Trabalho Prático
Sistema de Partilha de Bicicletas

Ricardo Ferreira[A82568] Hugo Faria[A81283]
Rodolfo Silva[A81716] Bruno Veloso[A78352]

Universidade do Minho, Departamento de Informática

Resumo

O presente relatório tem como objetivo abordar a modelação de um sistema multi agente para simular um Sistema de Partilha de Bicicletas (SPB), bem como a sua implementação. Inicialmente será feita a contextualização do problema e de seguida serão abordados os diagramas feitos para o modelação do sistema, evidenciando para cada um quais as razões das decisões tomadas. Futuramente no relatório será apresentada a sua implementação, apresentando as medidas tomadas.

Keywords: agente, sistema multi agente, classes, diagrama,implementação, Sistemas de Partilha de Bicicletas

Conteúdo

1 Introdução

Este relatório inicialmente é referente à primeira fase do trabalho prático da unidade curricular de Agentes Inteligentes do perfil de Sistemas Inteligentes e futuramente no mesmo é explicada a sua implementação que corresponde à segunda fase do trabalho prático. Este projeto consiste em modelar e implementar um sistema multi agente (SMA) que irá lidar com a distribuição de bicicletas em certas zonas. Este sistema multi agente irá permitir às pessoas requisitarem bicicletas para percorrer distâncias curtas e tentará lidar com a distribuição das mesmas de forma a não existir um aglomeramento elevado de bicicletas no mesmo espaço.

Nesta primeira fase, o objetivo é efetuar uma modelação do sistema multi agente. Para isso será apresentada a sua arquitetura e os protocolos de interação entre agentes utilizando Agent UML.

Numa segunda fase é abordada a implementação do sistema multi agente, onde explicaremos as medidas tomadas e como este foi implementado. Será apresentada uma interface que nos ajudará a simular o SMA e assim visualizar melhor o seu comportamento.

1.1 Estrutura do Relatório

O relatório encontra-se estruturado da seguinte forma:

Inicialmente, na *secção 1* será feita uma breve introdução e contextualização do problema do caso em estudo, a *secção 2* apresenta os vários agentes a implementar e quais as suas características, de seguida, na *secção 3* serão apresentadas as classes adicionais e os diferentes *behaviours* presentes no diagrama de classes, a *secção 4* apresenta os diferentes diagramas utilizados para modelar o sistema, explicitando para cada um as decisões tomadas. Na *secção 5* será abordada toda a implementação relativamente ao sistema multi agente. Na *secção 6* abordaremos os aspetos relacionados com a UI que nos permitiu visualizar o sistema multi agente e ver o seu comportamento de uma melhor forma. Já na *secção 7* é feita uma análise de simulação do sistema onde é possível visualizar o comportamento do sistema com e sem incentivo. Por fim o relatório termina com conclusões na *secção 8*, onde é também apresentada uma análise crítica aos resultados obtidos.

1.2 Contextualização

Nos dias que correm um dos meios de transporte mais utilizados pelas pessoas para deslocações curtas tem sido a bicicleta. Sendo maioritariamente utilizados nas cidades, uma vez que são amigas do ambiente e evitam o tráfego elevado quando comparado com outros meios de transporte. De forma a lidar com aumento acentuado de procura de bicicletas existe a necessidade de criação de um sistema possível de aluguer de bicicletas e respetiva orientação do utilizador no após aluguer e devolução das mesmas.

Para isto foi-nos proposto a implementação de um sistema multi agente que consiga resolver tal problema e assim facilitar as nossas vidas quotidianas, em

que numa fase inicial correspondente a este relatório se passará à modelação do mesmo sistema.

Este sistema possui alguns requisitos básicos como:

- Utilizador alugar bicicleta;
- Estações de bicicletas possam indicar melhores opções de devolução utilizando incentivos, de forma a tentar obter uma melhor distribuição das bicicletas, evitando assim um aglomerado;
- Permitir ao utilizador aceitar ou recusar incentivos.

2 Agentes

Após uma contextualização do problema proposto, serão abordados quais os agentes que o sistema multi agente deve implementar, descrevendo de forma detalhada cada um deles e explicando o porquê de tais decisões. São estes:

- **Agente Utilizador**;
- **Agente Estação**;
- **Agente Interface**.

2.1 Agente Utilizador

Este agente representará um utilizador do sistema ou seja, uma pessoa que pretende alugar uma bicicleta.

Com o objetivo de facilitar a resolução do problema proposto, este utilizador sempre que é criado no sistema começará com bicicleta e numa posição de uma das estações. Este agente terá também de saber qual a posição atual e o seu percurso a percorrer e será o mesmo a ter que calcular a distância percorrida até ao momento do percurso. Para isto, este terá então que guardar informações como:

- **Posição inicial**, para saber de onde partiu;
- **Posição destino**, que representa a posição da estação destino do utilizador;
- **Posição atual**, para que seja possível calcular a distância percorrida do percurso;
- **Estação destino**, para indicar o identificador da estação onde este tem de entregar a bicicleta;
- **Distância percorrida**, pois é preciso ser o utilizador a calcular a distância que este já percorreu do seu percurso por forma a mais tarde avisar o agente interface sobre a sua chegada;
- **Estação próxima**, que representará a estação mais próxima do mesmo utilizador, podendo ser a estação de destino ou não.
- **Aceitou Incentivo**, variável indicativa se o utilizador já aceitou um incentivo no passado.

2.2 Agente Estação

Este agente representará uma estação, representada por um ponto em que é possível requisitar e devolver bicicletas alugadas e por uma área de controlo que representa a área próxima dessa estação.

Agente estação terá que ter características como:

- Este terá que saber todos os utilizadores que se encontram na área de proximidade do mesmo;
- Estará em constante comunicação com os utilizadores para saber a posição dos mesmos, actualizando caso necessário, a sua lista de utilizadores na sua área de controlo;
- Terá que saber todas as informações relativamente a bicicletas (disponíveis e capacidade total de armazenamento);
- Tentar prever a quantidade de utilizadores que num futuro próximo possam devolver as bicicletas na sua estação;
- Tentar distribuir da melhor forma as bicicletas pelas diferentes estações oferecendo incentivos conforme a lotação de cada estação levando o utilizador a entregar a bicicleta numa estação que esteja com menos bicicletas.

Para responder a estas características, terá que guardar informações como:

- **Posição da estação**, para saber as suas coordenadas;
- **Capacidade**, que representa a capacidade total de bicicletas da estação;
- **Bicicletas**, para indicar quantas bicicletas a estação tem disponíveis de momento;
- **Coordenadas de área de controlo**, em que todas as coordenadas da sua área de proximidade são conhecidas e permitindo assim calcular quais os utilizadores dentro da sua área próxima;
- **Utilizadores na área de controlo**, para saber a quantidade de utilizadores que se encontram na sua área de controlo, identificando os mesmos, e assim prever a quantidade de utilizadores que possam devolver futuramente a bicicleta na sua estação;
- **Ocupação das estações**, para o mesmo saber o quão lotadas as outras estações estão e conseguir assim calcular melhor um incentivo a dar ao utilizador, de forma a atrair o mesmo para uma estação com menor taxa de ocupação.

2.3 Agente Interface

O agente interface será quem comunica com os agentes estação, de forma a que este(Agente Interface) consiga saber a disponibilidade de cada estação em termos de bicicletas (capacidade total e bicicletas disponíveis) e consequentemente saber também as posições dos utilizadores. Com estas informações obtidas depois de uma comunicação entre agentes, o agente interface mostrará toda esta informação graficamente para que o utilizador consiga visualizar a informação no sistema multi agente.

3 Classes Adicionais e Behaviours

Passaremos a abordar as classes adicionais que achamos pertinentes para a modelação deste sistema multi agente, anotando algumas características/atributos das mesmas. Estas classes passarão desde:

- classes de auxílio à implementação do sistema;
- classes que representarão os *behaviours*, que definirão qual o comportamento dos agentes;

3.1 Coordenadas

Esta classe serve para representar qualquer coordenada num mapa 2D. Para tal tem atributos como:

- **Coordenada X**, representa a posição no eixo do x num mapa 2D;
- **Coordenada Y**, que representa a posição no eixo do y num mapa 2D.

