Desenvolvimento de Sistemas de Informação Distribuídos (ACH2147)

Infra-estrutura para Suporte e Agentes Móveis

Maria Eduarda Rodrigues Garcia 11796621
Mirela Mei 11208392

Julho 2023

1 Resumo

O objetivo do trabalho consiste em realizar uma arquitetura cliente-servidor que utilize agentes móveis. Os agentes móveis podem viajar pela rede, circulando pelas agências a fim de executar suas tarefas onde for mais conveniente - caso sejam necessários recursos de outras máquinas, por exemplo, não há comunicação com estas, mas uma migração para o servidor que pode oferecê-los. Existe um serviço de nomes, que permite com que as agências consigam se comunicar entre si e acessar quais agentes pertencem a cada agência, bem como em quais máquinas estas estão localizadas. Neste sistema, os agentes podem enviar mensagens entre si, mesmo que estejam em agências diferentes, podem migrar para uma agência que não seja a sua e pode deixar de realizar operações (ser removido de sua agência e retirado do serviço de nomes).

```
public interface Agency extends Remote{
    public String addAgent(Agent agent) throws RemoteException;
    public boolean moveAgent(String agentID, String newAgencyName) throws RemoteException ;
    public boolean destroyAgent(String agentID) throws RemoteException;
    public String sendMessageToAgent(String originAgentID, String destinationAgentID, String r
    public Map<String, LinkedBlockingQueue<Message>> getReceivedMessages();
    public String receiveMessage(Message message);
    public void answerMessage(Message message);
    public String getID() throws RemoteException;
    public String getName() throws RemoteException;
    public Agent getAgentByID(String agentID) throws RemoteException;
    public String getAddress();
    public LinkedList<Agent> getAgentsList() throws RemoteException;
}
```

Figura 1: Esquematização da infraestrutura para suporte e agentes móveis.

2 Estrutura do projeto

2.1 Estrutura de classes

O projeto foi dividido em duas principais pastas:

- bin: pasta com arquivos .class
- src: pasta com arquivos .java

Dentro da pasta `src`, as classes foram divididas em quatro subpastas: `Agency`, `Agent`, `NamingService` e `User`.

Dentro de `Agency`, tem-se as classes:

- **Agency:** A classe `Agency` é uma interface remota que define os métodos que uma agência deve implementar. Algumas das principais operações são adicionar um agente, mover um agente para outra agência, enviar mensagens entre agentes e destruir um agente. Os métodos declarados nessa interface são:

```
public interface Agency extends Remote{
    public void addAgent(Agent agent) throws RemoteException;
    public void moveAgent(Agent agent, String destinationAgencyName) throws RemoteException;
    public void removeAgent(Agent agent) throws RemoteException;
    public String getID() throws RemoteException;
    public String getName() throws RemoteException;
    public List<Agent> getAgentsList() throws RemoteException;
    public String generateUniqueCode() throws RemoteException;
}
```

Figura 2: Captura de tela do código.

  - `addAgent`: adiciona um objeto `Agent` à agência em questão.
  - `moveAgent`: redireciona um objeto `Agent` a uma outra agência, descrita pelos parâmetros enviados à função.
  - `destroyAgent`: remove o agente da agência em questão.
  - `sendMessageToAgent`: envia mensagem de agente remetente para agente destinatário com seu conteúdo.
  - `getReceivedMessages`: retorna lista de mensagens recebidas por agente.
  - `receiveMessage`: recebe um objeto Message como parâmetro, representando a mensagem enviada por um agente para outro. Quando um agente envia uma mensagem para outro, a agência de destino recebe a mensagem através deste método. A mensagem é colocada na fila de mensagens recebidas do agente de destino, aguardando para ser processada. Em seguida, o método aguarda até que o agente de destino envie uma resposta de volta. Quando a resposta é recebida, o método verifica se ela corresponde à mensagem original (usando o ID da mensagem) e, em seguida, retorna a resposta ao agente de origem.

