMessage, isAdminChange: true); // Publish the notification.
Console.WriteLine($"Published notification: {notificationMessage}"); // Print the notification message.
return "200 OK"; // Return OK response.
}
else
{
return "404 NOT FOUND - Task not found"; // Return not found response.
}
}
catch (IOException) // Catch IO exceptions.
{
return "500 INTERNAL SERVER ERROR - IOException"; // Return internal server error response.
}
catch (Exception ex) // Catch any exceptions.
{
Console.WriteLine($"Error changing task status: {ex.Message}"); // Print error message.
return "500 INTERNAL SERVER ERROR"; // Return internal server error response.
}
}

private static bool IsValidStatus(string status) // Method to check if a status is valid.
{
string[] validStatuses = { "Nao alocada", "Concluido", "Em curso" }; // Define valid statuses.
return validStatuses.Contains(status); // Check if the status is valid.
}

private static void PrintWorkingDirectory() // Method to print the current working directory.
{
string currentDirectory = Directory.GetCurrentDirectory(); // Get the current working directory.
Console.WriteLine("Current working directory: " + currentDirectory); // Print the current working directory.
}

private static void LoadDataFromCSVForAllServices() // Method to load data from CSV files for all services.
{
string servicesFilePath = Directory.GetCurrentDirectory(); // Get the current working directory.
try
{
foreach (var serviceFile in Directory.GetFiles(servicesFilePath, "*.csv")) // Get all CSV files in the directory.
{
var serviceLines = File.ReadAllLines(serviceFile); // Read all lines from the file.
string serviceId = Path.GetFileNameWithoutExtension(serviceFile); // Get the service ID from the file name.

if (!taskDict.ContainsKey(serviceId)) // Check if the task dictionary contains the service ID.
{
taskDict[serviceId] = new List<string>(); // Add the service ID to the task dictionary.
}

for (int i = 1; i < serviceLines.Length; i++) // Iterate through the lines, skipping the header.
{
var line = serviceLines[i]; // Get the line.
var parts = line.Split(','); // Split the line to get parts.

if (parts.Length == 3) // Check if there are only 3 parts.
{
line = $"{parts[0].Trim()},{parts[1].Trim()},{parts[2].Trim()},"; // Create a new line with an empty field.
}
else if (parts.Length < 4) // Check if there are less than 4 parts.
{
line = $"{line.Trim()},"; // Add an empty field to the line.
}

taskDict[serviceId].Add(line.Trim()); // Add the line to the task dictionary.
}
}
}
catch (Exception ex) // Catch any exceptions.
{
Console.WriteLine($"Error loading data from CSV file {servicesFilePath}: {ex.Message}"); // Print error message.
}
}

public static string AllocateTask(string clientId) // Method to allocate a task.
```

```
{
mutex.WaitOne(); // Acquire the mutex.
try
{
if (!serviceDict.ContainsKey(clientId)) // Check if the service dictionary contains the client ID.
{
return "ERROR: Service not found for client"; // Return error response.
}

string serviceId = serviceDict[clientId].ServiceId; // Get the service ID for the client.

if (taskDict.ContainsKey(serviceId)) // Check if the task dictionary contains the service ID.
{
var unallocatedTask = taskDict[serviceId].FirstOrDefault(task => task.Split(',')[2].Trim().ToLower() == "nao alocada"); // Get the first unallocated task.

if (unallocatedTask != null) // Check if there is an unallocated task.
{
var taskParts = unallocatedTask.Split(','); // Split the task to get parts.

if (taskParts.Length != 4) // Check if there are not exactly 4 parts.
{
return "500 INTERNAL SERVER ERROR"; // Return internal server error response.
}

taskParts[2] = "Em curso"; // Set the status to "Em curso".
taskParts[3] = clientId; // Set the client ID.

string updatedTask = string.Join(",", taskParts); // Create an updated task string.
int taskIndex = taskDict[serviceId].IndexOf(unallocatedTask); // Get the index of the unallocated task.
taskDict[serviceId][taskIndex] = updatedTask; // Update the task dictionary.

string serviceFilePath = Path.Combine(Directory.GetCurrentDirectory(), $"{serviceId}.csv"); // Create file path for the service.
File.WriteAllLines(serviceFilePath, taskDict[serviceId]); // Write the tasks to the file.

string message = $"TASK_ALLOCATED:{taskParts[1]}"; // Create an allocation message.