3.2 Mapa

O mapa vai representar o estado a ser desenhado pelo agente interface. Com esse efeito, possui o seguinte atributo:

- **Posição Estações**, um map que possui as Coordenadas de todas as estações e a sua respetiva identificação.

3.3 System

Devido ao facto de este sistema servir como "uma simulação", é necessário que de alguma forma seja possível simular o mesmo. Nesse sentido, o grupo decidiu modelar uma classe chamada System, que tem como objetivo criar utilizadores de x em x tempo e assim conseguir testar o mesmo para verificar a sua viabilidade.

3.4 Ticker Behaviour

Utilizado nos seguintes comportamentos dos agentes:

- **Movimento do utilizador**, em que de x em x tempo este *behaviour* irá movimentar o utilizador;
- **Geração de utilizadores**, no qual a cada y segundos, este irá gerar utilizadores para o sistema;
- **Desenho da interface**, em que de x em x segundos desenha o mapa e as informações das estações.

3.5 One-Shot Behaviour

Vai ser aplicado para fazer os comportamentos que requerem mandar uma mensagem a outro agente. A resposta vai ser recebida por parte dos *cyclic behaviours* que os agentes vão apresentar.

Serão utilizados nos seguintes momentos:

- Utilizador pedir a identificação da estação que controla a área do seu destino;
- Utilizador informar a estação da sua área sobre a sua nova posição;
- Estação avisar o utilizador que necessita de entregar a bicicleta quando este chegar ao seu destino;
- Estação fornecer o incentivo um utilizador para este se deslocar para outra estação que contenha poucas bicicletas disponíveis.

3.6 Cyclic Behaviour

O *cyclic behaviour* vai ser maioritariamente utilizado nos agentes como forma de continuamente esperar por *informs* ou *requests* feitos por outros agentes. Estes serão utilizados:

- Na espera por resposta por parte de uma estação ou utilizador.

4 Diagramas

De seguida serão apresentados e explicados alguns diagramas que serão a base da implementação do projeto numa segunda fase. Estes diagramas servem como base para apresentar uma arquitetura do sistema no geral, bem como os protocolos de comunicação usados entre os agentes e para designar o fluxo do sistema.

Para isto, foram desenvolvidos os seguintes diagramas: diagrama de classe, comunicação, atividade, estados e sequência.

4.1 Diagrama de Classes

Nesta secção é apresentada a arquitetura do sistema através do diagrama de classes. Neste diagrama é possível verificar as diferentes classes a implementar e visualizar também as diferentes relações que terão umas com as outras de forma a que o sistema funcione como um todo.

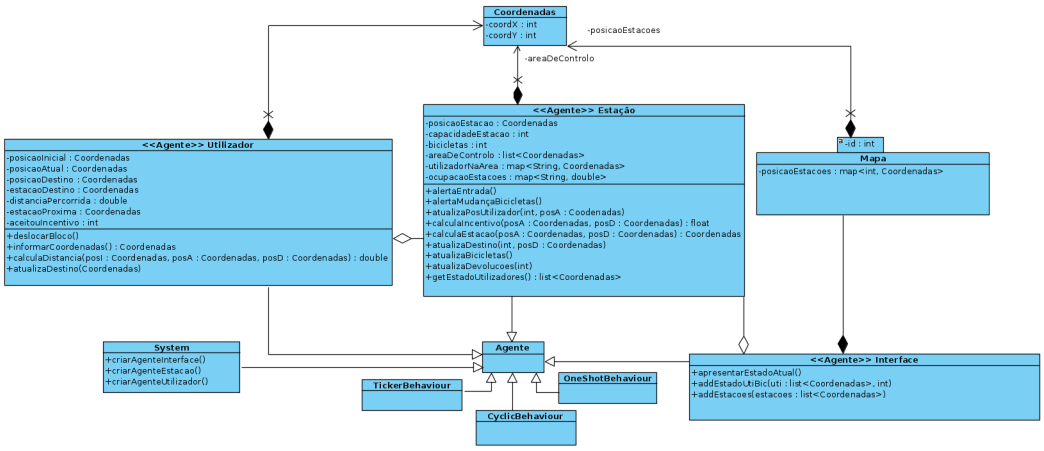

Figura 1. Arquitetura do sistema

Uma vez que as classes foram previamente explicadas na *secção 3*, passaremos agora a abordar as relações mais importantes existentes entre estas.

É de notar que o agente interface tem uma relação com a classe Mapa, isto deve-se ao facto de que, caso seja pertinente, o agente interface possa aceder a Mapa de forma a ser mais fácil obter informação sobre as posições de estações em vez de estar em constante comunicação para saber a sua posição visto que esta é fixa(Posição das Estações). Como era de esperar, existe uma relação entre o utilizador e a estação, sendo esta a principal comunicação que existirá neste sistema, uma vez que o utilizador precisa de fornecer a sua posição à estação, saber a lotação das mesmas, e a estação de fornecer incentivos ao utilizador. Por fim, é possível verificar ainda que também o agente interface tem relação com o agente estação, no qual servirá para a interface conhecer as lotações de cada estação e as posições de cada utilizador na sua área.

Sendo importante referir que algumas das relações identificadas anteriormente serão, na prática abstraídas por mecanismos de comunicação disponibilizados pelo JADE.

4.2 Diagrama de Comunicação

De forma a dinamizar o sistema, é necessário que os seus agentes comuniquem entre si, para isso é necessário que existam protocolos de comunicação estabelecidos, que serão as performatives(2), utilizados no auxílio da comunicação dos agentes. Na seguinte imagem, serão apresentadas os diferentes performatives usados entre cada par de agentes.

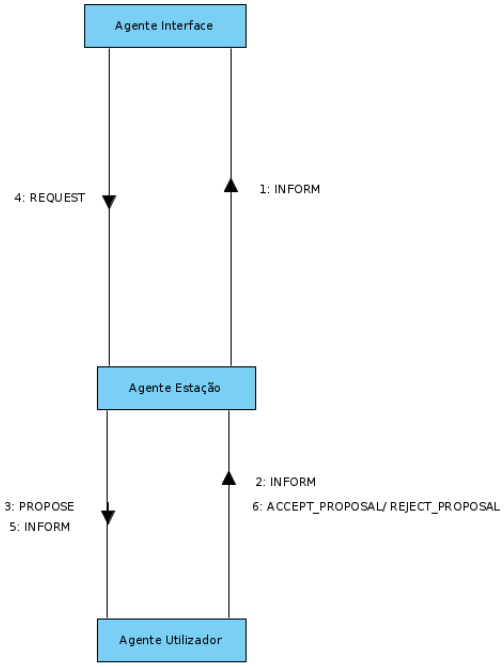

Figura 2. Comunicação entre agentes

Passaremos a explicar o porquê do uso de determinadas *performatives*.

No que diz respeito à comunicação entre Agente Estação e Agente Utilizador, existe um *inform* feito pelo utilizador no qual indica que se moveu e a respetiva distância. Já do lado do Agente Estação, este utiliza as *performatives propose* e *inform*. A *propose* é utilizada quando quer fornecer ao utilizador uma opção de escolha, que no caso deste sistema será o incentivo, e o *accept_proposal/reject_proposal* será quando o utilizador decidir escolher ou não o incentivo fazendo com que o Agente Estação informe o utilizador se mudou ou não a sua estação original ou a manteve usando para tal o *Inform*.

Quanto à comunicação entre Agente Interface e Agente Estação, o Agente Interface utiliza uma *performatives request* que servirá para quando o Agente Interface quiser saber do estado atual do mapa (utilizadores e informações relativamente a cada estação) e do lado do Agente Estação é utilizado o *Inform*. para fornecer os dados que o Agente Interface pretende saber.

Definidos assim os protocolos de comunicação, fica então assim definida toda a comunicação envolvente no sistema multi agente.