- `answerMessage`: recebe um objeto Message como parâmetro, representando a resposta a uma mensagem previamente enviada por um agente. Quando um agente recebe uma mensagem de outro agente, ele processa a mensagem e prepara uma resposta. O agente, então, chama o método answerMessage da agência para enviar a resposta. A agência coloca a resposta na fila de mensagens respondidas do agente de destino. Enquanto o método receiveMessage espera pela resposta, o agente de destino acessa a fila de mensagens respondidas e obtém a resposta correspondente à mensagem original. O agente de destino, então, pode continuar sua execução normalmente, utilizando a resposta recebida para qualquer ação ou decisão que seja apropriada.
- `getID`: retorna o atributo `agencyID` da agência em questão.
- `getName`: retorna o atributo `agencyNamename` da agência em questão (objeto Agency).
- `getAgentByID`: retorna instância da classe `Agent` relacionado ao id enviado por parâmetro.
- `getAddress`: retorna o atributo `address` relacionado à agência em questão.
- `getAgentsList`: retorna os agentes associados à agência em questão.

- **AgencyImpl:** A classe `AgencyImpl` é implementação da interface Agency. Nesta classe, são realizadas operações como adicionar agentes, mover agentes entre agências, enviar e receber mensagens, além de gerenciar a lista de agentes da agência.
  - `agencyID:` Uma string que representa o código único da agência.
  - `agencyName:` Uma string que representa o nome da agência.
  - `agencyPort:` Um inteiro que indica a porta em que a agência está conectada.
  - `agentsList:` Uma lista ligada que contém todos os agentes associados à agência em questão.
  - `receivedMessages:` Um HashMap que contém todas as mensagens recebidas pela agência, mapeando uma string com o identificador do agente destinado à mensagem e uma fila ligada bloqueante de mensagens.
  - `answeredMessagens:` Um HashMap que contém todas as mensagens respondidas pela agência, mapeando uma string com o identificador do agente responsável pela mensagem e uma fila ligada bloqueante de mensagens.

- **Message:** A classe `Message` representa uma mensagem enviada entre agentes. Cada mensagem possui um identificador único, o ID do agente de origem, o ID do agente de destino e o conteúdo da mensagem.

- `messageID`: Uma string que representa o código único da mensagem.
- `messageString`: Uma string que representa o conteúdo da mensagem.
- `responseString`: Uma string que contém o conteúdo de resposta da mensagem.
- `originAgentID`: Uma string que contém o ID do agente que enviou a mensagem originalmente.
- `destinationAgentID` Uma string que contém o ID do agente a quem a mensagem se destina.

Dentro de `Agent`, tem-se as classes:

- **Agent:** A classe `Agent` representa um agente do sistema, sendo uma interface remota deste. Cada agente possui um ID, um nome, uma referência para a agência a qual pertence e métodos para enviar e receber mensagens. Os métodos declarados nessa interface são:
  - `id`: Uma string que representa o código único do agente em questão.
  - `name`: Uma string que represente o atributo `agentName` do agente em questão.
  - `agency`: Uma instância de `Agency` à qual o agente pertence.

Dentro de `NamingService`, tem-se:

- **NamingService:** A classe `NamingService` é uma interface remota que define os métodos que o serviço de registro das agências deve implementar, como obter uma agência pelo ID ou nome, adicionar agentes e mover agentes entre agências. A interface implementa os métodos:

```
public interface NamingService extends Remote {
    public Agency getAgencyByID(String agencyID) throws RemoteException;
    public Agency getAgencyByName(String agencyName) throws RemoteException;
    public LinkedList<Agency> getAgenciesList() throws RemoteException;
    public Agency getAgencyByAgentID(String agentID) throws RemoteException;
    public Agent getAgentByID(String agentID) throws RemoteException;
    public void registerAgency(Agency agency) throws RemoteException;
    public boolean removeAgent(String agentID) throws RemoteException;
    public boolean removeAgency(String agencyID) throws RemoteException;
    public boolean addNewAgent(String agentID, String agentName, String agencyID) throws Remot
    public boolean moveAgent(String agentID, String oldAgencyID, String newAgencyID) throws Re
    public LinkedList<Agent> getAgentsByAgencyID(String agencyID) throws RemoteException;
}
```

Figura 3: Captura de tela do código.