return message; // Return the allocation message.
}
else
{
return "NO_TASK_AVAILABLE"; // Return no task available response.
}
}
else
{
return "NO_TASK_AVAILABLE"; // Return no task available response.
}
}
finally
{
mutex.ReleaseMutex(); // Release the mutex.
}
}

public static string MarkTaskAsCompleted(string clientId, string taskDescription) // Method to mark a task as completed.
{
mutex.WaitOne(); // Acquire the mutex.
try
{
if (!serviceDict.ContainsKey(clientId)) // Check if the service dictionary contains the client ID.
{
return "ERROR: Service not found for client"; // Return error response.
}

string serviceId = serviceDict[clientId].ServiceId; // Get the service ID for the client.

if (taskDict.ContainsKey(serviceId)) // Check if the task dictionary contains the service ID.
{
var taskIndex = taskDict[serviceId].FindIndex(task => task.Split(',')[1].Trim() == taskDescription); // Get the index of the task.

if (taskIndex != -1) // Check if the task was found.
{
var taskParts = taskDict[serviceId][taskIndex].Split(','); // Split the task to get parts.

if (taskParts.Length != 4) // Check if there are not exactly 4 parts.
{
return $"ERROR: Incorrect task format for task: {taskDict[serviceId][taskIndex]}"; // Return error response.
}

taskParts[2] = "Concluido"; // Set the status to "Concluido".
taskParts[3] = clientId; // Set the client ID.

string updatedTask = string.Join(",", taskParts); // Create an updated task string.
taskDict[serviceId][taskIndex] = updatedTask; // Update the task dictionary.

string serviceFilePath = Path.Combine(Directory.GetCurrentDirectory(), $"{serviceId}.csv"); // Create file path for the service.
File.WriteAllLines(serviceFilePath, taskDict[serviceId]); // Write the tasks to the file.

string notificationMessage = $"TASK_COMPLETED:{clientId}: {taskDescription}"; // Create a notification message.
PublishNotification(notificationMessage, isAdminChange: false); // Publish the notification.
return $"TASK_MARKED_AS_COMPLETED:{taskDescription}"; // Return the completion message.
}
else
{
return $"ERROR_TASK_NOT_FOUND:{taskDescription}"; // Return task not found response.
}
```

```
}
else
{
return $"ERROR_SERVICE_NOT_FOUND:{serviceId}"; // Return service not found response.
}
}
catch (Exception)
{
return "500 INTERNAL SERVER ERROR"; // Return internal server error response.
}
finally
{
mutex.ReleaseMutex(); // Release the mutex.
}
}

private static void PublishNotification(string message, bool isAdminChange) // Method to publish a notification.
{
var body = Encoding.UTF8.GetBytes($"{message}"); // Convert the message to bytes.
string exchange = "service_notifications"; // Define the exchange name.
string routingKey = $"NOTIFICATION.{message.Split(':')[1]}"; // Define the routing key using the service ID.
rabbitChannel.BasicPublish(exchange: exchange, routingKey: routingKey, basicProperties: null, body: body); // Publish the message.
Console.WriteLine($"Sent notification: {message} with routing key {routingKey}"); // Print the notification message.
}

private static void PublishUnsubscribeNotification(string clientId, string serviceId) // Method to publish an unsubscribe notification.
{
var body = Encoding.UTF8.GetBytes($"\nUNSUBSCRIBE:{clientId}:{serviceId}"); // Convert the unsubscribe message to bytes.
string exchange = "service_notifications"; // Define the exchange name.
string routingKey = $"NOTIFICATION.{serviceId}"; // Define the routing key using the service ID.
rabbitChannel.BasicPublish(exchange: exchange, routingKey: routingKey, basicProperties: null, body: body); // Publish the unsubscribe message.
Console.WriteLine($"Sent unsubscription notification for client {clientId} from service {serviceId}"); // Print the unsubscribe notification.
}

public static Dictionary<string, List<string>> subscriptions = new Dictionary<string, List<string>>(); // Dictionary to store subscriptions.

private static void SubscribeToService(string clientId, string serviceId) // Method to subscribe to a service.
{
if (!subscriptions.ContainsKey(serviceId)) // Check if the subscriptions dictionary contains the service ID.
{
subscriptions[serviceId] = new List<string>(); // Add the service ID to the subscriptions dictionary.
}
if (!subscriptions[serviceId].Contains(clientId)) // Check if the client ID is already subscribed.
{
subscriptions[serviceId].Add(clientId); // Add the client ID to the subscriptions.
}
Console.WriteLine($"Client {clientId} subscribed to {serviceId}"); // Print the subscription message.
}

private static void UnsubscribeFromService(string clientId, string serviceId) // Method to unsubscribe from a service.
{
if (subscriptions.ContainsKey(serviceId)) // Check if the subscriptions dictionary contains the service ID.
{
subscriptions[serviceId].Remove(clientId); // Remove the client ID from the subscriptions.
}
Console.WriteLine($"Client {clientId} unsubscribed from {serviceId}"); // Print the unsubscription message.
}
}
```