4.3 Diagrama de Atividade

Quanto ao fluxo que indica como todo o sistema deve fluir, foi utilizado para expressar este fluxo um diagrama de atividade.

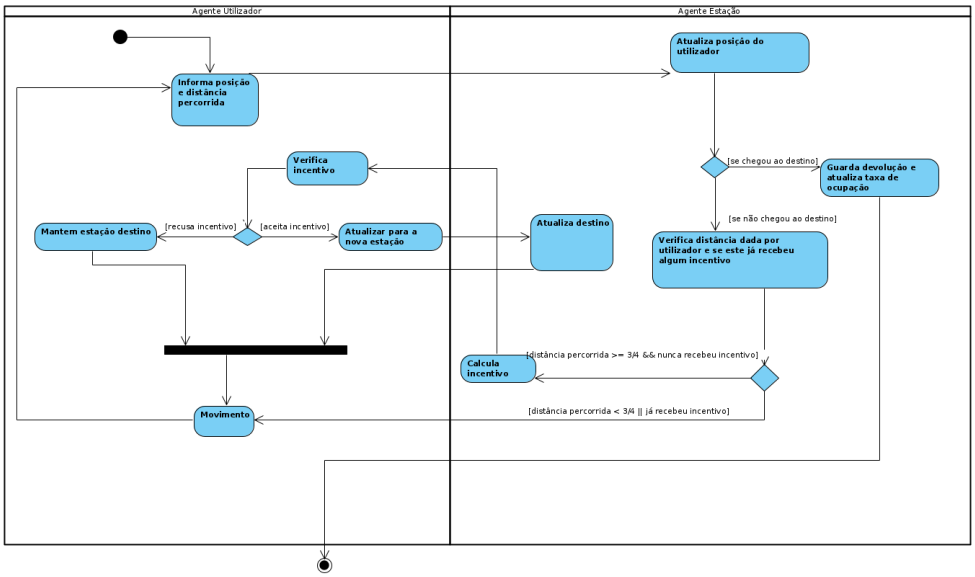

Figura 3. Fluxo do sistema

Neste diagrama é possível verificar de forma sucinta que é o utilizador o responsável pelo calculo da distância total percorrida e da comunicação deste calculo e respectiva posição atual á Estação mais próxima (da sua área). Esta verifica se este chegou ao destino ou não. Se chegou ao destino atualiza a sua lotação e guarda a devolução da bicicleta. Se não chegou ao destino, a estação verifica se a distancia total percorrida ja atingiu os 3/4 do caminho total proposto, se exceder esse valor esta calcula um incentivo que indica o utilizador para outra estação ou não. O incentivo é sempre calculado quando o utilizador já percorreu no mínimo 3/4 do caminho e quando este nunca recebeu incentivos, e este incentivo, caso não seja benéfico dar um incentivo ao utilizador, será 0. Caso se calcule um incentivo, o utilizador pode recusar o mesmo ou não. Se recusar, o destino dele mantêm-se. Se aceitar, é atualizado o seu destino. Caso não se calcule incentivo, este utilizador pode continuar o seu caminho até ao destino.

Posto isto, este será todo o fluxo a ter em conta para a futura implementação deste sistema.

4.4 Diagrama de Estados

Quanto ao estado interno dos agentes utilizador, foi modelado um diagrama de estado para o mesmo. Neste diagrama pode-se verificar todos os estados que estes vão tomar, desde que saíram da estação inicial, desde ao estado após aceitação de incentivo ou não, até à devolução da bicicleta.

Estes podem ser observados na figura seguinte:

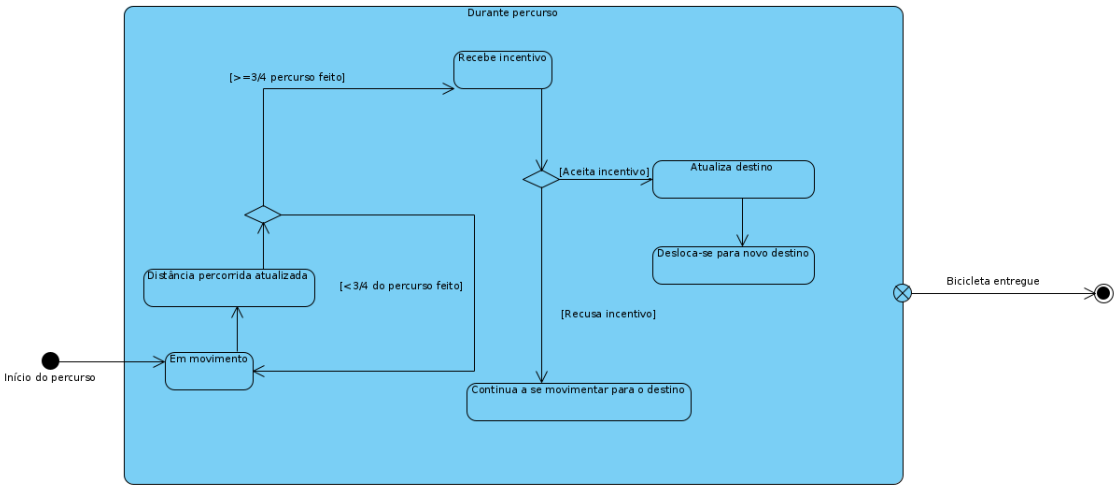

Figura 4. Estado interno dos agentes utilizador.

4.5 Diagramas de Sequência

No diagrama de sequência que se segue, é possível verificar todo o processo de obter informação até ao desenho da interface com tal informação. Para isso o agente interface pergunta à classe Mapa as posições das estações, ao qual esta lhe fornece tal informação. De seguida o agente interface pede a cada estação, os utilizadores que se encontram na área bem como o número de bicicletas disponíveis nas estações.

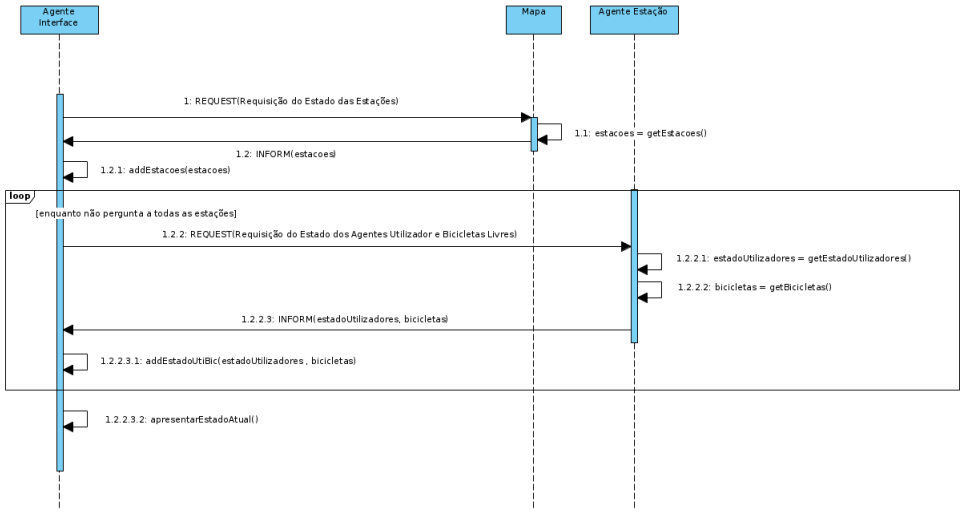

Figura 5. Diagrama de sequência que representa desenho da interface

No diagrama de sequência seguinte podemos observar toda a comunicação que existirá entre agente utilizador e agente estação. Esta comunicação já foi explicada anteriormente, no entanto é de notar que agora foram usadas *performatives* para explicar a mesma.

Sendo possível verificar que quando é fornecido um incentivo ao utilizador por parte da estação (usando a *performative PROPOSE*), este utilizador utilizará a *performative ACCEPT_PROPOSAL* caso queira aceitar o mesmo, ou *REJECT_PROPOSAL* caso o queira recusar. Existe também o uso da *performative Inform* por parte da estação para dizer que o destino foi alterado e *Inform* por parte do utilizador para dizer que chegou ao destino. De resto todo o diagrama flui da mesma forma que foi explicado no diagrama de actividades, por exemplo.