- `getAgencyByID`: retorna instância da classe `Agency` relacionada ao identificador enviado como parâmetro registrada no serviço de nomes.

- `getAgencyByName`: retorna instância da classe `Agency` relacionada ao atributo `agencyName` enviado como parâmetro registrada no serviço de nomes.
- `getAgenciesList`: retorna lista ligada de agências registradas no serviço de nomes.
- `getAgencyByAgentID`: retorna instância da classe `Agency` registrada no serviço de nomes relacionada ao identificador de um agente contido nela enviado como parâmetro.
- `getAgentByID`: retorna instância da classe `Agent` relacionado ao seu identificador enviado como parâmetro.
- `registerAgency`: registra nova agência à lsita de agências ativas no serviço de nomes através dos atributos `agencyId` e `agencyName`.
- `removeAgency`: remove agência da lista de agências ativas no serviço de nomes através do atributo `agentId`.
- `moveAgent`: altera a agência à qual o agente enviado por parâmetro através do `agentId` está associado.
- `removeAgent`: remove agente associado à agência registrada no serviço de nomes através de seu identificador.
- `addNewAget`: adiciona agente associando-o à agência registrada no serviço de nomes através de seu identificador, nome e identificador da agência.
- `getAgentsByAgencyID`: retorna uma lista ligada dos agentes associados às agências, ou seja, a lista de agentes de uma agência específica, identificada pelo `agencyId`.

- **NamingServerImpl:** A classe `NamingServerImpl` é a implementação da interface `NamingService`. Nesta classe, são registradas as agências e agentes disponíveis no sistema, e é fornecida uma maneira de acessar as informações das agências e agentes registrados.
  - `registeredAgencies:` Um HashMap que correlaciona os atributos `agencyId` e `agencyName` (instâncias Agency) entre si das agências registradas no serviço de nomes.

  Dentro de `User`, tem-se:

- **UserInterface:** A classe `UserInterface` é responsável por interagir com o usuário, exibir mensagens no console e obter comandos de entrada do usuário. Ela fornece métodos simples para exibir mensagens, ler comandos e lidar com erros de exceção.

```java
public class UserInterface {
    private static final Scanner scanner;

    private UserInterface() {
        throw new AssertionError();
    }

    static {
        scanner = new Scanner(System.in);
    }

    public static String getUserCommand() {
        System.out.print(">> ");
        return scanner.nextLine().trim();
    }

    public static void displayMessage(String message) {
        System.out.println(message);
    }

    public static void displayError(String errorMessage, Exception exception) {
        System.err.println(errorMessage);
        System.err.println(exception.getMessage());
    }

    public static void printLine() {
        System.out.println();
    }
}
```

Figura 4: Captura de tela do código.

- **Server:** A classe `Server` é a classe responsável por iniciar uma agência específica, tornando-a disponível para os clientes e registrando-a no serviço de registro das agências.

```java
import Agency.AgencyImpl;
import NamingService.NamingService;

public class Server {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("Usage: Server <agency_name> <port>");
            System.exit(1);
        }

        String agencyName = args[0];
        int port = Integer.parseInt(args[1]);
        try {
            AgencyImpl agency = new AgencyImpl(agencyName, port);
            Registry registry = LocateRegistry.createRegistry(port);
            Naming.rebind(agencyName, agency);

            NamingService namingService = (NamingService) Naming.lookup("rmi://localhost:8080/
            namingService.registerAgency(agency.getID(), agency.getName());

            UserInterface.displayMessage("Server bound");
        } catch (Exception e) {
            UserInterface.displayError("Server Exception.", e);
        }
    }
}
```

Figura 5: Captura de tela do código.

- **NamingServiceServer:** A classe `NamingServiceServer` é classe responsável por iniciar o serviço de registro das agências (`NamingService`) e torná-lo disponível para as outras partes do sistema.

```java
public class NamingServiceServer {
    public static void main(String[] args) {
        try {
            NamingServiceImpl namingService = new NamingServiceImpl();
            LocateRegistry.createRegistry(8080);
            Naming.rebind("rmi://localhost:8080/namingservice", namingService);
            UserInterface.displayMessage("NamingService bound");
        } catch (Exception e) {
            UserInterface.displayError("NamingService Exception.", e);
        }
    }
}
```

Figura 6: Captura de tela do código.