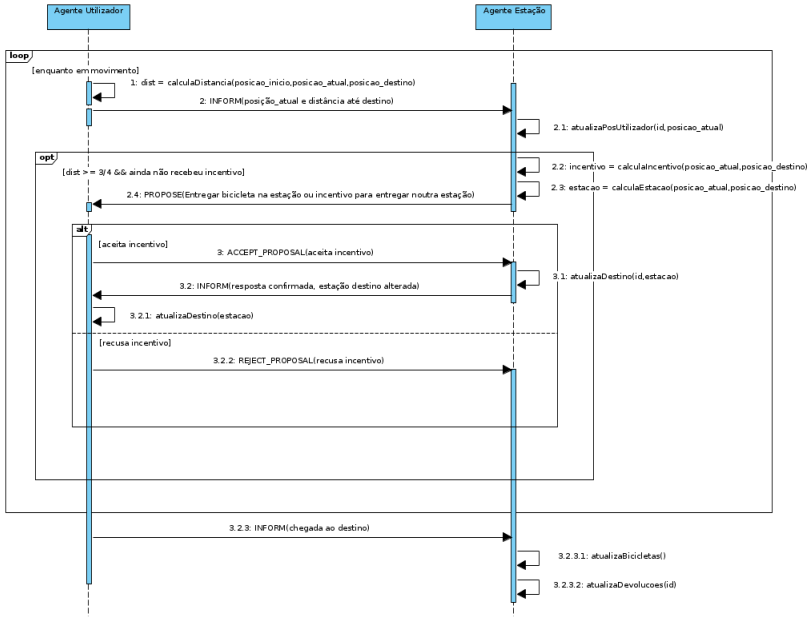

Figura 6. Diagrama de sequência que representa comunicação entre utilizador e estação

5 Implementação

Nesta secção será apresentada a estratégia seguida na implementação, acompanhada com excertos de código relevantes, onde será possível visualizar os *Behaviours* necessários para o bom funcionamento dos agentes e do sistema, assim como a comunicação estabelecida entre os diversos intervenientes e as respectivas *performatives*.

5.1 Considerações Iniciais

Com o objectivo de desenvolvimento de um sistema que se aproximasse de um simulador, sendo este totalmente controlado por agentes, foi necessário fazer algumas alterações no diagrama da figura 4.2, pois há mais comunicação entre os agentes do que aquilo que se propôs sendo de notar que há comunicação com o Agente System, ao qual este não tinha qualquer tipo de comunicação.

Também de notar que um utilizador receberá incentivos (caso ainda não tenha aceite um anteriormente) até este aceitar algum, ao contrário do que tinha sido proposto que era só receber um incentivo em toda a sua vida no sistema.

Para uma melhor visualização do sistema, foi criado uma UI de forma a se poder simular este sistema multi agente. Nesta UI podemos encontrar elementos como: a visualização de um mapa (em grelha) em que neste mapa se encontram as estações e os utilizadores (seguindo sempre as suas movimentações), também na UI é possível ver a ocupação total de cada estação (entre 0 e 1) e, por fim, é possível visualizar gráficos nos quais, um dos gráficos representa o número de bicicletas presentes em cada estação em que este é atualizado de x em x segundos, e o outro gráfico representa o número de falhas que cada estação teve que é atualizado de x em x segundos também. No nosso sistema, uma falha significa que um utilizador tentou começar numa estação em que esta tinha 0 bicicletas e não que uma estação chegou a 0 bicicletas.

De seguida apresenta-se o novo diagrama de comunicação que simboliza as novas comunicações existentes entre agentes.

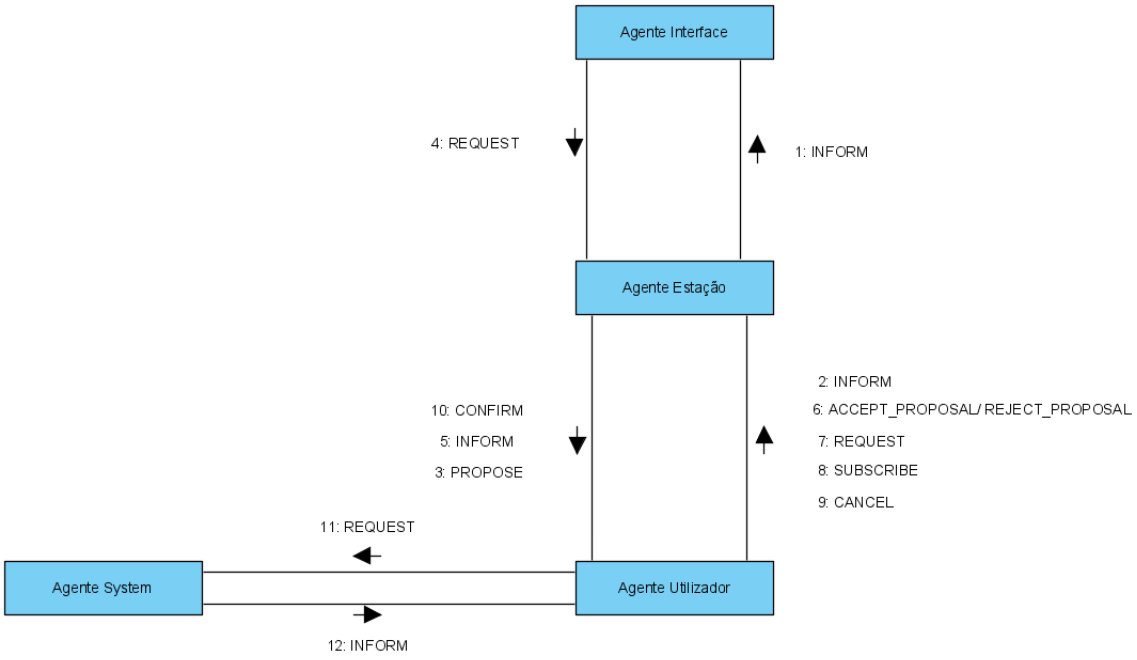

Figura 7. Comunicação entre agentes

5.2 Mapa

A classe mapa tem várias variáveis que representam informações relativas ao mapa. Sendo de seguida enunciadas e explicadas cada uma destas:

- **Tamanho do mapa**, definido inicialmente pelo utilizador que pretende fazer o teste no simulador;
- **Numero de estações**, definidas pelo utilizador que pretende fazer o teste no simulador;
- **Posição fixa das estações**, em que segundo o tamanho do mapa e número de estações escolhidas adquirem certa posição;
- **Zonas**, que define quais as posições em que uma estação tem controlo (matriz).

5.3 Agente Utilizador

O agente Utilizador é responsável por fazer uma viagem entre duas estações, sendo que decide de uma forma aleatória no momento em que é criado, em que estação vai dar início à sua viagem, enviando um *performative Request* seguido de um *performative Subscribe* caso obtenha permissão. Caso não consiga começar na estação desejada, é considerado como falha de sistema, pois não haviam bicicletas suficientes na estação e escolhe outra estação para começar, executando outra vez os mesmos passos. Este agente contem vários behaviours, dos quais:

- **Movimento**, *behaviour* responsável pelo movimento do utilizador;
- **InformarMovimento**, *behaviour* responsável pelo *inform* do movimento à estação;
- **InformarFim**, *behaviour* responsável pelo *inform* de chegada ao destino final;
- **TratamentoIncentivo**, *behaviour* responsável pela decisão de aceitar ou não um incentivo, e ainda pela receção de mensagens que o agente recebe.

Movimento

A partir de um **TickerBehaviour**, o Agente Utilizador escolhe a próxima célula á qual o utilizador se deve dirigir, com o objectivo de atingir a estação final. Por cada movimento feito, é criado um **OneShotBehaviour** *Informar-Movimento*.

```
1  private class Movimento extends TickerBehaviour{
2    // Behaviour responsavel pela movimentacao do utilizador
3    public Movimento(Agent a,long timeout){super(a,timeout);}
4
5    protected void onTick(){
6      if(concluido == 0) {
7        int atualX = posicaoAtual.getCoordX();
8        int atualY = posicaoAtual.getCoordY();
9        int destinoX = estacaoDestino.getCoordX();
10       int destinoY = estacaoDestino.getCoordY();
11
12       //Para variar movimento Utilizador
13       Random rand = new Random();
14
15       if (atualX < destinoX && atualY < destinoY){(...)}
16       if (atualX < destinoX && atualY > destinoY){(...)}
17       if (atualX < destinoX && atualY == destinoY){(...)}
18       if (atualX > destinoX && atualY < destinoY){(...)}
19       if (atualX > destinoX && atualY > destinoY){(...)}
20       if (atualX > destinoX && atualY == destinoY){(...)}
21       if (atualX == destinoX && atualY < destinoY){(...)}
22       if (atualX == destinoX && atualY > destinoY){(...)}
23       if (atualX == destinoX && atualY == destinoY){
```

```
        concluido = 1;
        myAgent.addBehaviour(new
          InformarFim(nomeEstacaoDestino));
      }
      if(concluido == 0) {
        ( ... )
        myAgent.addBehaviour(new InformarMovimento( ... ));
      }
    }
  }
}
```

InformarMovimento

A partir de um **OneShotBehaviour** o Agente Utilizador, após cada movimento, comunica com o agente estação qual a sua nova posição, através da *performative Inform*.

```
private class InformarMovimento extends OneShotBehaviour{
  double dtPercorrida;
  Coordenadas posAtual;

  public InformarMovimento( ... ){ ( ... ) }

    public void action(){
    try {
      //Construir a descricao de agente Estacao
      DFAgentDescription dfd = new DFAgentDescription();
      ServiceDescription sd = new ServiceDescription();
      sd.setType("Estacao");
      dfd.addServices(sd);

      //Array de Resultados da procura por estacoes
      DFAgentDescription[] resultados =
            DFService.search(myAgent, dfd);

      //Mandar mensagem a todas as Estacoes
      for (int i = 0; i < resultados.length; i++) {
        DFAgentDescription dfd1 = resultados[i];
        AID estacao = dfd1.getName();

        String mensagem = ( ... ) ;

        ACLMessage aEnviar = new INFORM;
        aEnviar.addReceiver(estacao);
        aEnviar.setContent(mensagem);
        myAgent.send(aEnviar);
      }
    } catch (FIPAException fe){ fe.printStackTrace(); }}}
```

InformarFim

Semelhante ao *behaviour* **InformarMovimento**, o *behaviour* **InformarFim** informa o agente estação, através de um *OneShotBehaviour* que chegou ao destino final, utilizando a *performative Inform*.

```
private class InformarFim extends OneShotBehaviour{

  private String nomeEstacao;

  public InformarFim(String nome){ (...) }

  public void action(){
    AID receiver = new AID();
    receiver.setLocalName(nomeEstacao);
    ACLMessage aEnviar = new ACLMessage(ACLMessage.INFORM);
    aEnviar.addReceiver(receiver);
    aEnviar.setContent("Cheguei ao destino.");
    myAgent.send(aEnviar);
  }
}
```

TratamentoIncentivo

Corresponde a um *CyclicBehaviour* por parte do Agente Utilizador, o objetivo deste *behaviour* é receber as propostas de destinos alternativos propostos pela estação e efetuar uma decisão em relação ao incentivo proposto. Para a execução desta decisão são efetuados os seguintes passos:

1. O Agente Utilizador recebe uma lista com o nome das estações recomendadas.
2. O Agente System é questionado pelo utilizador sobre a posição das estações.
3. De seguida, para cada par nome estação e coordenadas estação, o utilizador calcula a distância entre ele e a estação destino (*d1*) e entre ele e a estação proposta (*d2*).
4. É calculada a razão (*divDistancia*) entre *d2* e *d1* (*d2/d1*) até uma razão máxima de 2,0.
5. De seguida é feita a diferença entre a ocupação da estação destino e o valor maximo de ocupação definido para ser requisitada uma alteração do destino (0.25), assim temos, (0.25(25%) - Ocupação_Atual) * 2 (*extraOcupação*).
6. Depois é multiplicado o número de vezes que este utilizador já recusou incentivos por 0.1 (*fatorDeRecuso*), com o objetivo de se poder fornecer um desconto maior a utilizadores que recusam frequentemente.
7. Após isto, faz-se a soma total de *divDistancia + extraOcupação + fatorDeRecuso* e multiplica-se por 12.5% (percentagem de desconto base).
8. Faz-se o calculo do *fatorFinal* que consiste no valor de desconto calculado em cima, a dividir pela diferença entre d2 e d1.

9. Por fim, é comparado o *fatorFinal* com o valor base de decisão 0.9 + *greedinessDoUtilizador* (*[-0.3,0.3]*).
10. Se o *fatorFinal* for maior, o utilizador aceita o incentivo, se for menor recusa.
11. Se houver mais do que um incentivo que era aceitvel pelo o utilizador, este vai escolher o maior incentivo.

5.4 Agente Estação

O Agente Estação é responsável por saber quais os Agentes Utilizador na sua área, e, dependendo das taxas de ocupação das estações, incentivar utilizadores a partirem para um novo destino. Este é responsável por permitir um utilizador começar a viagem na sua estação através de uma *performative Inform* e monitorizar os utilizadores na sua área de controlo. Com estes objectivos, este agente contem os seguintes behaviours:

- **ReceiveMessages**, *behaviour* responsável por receber mensagens do Agente Utilizador e Agente Interface;
- **EntregaBicicleta**, *behaviour* responsável por processar as entregas das bicicletas na fila de espera.

ReceiveMessages

O *behaviour* **ReceiveMessages** tem como tipo **CyclicBehaviour**. Este fica ciclicamente à escuta de mensagens de forma a conseguir responder ás mesmas. Estas mensagens podem ser provenientes de Agente Interface ou Agente Utilizador.

Quanto ás mensagens com o Agente Interface, estas servem para fornecer informação ao Agente Interface tais como: taxa de ocupação das estações, número de bicicletas disponíveis em cada estação, número de falhas de cada estação até à data e a posição dos utilizadores.

Quanto à troca de mensagens com o Agente Utilizador, estas servem para fornecer/receber informações ou aceitar/recusar pedidos tais como: recebe informação sobre a conclusão do trajeto do utilizador para o poder remover, recebe informação do movimento do utilizador para poder atualizar a sua posição e de seguida decidir se quer mandar incentivo ou não, recebe *Requests* do utilizador para ver se este pode começar na estação, recebe *Subscribe* do utilizador para esta saber a ocupação futura, recebe *Cancel* do utilizador para saber que o utilizador já não a tem como destino. O Agente Estação também informa o utilizador se este pode começar nesta estação ou não, sobre a existência de necessidade de incentivos e se este pode sair do sistema.

```
1 private class ReceiveMessages extends CyclicBehaviour{
2
3 public void action(){
4   ACLMessage msg = receive();
5   if(msg != null){
```

```
6       if(msg.getPerformative() == REQUEST)
7       {
8           parseRequests(msg);
9       }else if(msg.getPerformative() == INFORM){
10          parseInforms(msg);
11      }else if(msg.getPerformative() == SUBSCRIBE){
12          parseSubscribe(msg);
13      }else if(msg.getPerformative() == CANCEL) {
14          parseCancel(msg);
15      }else if(msg.getPerformative() == ACCEPT_PROPOSAL){
16          ( ... )
17      }else if(msg.getPerformative() == REJECT_PROPOSAL){
18          ( ... )
19      }
20    }
21  }
22  (...)
23  }
```

EntregaBicicletas

Através do uso de um **TickerBehaviour**, com um *tick* de 1 segundo, vai ser feito um processamento de uma lista de espera para a entrega das bicicletas por ordem de chegada, quando a entrega é feita, um aviso é enviado ao utilizador a notificar-lhe que pode sair do sistema.

```
1   private class EntregaBicicletas extends TickerBehaviour {
2
3     public EntregaBicicletas(Agent a,long timeout){
4       super(a,timeout);
5     }
6
7     protected void onTick(){
8       int diff = capacidadeEstacao - bicicletas;
9       if(listaDeEspera.size() > 0 && diff > 0){
10        String nomeUtilizador = listaDeEspera.get(0);
11        listaDeEspera.remove(0);
12
13        AID receiver = new AID();
14        receiver.setLocalName(nomeUtilizador);
15        ACLMessage aEnviar = new CONFIRM;
16
17        aEnviar.addReceiver(receiver);
18        aEnviar.setContent("Entrega Completa!");
19        myAgent.send(aEnviar);
20
21        bicicletas++;
22
23        String fullName = myAgent.getAID().getName();
```

```
24        int index = fullName.indexOf("@");
25        String nome = fullName.substring(0,index);
26
27        double ocup = ( ... )
28
29        ocupacaoEstacao.put(nome,ocup);
30      }
31    }
32  }
```

5.5 Agente System

O Agente System é responsável pela criação de novos Agente Utilizador e ainda também na resolução de algumas queries feitas pelos utilizadores.