- **Client:** A classe `Client` representa o cliente do sistema distribuído. O cliente pode interagir com o sistema através de comandos no terminal, como listar agências, criar agentes, mover agentes entre agências e enviar mensagens.

```java
public static void main(String[] args) {
    try {
        bindNamingService();
        boolean running = true;
        String command;
        showCommands();
        while (running) {
            command = UserInterface.getUserCommand();
            switch (command) {
                case "show-commands":
                    showCommands();
                    break;
                case "bind":
                    UserInterface.displayMessage("Enter agency name:");
                    String agency = UserInterface.getUserCommand();
                    bindAgency(agency);
                    break;
                case "list-agencies":
                    listAgencies();
                    break;
                case "create-agent":
                    UserInterface.displayMessage("Enter agent ID:");
                    String agentID = UserInterface.getUserCommand();
                    UserInterface.displayMessage("Enter agency name:");
                    String agencyName = UserInterface.getUserCommand();
                    createAgent(agentID, namingService.getAgencyByName(agencyName));
                    break;
```

Figura 7: Captura de tela do código.

```java
                case "move-agent":
                    UserInterface.displayMessage("Enter agent ID:");
                    String agentToMessage = UserInterface.getUserCommand();
                    UserInterface.displayMessage("Enter agency name to move:");
                    String agencyDestination = UserInterface.getUserCommand();
                    moveAgent(agentToMessage, agencyDestination);
                    break;
                case "list-agents":
                    listAgents();
                    break;
                case "message":
                    UserInterface.displayMessage("Enter the message:");
                    String message = UserInterface.getUserCommand();
                    break;
                case "quit":
                    UserInterface.displayMessage("Client terminated.");
                    running = false;
                    break;
                default:
                    UserInterface.displayMessage("Invalid command.");
                    break;
            }
        }
    } catch (Exception e) {
        UserInterface.displayError("Client exception.", e);
    }
}
```

Figura 8: Captura de tela do código.

2.2 Descrição das escolhas de estrutura de dados

A utilização dessas estruturas de dados no contexto do programa tem as seguintes vantagens/motivos:

- Classe `AgencyImpl`:
  - `agentsList: LinkedList<Agent>` - Essa lista encadeada armazena os agentes registrados na agência. Ela é utilizada para gerenciar os agentes e suas operações.
  - `receivedMessages: Map<String, LinkedBlockingQueue<Message>>` - Esse mapa armazena as mensagens recebidas por cada agente. A chave do mapa é o ID do agente, e o valor é uma fila bloqueante (`LinkedBlockingQueue`) que contém as mensagens recebidas por esse agente. Essa estrutura é usada para armazenar e gerenciar as mensagens recebidas por cada agente.
  - `answeredMessages: Map<String, LinkedBlockingQueue<Message>>` - Similar ao `receivedMessages`, esse mapa armazena as mensagens respondidas por cada agente. A chave do mapa é o ID do agente, e o valor é uma fila bloqueante (`LinkedBlockingQueue`) que contém as mensagens respondidas por esse agente. Essa estrutura é usada para armazenar e gerenciar as respostas enviadas por cada agente.

  Essas estruturas de dados são usadas para manter informações importantes sobre os agentes e suas interações na agência. As listas e mapas fornecem uma maneira eficiente de armazenar e acessar agentes e mensagens. As filas bloqueantes (`LinkedBlockingQueue`) são usadas para garantir a sincronização e a ordem correta das mensagens enviadas e recebidas pelos agentes, garantindo que as operações de envio e recebimento sejam seguras em um ambiente de concorrência.