- *GerarUtilizadores*, behaviour responsável pela criação de novos utilizadores;
- *ReceiveMessages*, behaviour responsável por receber queries do agente Utilizador, usado especificamente para ajuda no calculo do incentivo.

GerarUtilizadores

Responsável pela criação de novos Agentes Utilizador, para isso utiliza um **TickerBehaviour** com um *tick* definido de 2000ms. Denotando que a responsabilidade de saber onde este novo agente Utilizador vai nascer é da responsabilidade do Agente Utilizador e não do Agente System, visto que o Agente System apenas faz a sua criação.

```
1  private class gerarUtilizadores extends TickerBehaviour{
2      public gerarUtilizadores(Agent a,long timeout){
3          super(a,timeout);
4      }
5      protected void onTick(){
6          String nome = "Utilizador" + numero_utilizador;
7          mc.startUtilizador(nome, mapa);
8          numero_utilizador++;
9      }
10 }
```

ReceiveMessages

CyclicBehaviour utilizado para poder comunicar com o utilizador de modo a fornecer informação que o assiste no calculo da escolha de aceitar ou recusar o incentivo proposto por uma estação.

```
private class receiveMessages extends CyclicBehaviour {
  public void action(){
    ACLMessage msg = myAgent.receive();
    if (msg != null) {
      try {
        HashMap<String,Double> estacoes =
          msg.getContentObject();

        HashMap<String, Coordenadas> toReturn =
          new HashMap<>();

        Iterator it = estacoes.entrySet().iterator();
        while(it.hasNext()){
          ( ... )
        }
        ACLMessage resposta = new ACLMessage();
        resposta.setConversationId(msg.getConversationId());
        resposta.setContentObject(toReturn);
        resposta.addReceiver(msg.getSender());
        resposta.setPerformative(ACLMessage.INFORM);
        myAgent.send(resposta);

      } catch (IOException | UnreadableException e){
      e.printStackTrace();
      }
    }
  }
}
```

5.6 Agente Interface

O Agente Interface é responsável pela atualização e funcionamento da interface.

Este inicialmente recebe da classe Mapa as posições das estações, visto que estas nunca mudam. O agente contem vários *behaviours*, dos quais:

- **PedirOcupacao**, *behaviour* responsável por pedir a ocupação a uma estação;
- **ReceiveInfo**, *behaviour* responsável por receber a toda a informação que a interface precise para mostrar ao utilizador;
- **drawOcupacao**, *behaviour* responsável por mostrar na interface as estações com as suas ocupações correspondentemente;
- **PedirUtilizadores**, *behaviour* responsável por pedir a cada estação, os utilizadores que se encontram na sua área de forma a poder desenhá-los na UI;

- **PedirBicicletas**, este serve para pedir ás estações a quantidade de bicicletas que estas têm ainda disponíveis;
- **PedirFalhas**, responsável por pedir ás estações a quantidade de falhas que estas já tiveram, ou seja, número de vezes que um utilizador queria começar numa estação e não havia bicicletas.

PedirOcupacao

Este *behaviour* do Agente Interface é responsável pelo envio de uma mensagem à primeira estação a pedir a ocupação das estações, para isso utiliza um **TickerBehaviour** com um *tick* definido em 1.5 segundos. Este só pede informação à primeira estação pois cada estação tem um *ConcurrentHashMap* partilhado por todas as estações, em que neste tem as respectivas taxas de ocupação.

```
public class PedirOcupacao extends TickerBehaviour {

    public PedirOcupacao(Agent a, long timeout){
        super(a,timeout);
    }

    protected void onTick(){
        try {

            DFAgentDescription dfd= new DFAgentDescription();
            ServiceDescription sd = new ServiceDescription();
            sd.setType("Estacao");
            dfd.addServices(sd);

            DFAgentDescription[] resultados =
              DFService.search(myAgent, dfd);

            DFAgentDescription dfd1 = resultados[0];
            AID estacao = dfd1.getName();

            String mensagem ="Ocupacao";

            ACLMessage aEnviar = new ACLMessage(REQUEST);
            aEnviar.addReceiver(estacao);
            aEnviar.setContent(mensagem);
            myAgent.send(aEnviar);

        }
        catch (FIPAException fe){
            fe.printStackTrace();
        }
    }
}
```

ReceiveInfo

O *behaviour* **ReceiveInfo** utiliza um **CyclicBehaviour** que espera pela informação pedida ás estações através da confirmação de recepção de uma *performative Inform*. Estas mensagens recebidas podem conter informação relativamente a: taxa de ocupação das estações, número de bicicletas que cada estação tem, número de falhas que cada estação já teve até ao momento, e a posição dos utilizadores. Também é neste *behaviour* que será decidido se já é o momento de a UI atualizar informação ou não, isto é, uma vez que são recebidas mensagens separadas de cada estação com informação, é preciso saber quantas mensagens foram recebidas com tal informação de forma a não haver informação incompleta e quando já temos a informação toda podemos passar a atualizar a UI. Por exemplo, cada estação manda uma mensagem com as suas bicicletas disponíveis e temos 25 estações, é necessário ter um contador que a cada mensagem de estação incremente um contador e quando este chegar a 25 significa que recebeu tudo e pode atualizar a UI.

```
private class ReceiveInfo extends CyclicBehaviour {
  public void action() {

    if (contagem >= estacoes) {
      ui.drawUtilizadores(posicaoUtilizadores);
      posicaoUtilizadores.clear();
      contagem = 0;
    }
    if (contagem_graph_1 >= estacoes){
      ui.drawBicicletas(bicicletas);
      contagem_graph_1 = 0;
    }
    if(contagem_graph_2 >= estacoes){
      ui.drawFalhas(falhas);
      contagem_graph_2 = 0;
    }
    ACLMessage msg = receive();
    if (msg != null && msg.getPerformative() == INFORM) {
      if (msg.getContent().contains("Estacao")) {
        ocupacaoEstacao = msg.getContent();
      }
      else if(msg.getContent().contains("Bic")){
        contagem_graph_1++;
        (...)
      }
      else if(msg.getContent().contains("Fail")){
        contagem_graph_2++;
        (...)
      }
      else if(msg.getContent() != null){
        contagem++;
        String str = msg.getContent();
```

```
        str = str.replaceAll("[\n]+", " ");
        String posicoes[] = str.split(" ");

        for (i < posicoes.length && posicoes.length > 1){
          if (posicoes == null) break;

          if ( ... ){
            Coordenadas c =
              new Coordenadas(posicoes[i], posicoes[i+1]);

            posicaoUtilizadores.add(c);
          }
        }
      }
    }
  }
}
```

drawOcupacao

O *behaviour* **drawOcupacao** utiliza um **TickerBehaviour** com um *tick* de 1.5 segundos. Este, através da informação já obtida por parte do Agente Interface, irá atualizar a UI com a informação da taxa de ocupação de cada estação.

```
private class drawOcupacao extends TickerBehaviour {

    public drawOcupacao(Agent a, long timeout){
        super(a,timeout);
    }

    protected void onTick(){
        ui.drawOcupacaoEstacao(ocupacaoEstacao);
    }
}
```

```
 public void drawOcupacaoEstacao(String ocupacoes){

        texto.setText(ocupacoes);
        panel_1.add(texto);
        panel_1.revalidate();

    }
```

PedirUtilizadores

O *behaviour* **PedirUtilizadores** utiliza um **TickerBehaviour** com um *tick* de 1 segundo. Este serve para o Agente Interface pedir aos Agentes Estação os utilizadores que se encontram na sua área de controlo.

```
public class PedirUtilizadores extends TickerBehaviour {
    public PedirUtilizadores(Agent a, long timeout){
        super(a,timeout);
    }

    protected void onTick(){
        try {
            (...)
            DFAgentDescription[] resultados = DFService.
    search(myAgent, dfd);

            for (int i = 0; i < resultados.length; i++) {
                DFAgentDescription dfd1 = resultados[i];
                AID estacao = dfd1.getName();

                String mensagem = "Utilizadores";

                ACLMessage aEnviar = new ACLMessage(
    ACLMessage.REQUEST);
                aEnviar.addReceiver(estacao);
                aEnviar.setContent(mensagem);
                myAgent.send(aEnviar);
            }
        }
        catch (FIPAException fe){
            fe.printStackTrace();
        }
    }
}
```

PedirBicicletas

O *behaviour* **PedirBicicletas** utiliza um **TickerBehaviour** com um *tick* de 3 segundos. Este é usado para pedir informação, a cada Agente Estação, relativamente ao número de bicicletas que cada um tem disponível de momento.

```
public class PedirBicicletas extends TickerBehaviour {

    public PedirBicicletas(Agent a, long timeout){
        super(a,timeout);
    }

    protected void onTick(){
        try {
            (...)
            DFAgentDescription[] resultados = DFService.
    search(myAgent, dfd);

            for (int i = 0; i < resultados.length; i++) {
                DFAgentDescription dfd1 = resultados[i];
                AID estacao = dfd1.getName();

                String mensagem = "Bicicletas";

                ACLMessage aEnviar = new ACLMessage(
    ACLMessage.REQUEST);
                aEnviar.addReceiver(estacao);
                aEnviar.setContent(mensagem);
                myAgent.send(aEnviar);
            }
        }
        catch (FIPAException fe){
            fe.printStackTrace();
        }
    }
}
```

PedirFalhas

O *behaviour* **PedirFalhas** utiliza um **TickerBehaviour** com um *tick* de 6 segundos. Este é usado para pedir informação, a cada Agente Estação, relativamente ao número de falhas que este já teve até ao momento. Como dito anteriormente, uma falha dá-se quando um utilizador quer começar numa estação e a mesma não tem bicicletas disponíveis. Cada estação guarda as suas próprias falhas.

```
1  public class PedirFalhas extends TickerBehaviour {
2      public PedirFalhas(Agent a, long timeout){
3          super(a,timeout);
4      }
5
6      protected void onTick(){
7          try {
8              (...)
9              DFAgentDescription[] resultados = DFService.
       search(myAgent, dfd);
10
11             for (int i = 0; i < resultados.length; i++) {
12                 DFAgentDescription dfd1 = resultados[i];
13                 AID estacao = dfd1.getName();
14
15                 String mensagem = "Falhas";
16
17                 ACLMessage aEnviar = new ACLMessage(
       ACLMessage.REQUEST);
18                 aEnviar.addReceiver(estacao);
19                 aEnviar.setContent(mensagem);
20                 myAgent.send(aEnviar);
21             }
22         }
23         catch (FIPAException fe){
24             fe.printStackTrace();
25         }
26     }
27 }
```

6 Interface

Nesta secção será apresentado o resultado final da interface, assim como os métodos desenvolvidos para implementação dos gráficos que decidimos utilizar para analisar as simulações que são efectuadas. Todos os gráficos foram feitos utilizando **JFreeChart**.

6.1 Interface de Simulação

A interface que permite correr as simulações e visualizar as diferentes interações entre os diferentes agentes encontra-se representada na figura 6.1. Esta foi desenvolvida com recurso a **Swing**. Quando este sistema é iniciado, é pedido ao utilizador para escolher entre os tamanhos de mapa disponíveis, bem como o número de estações que este queira, no qual o número de estações total será o número de estações escolhido ao quadrado, e também pode escolher o número de bicicletas que cada estação começa.

Após a decisão destes 3 parâmetros, é apresentada a UI principal que consiste numa visualização do mapa com as estações e utilizadores, também é apresentado no canto superior direito desta as taxas de ocupações das estações e no canto inferior direito encontram-se 2 botões, um dos botões serve para a UI apresentar o gráfico feito usando **JFreeChart** que representa o número de bicicletas disponíveis em cada estação, o outro botão serve para mostrar o número de falhas de cada estação em que também apresenta um gráfico feito em **JFreeChart** como auxílio para a visualização das mesmas.

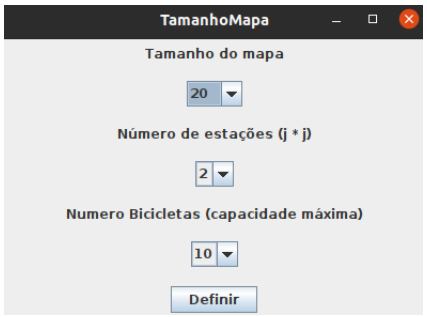

Figura 8. Menu inicial para escolha de parâmetros.

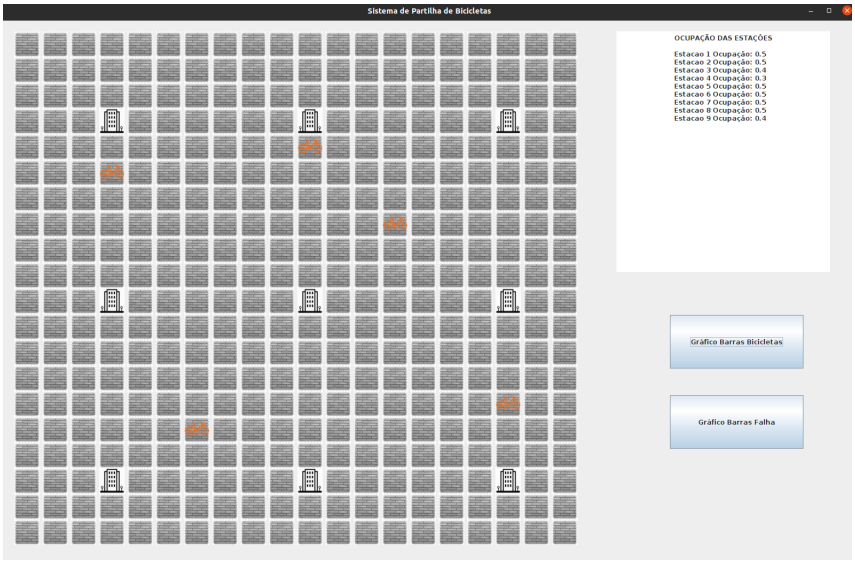

Figura 9. Interface da simulação.