- Classe `Message`:
  - `messageId: String` - O ID único da mensagem é gerado usando a classe `UUID`. Ele é usado para identificar exclusivamente cada mensagem.
  - `messageString: String` - A mensagem em si, que contém o conteúdo da comunicação entre os agentes.
  - `responseString: String` - A resposta associada à mensagem, caso exista. Essa variável é preenchida pelo receptor da mensagem após processá-la.
  - `originAgentID: String` - O ID do agente que enviou a mensagem.
  - `destinationAgentID: String` - O ID do agente que deve receber a mensagem.

Essas variáveis são usadas para representar uma mensagem entre agentes. O ID único é usado para rastrear as mensagens e suas respostas. As variáveis `originAgentID` e `destinationAgentID` são usadas para indicar o remetente e o destinatário da mensagem, permitindo que ela seja encaminhada corretamente.

- Classe `Agent`:
  - `id: String` - O ID do agente, que é usado para identificá-lo de maneira exclusiva.
  - `name: String` - O nome do agente, que é uma descrição identificativa do agente.
  - `agency: Agency` - A agência à qual o agente está associado.

  Essas variáveis são usadas para representar um agente. O ID é importante para identificar o agente em operações de movimento e destruição, e o nome é usado para exibir informações sobre o agente. A variável `agency` é usada para rastrear a agência à qual o agente está associado.

- Classe `NamingServiceImpl`:
  - `registeredAgencies: LinkedList<Agency>` - Essa lista encadeada armazena todas as agências registradas no serviço de nomes.

3 Execução do projeto

3.1 Para compilar o projeto

Dentro da pasta `distributed-agencies`, executar o comando

```
$ find src -name "*.java" -print | xargs javac -d bin
```

3.2 Para executar o projeto

Dentro da pasta `distributed-agencies`, execute os seguintes comandos:

```
$ cd bin; rmiregistry
```

Abra outro terminal:

```
$ java -cp bin User.NamingServiceServer
```

Abra outro terminal. Troque `<agencyname>` pelo novo nome do servidor que será criado e `<port>` pelo número da porta que deseja conectar:

```
$ java -cp bin User.Server <agency_name> <port>
```

Abra outro terminal:

```
$ java -cp bin User.Client
```

3.3 Para usufruir do sistema

O usuário possui as seguintes possibilidades de comando:

- `show-commands`: Exibe a lista de comandos disponíveis no cliente.
- `bind <agency_name>`: Conecta o cliente a uma agência específica pelo nome. Após executar este comando, o cliente poderá interagir com a agência selecionada.
- `list-agencies`: Lista todas as agências disponíveis no sistema, mostrando seus nomes e IDs.
- `create-agent`: Cria um novo agente e o associa a uma agência específica. O cliente solicitará o ID do agente e o nome da agência onde o agente deve ser criado.
- `move-agent`: Move um agente de uma agência para outra. O cliente solicitará o ID do agente que será movido e o nome da agência de destino.
- `list-agents`: Lista todos os agentes registrados no sistema, mostrando seus IDs e nomes, agrupados por agência.
- `message`: Envia uma mensagem de um agente para outro. O cliente solicitará o ID do agente de origem, o ID do agente de destino e o conteúdo da mensagem.
- `quit`: Encerra o cliente, terminando sua execução.

O fluxo do sistema é o seguinte:

- **Iniciar o Servidor NamingService**
  - O usuário inicia o servidor NamingService executando o programa "NamingServiceServer". Isso cria uma instância da classe NamingServiceImpl e a registra no RMI Registry, tornando-o acessível para os clientes.
- **Iniciar Agências**
  - O usuário inicia uma ou mais agências executando o programa "Server"com o nome da agência e a porta desejada. Cada agência criada é registrada no NamingService para que os clientes possam encontrá-las.
- **Iniciar o Cliente**
  - O cliente é executado pelo usuário usando o programa "Client". O cliente se conecta ao NamingService para descobrir as agências disponíveis no sistema.
- **Comando 'show-commands'**

  - O cliente digita o comando "show-commands"no prompt.
  - O programa exibe a lista de comandos disponíveis para o usuário, mostrando uma descrição breve de cada um.

- **Comando 'bind'**
  - O cliente digita o comando "bind ¡agencyname¿"no prompt, onde ≪agencyname¿"é o nome da agência que deseja se conectar.
  - O cliente obtém o objeto remoto da agência selecionada a partir do NamingService usando o nome da agência fornecido.
  - A conexão com a agência é estabelecida, permitindo que o cliente interaja com ela.

- **Comando 'list-agencies'**
  - O cliente digita o comando "list-agencies"no prompt.
  - O programa obtém a lista de todas as agências registradas no NamingService.
  - O programa exibe a lista de agências disponíveis, mostrando seus nomes e IDs.

- **Comando 'create-agent'**
  - O cliente digita o comando "create-agent"no prompt.
  - O programa solicita ao usuário que insira o ID do agente que deseja criar e o nome da agência onde o agente deve ser criado.
  - O cliente chama o método remoto "addAgent"da agência selecionada, passando o novo agente como parâmetro.
  - A agência cria o agente com o ID e o nome fornecidos e o associa a si mesma.
  - O agente é registrado no NamingService para que possa ser localizado por outros clientes.
  - Um novo Thread é criado para o agente e a execução do agente é iniciada.

- **Comando 'move-agent'**
  - O cliente digita o comando "move-agent"no prompt.
  - O programa solicita ao usuário que insira o ID do agente que deseja mover e o nome da agência de destino.
  - O cliente chama o método remoto "moveAgent"da agência atual, passando o ID do agente e o nome da agência de destino como parâmetros.
  - A agência atual verifica se possui o agente com o ID fornecido e se a agência de destino existe no sistema.

- Se a agência atual não possuir o agente, ela tenta localizá-lo usando o NamingService.
- Se o agente for encontrado, a agência atual chama o método "moveAgent"da agência de destino para transferir o agente.
- O agente é removido da agência atual e adicionado à agência de destino.

- **Comando 'list-agents'**
  - O cliente digita o comando "list-agents"no prompt.
  - O programa obtém a lista de todas as agências registradas no NamingService.
  - Para cada agência, o programa obtém a lista de agentes registrados naquela agência.
  - O programa exibe a lista de agentes disponíveis, mostrando seus IDs e nomes, agrupados por agência.

- **Comando 'message'**
  - O cliente digita o comando "message"no prompt.
  - O programa solicita ao usuário que insira o ID do agente de origem, o ID do agente de destino e o conteúdo da mensagem.
  - O cliente chama o método remoto "sendMessageToAgent"da agência atual, passando os IDs dos agentes e o conteúdo da mensagem como parâmetros.
  - A agência de origem cria um objeto Message com as informações da mensagem e o envia para a agência de destino.
  - A agência de destino recebe a mensagem e a coloca na fila de mensagens recebidas do agente de destino.
  - A agência de destino envia uma resposta ao agente de origem, que coloca a resposta na fila de mensagens respondidas do agente de origem.
  - O cliente recebe a resposta e exibe-a.

- **Comando 'quit'**
  - O cliente digita o comando "quit"no prompt.
  - O programa encerra o cliente, terminando sua execução.

Em resumo, o programa permite ao usuário interagir com as agências e agentes utilizando os comandos disponíveis no cliente. O cliente se conecta ao NamingService para localizar as agências e pode criar agentes, movê-los entre agências e enviar mensagens entre eles. As agências utilizam o NamingService para registrar os agentes e permitir a comunicação entre eles, possibilitando simulações distribuídas de sistemas multiagentes.

3.4 Informações adicionais da máquina

```
$ lsb_release -a
```

No LSB modules are available. Distributor ID: Ubuntu Description: Ubuntu 22.04.2 LTS Release: 22.04 Codename: jammy

```
$ java -version
```

openjdk version "11.0.19"2023-04-18 OpenJDK Runtime Environment (build 11.0.19+7-post-Ubuntu-0ubuntu122.04.1) OpenJDK 64-Bit Server VM (build 11.0.19+7-post-Ubuntu-0ubuntu122.04.1, mixed mode, sharing)

3.5 Código-fonte

O repositório com o código-fonte pode ser acessado na plataforma Github pelo link: https://github.com/mirelameic/distributed-agencies.