6.2 Gráfico de barras com distribuição de bicicletas

Quando o utilizador carrega no botão "Gráfico Barras Bicicletas", este consegue visualizar uma nova *frame* no qual será apresentado um gráfico de barras com a informação de bicicletas disponíveis de cada estação. Em seguida será apresentado o código do mesmo, bem como a visualização de um gráfico de barras para um número de estações igual a 9.

```
1  public void drawBicicletas(int bicicletas[]){
2      DefaultCategoryDataset dataset = new
       DefaultCategoryDataset();
3
4      for(int i = 0 ; i < (mapa.getEstacoes() * mapa.
       getEstacoes()) ; i++) {
5          dataset.addValue(bicicletas[i],"Estacao",""+(i+1));
6      }
7
8      JFreeChart chart=ChartFactory.createBarChart(
9              "Disposicao de Bicicletas", //Chart Title
10             "Estacao", // Category axis
11             "Bicicletas", // Value axis
12             dataset,
13             PlotOrientation.VERTICAL,
14             false,false,false
15     );
16
17     ChartPanel CP = new ChartPanel(chart);
```

```
18      graficoBicicletas.add(CP,BorderLayout.CENTER);
19      histogramaBicicletas.add(graficoBicicletas);
20      graficoBicicletas.validate();
21      histogramaBicicletas.validate();
22  }
```

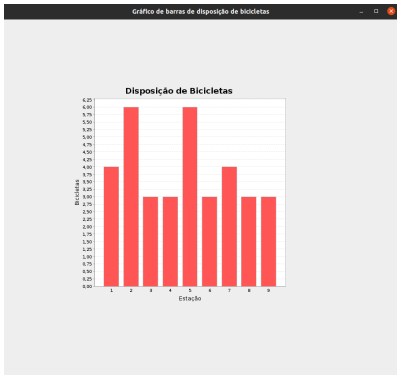

Figura 10. Gráfico de barras com disposição de bicicletas.

6.3 Gráfico de barras com o número de falhas

Caso o utilizador queira visualizar o número de falhas correspondente a cada estação, basta clicar no botão "Gráfico Barras Falha"e será apresentado um gráfico de barras com o número de falhas. De seguida será apresentado o código do gráfico feito com o auxílio de **JFreeChart**, bem como a visualização do mesmo para um número de estações igual a 9.

```
1  public void drawFalhas(int falhas[]){
2      DefaultCategoryDataset dataset = new
       DefaultCategoryDataset( );
3
4      for(int i = 0 ; i < (mapa.getEstacoes() * mapa.
       getEstacoes()) ; i++) {
5          dataset.addValue(falhas[i],"Estacao",""+(i+1));
6      }
7      JFreeChart chart=ChartFactory.createBarChart(
8              "Falta de bicicletas", //Chart Title
9              "Estacao", // Category axis
10             "Vezes", // Value axis
11             dataset,
12             PlotOrientation.VERTICAL,
13             false,false,false
14     );
```

```
15
16      ChartPanel CP = new ChartPanel(chart);
17      graficoFalhas.add(CP,BorderLayout.CENTER);
18      graficoFail.add(graficoFalhas);
19      graficoFalhas.validate();
20      graficoFail.validate();
21  }
```

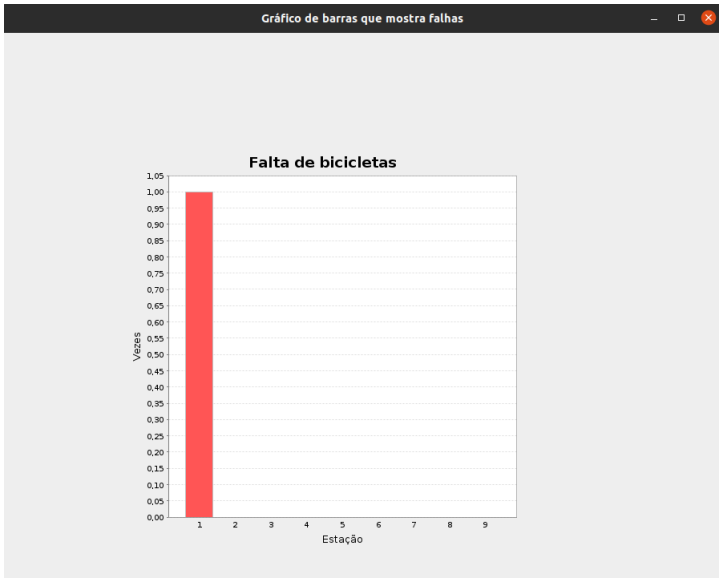

Figura 11. Gráfico de barras com falhas.

7 Análise de Simulação

Para uma melhor noção do que foi implementado em relação ao incentivo, foram feitos dois testes, um em que o utilizador recebe incentivos e outro sem incentivos, como se segue nas seguintes imagens:

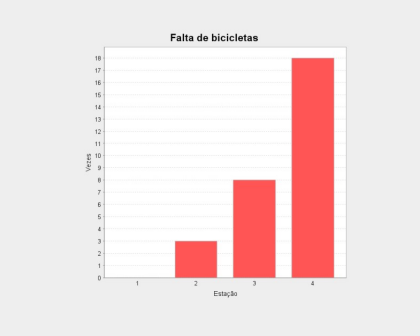

Figura 12. Gráfico de barras com falhas sem incentivo

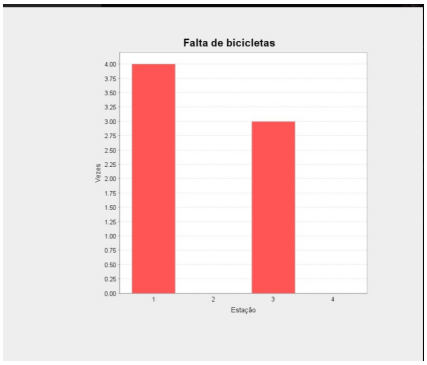

Figura 13. Gráfico de barras com falhas com incentivo

Foram feitos dois testes, de cerca de 3 minutos ao sistema com um mapa 20*20, 4 estações e 10 bicicletas de máximo onde havia um novo utilizador a cada segundo. O que permitiu verificar que quando existe incentivo o número de falhas de bicicletas nas estações é muito menor do que quando não existe incentivo.

8 Conclusão e perspectiva de trabalho futuro

Na primeira fase do trabalho foi apresentada uma contextualização do nosso sistema multi agente (SMA) bem como a sua modelação.

Com o objectivo de esta modelação ser a arquitectura final do nosso sistema, foram identificados e explicados certos aspectos que futuramente na parte 2 do projecto ajudaram bastante na implementação do mesmo. Estes aspectos vão desde: explicação das classes a implementar, tendo sempre em conta os *behaviours* que indicarão o comportamento dos agentes, explicação dos diagramas que mostram tudo que deve ser implementado numa versão final e a comunicação entre agentes.

Este sistema, numa segunda fase do projecto, foi implementado usando JADE em que na sua comunicação são usadas mensagens com a especificação FIPA(1) e através de uma interface o utilizador é capaz de interagir e definir as suas intenções com o sistema, como previsto na primeira fase do projecto.

Sendo um dos maiores problemas dos SPB o reequilíbrio do sistema devido á natural maior ocupação de bicicletas em determinadas estações, o que acaba por provocar congestionamento e um mau alocamento de recursos e tendo como finalidade a implementação de um SMA para um SPB, foi implementado um SMA que através de incentivos leva os utilizadores a optarem por uma nova estação de destino se a inicial estiver demasiado ocupada. Com a aplicação destes incentivos foi possível verificar que esse reequilíbrio aconteceu, sendo que a ocupação é em média 50% em todas as estações num futuro finito. O mesmo não garante que numa certa iteração esses valores variem, mas futuramente o algoritmo implementado garante um reequilíbrio total das estações.

Futuramente, o grupo considera que seria possível fazer uma análise mais aprofundada da informação dos utilizadores e dos gráficos gerados pelo sistema em relação á ocupação e falhas de cada estação, o que permitiria uma análise em modelos preditivos de qual o número de bicicletas e estações ideal. Além disso seria possível ainda tentar desenvolver outro tipo de incentivo ou diminuição de recursos em algumas estações para permitir um melhor funcionamento do sistema e consequentemente um menor número de falhas.

Bibliografia

[1] FIPA website, http://www.fipa.org/, last visited 28 November 2020.

[2] FIPA-Performatives,http://jmvidal.cse.sc.edu/talks/agentcommunication/performatives.html?style=White, last visited 28 November 2020.

[3] JADE website, https://jade.tilab.com/, last visited 9 December 2020.

[4] Bauer, B., Müller, J. P., & Odell, J. (2001). Agent UML: A formalism for specifying multiagent software systems. International Journal of Software Engineering and Knowledge Engineering, 11(3), 207–230

[5] JfreeChart website, https://www.jfree.org/jfreechart/, last visited 15 December 2020