Guia de Programação JSensor

Exemplo Mobile Phone

Sumário

1 Introdução

JSensor é um simulador para redes de sensores sem fio de larga escala, que pode ser executado, eficientemente, em arquiteturas de computadores com memória compartilhada ou arquiteturas multicore (para mais informações sobre memória compartilhada, visite https://en.wikipedia.org/wiki/Shared_memory

O arquivo JSensorInstallationGuide.pdf servirá como um guia de instalação para o JSensor e também o ajudará a criar um projeto padrão neste simulador, que contém um conjunto de classes e métodos que servirão como base para o desenvolvimento de um modelo de simulação mais específico.

Exemplos de modelos de simulações para o JSensor podem ser encontrados no diretório **jsensor/src/projects**. Neste documento, será demonstrado um exemplo que simula uma rede de telefonia móvel de modo assíncrono. Sendo assim, foram incluídas algumas características não vistas no modelo flooding, tais como mobilidade, perda de mensagens, criação de fatos, distribuição em grade e múltiplos modelos de nós. Tal modelo encontra-se no diretório **jsensor/src/projects/mobilephone**.

Este guia abordará cada uma das classes que compõem o modelo Mobile Phone.

2 MyCell

O JSensor divide a área de simulação em células, o que otimiza a tarefa de busca de vizinhança. Além disso, usuários podem criar seu próprio tipo de célula, contendo características próprias como: nível de poluição, temperatura, umidade, etc. Para isso o desenvolvedor deve estender a classe DefaultJSensorCell do JSensor.

Neste exemplo, uma célula é representada pela classe **MyCell** e, com a ajuda de seus campos **thunder** (boolean) e **water** (float), será possível controlar o ambiente de simulação.

Classes que estendem **DefaultJSensorCell** devem sobrescrever seu método **clone**, que retorna um novo objeto com atributos idênticos aos do objeto original.

```
5   public class MyCell extends DefaultJsensorCell{
6           private boolean thunder;
7           private Float water;
8
9           public Float getWater() {
10                  return water;
11          }
12
13          public void setWater(Float water) {
14                  this.water = water;
15          }
16
17          public MyCell(){
18                  this.thunder = false;
19                  this.water = new Float(0);
20          }
21
22          public MyCell(boolean thunder, Float water){
23                  this.thunder = thunder;
24                  this.water = water;
```

```
25
26          }
27
28          public boolean getThunder() {
29                  return this.thunder;
30          }
31
32          public void setThunder(boolean thunder) {
33                  this.thunder = thunder;
34          }
35
36
37          @Override
38          public DefaultJsensorCell clone() {
39                  return new MyCell(this.thunder, this.water);
40          }
41  }
```

Ao criar células e fatos, o usuário consegue simular alterações no ambiente de simulação. No exemplo mostrado, o campo **thunder** indica se a célula foi atingida por um raio, o que pode desativar alguns dos nós do tipo Antenna que habitam a célula. Já o atributo **water**, informa o nível de água naquela célula, simulando a ocorrência de tempestade.

3 CellPhone

Esta classe define um nó que é uma abstração de um celular. Assim como qualquer tipo de nó criado no JSensor, é necessário estender a classe **Node**. A classe **CellPhone** possui dois atributos: **int myAntenna** e **LinkedList<Long>messagesIDs**. O primeiro representa a antena à qual o celular está conectado, enquanto o segundo armazena os identificadores de mensagens já recebidas pelo nó.

```
18  public class CellPhone extends Node{
19
20      private int myAntennaID;
21      private LinkedList<Long> messagesIDs;
```

No momento da criação de um nó, o método **onCreation()** é chamado.

```
66      @Override
67      public void onCreation()
68      {
69          //inicializa a lista de mensagens recebidas pelo nó
70          this.messagesIDs = new LinkedList<Long>();
71          this.myAntennaID = -1;
72          //envia uma mensagem caso o nó seja um dos selecionados
73          if(this.getRandom().nextDouble() < 0.4){
74              int time = 10 + this.getRandom().nextInt(100);
75              SMSTimer st = new SMSTimer();
76              st.startRelative(time, this);
77              CellPhoneMobilityTimer ct = new CellPhoneMobilityTimer();
78              ct.start(this, 1, 5000, 10);
79          }
80      }
```

Neste exemplo, sua função é inicializar a lista **messagesIDs** (linha 70), atribuir o valor -1 ao campo **myAntenna** (linha 71) e, eventualmente, criar um evento de envio de mensagem. O método **getRandom().nextDouble()**, da classe **Node** do JSensor, retorna um número double que varia entre 0 e 1.0,

a partir da semente padrão do JSensor ou de outra, definida pelo usuário. Observe que o evento de envio de mensagem só será criado se o valor retornado por esse método for menor que 0.4, o que faz com que a probabilidade de criação do evento seja de 40%.

O método **getRandom().nextInt()** (linha 74) é usado para definir o tempo no qual o evento será disparado. O valor 100, que é o argumento desse método, indica que será gerado um valor inteiro entre 0 e 99. O tempo exato de envio da mensagem será o resultado da soma entre o valor inteiro retornado por **getRandom().nextInt()** e 10.

Na próxima linha, a criação do evento propriamente dito é realizada. Primeiramente, uma instância da classe SMSTimer é criada (linha 75). Após isso, é necessário fazer a chamada ao método **startRelative()**, que fará com que a mensagem seja enviada no tempo definido.

Além do evento que representa o envio de um SMS, também foi criado um objeto do tipo **CellPhoneMobilityTimer**, que simulará a mobilidade do celular (linha 77). Na próxima linha, a chamada ao método **start()** define a criação de um evento de mobilidade. Seus quatro argumentos indicam, respectivamente, o nó que irá mover-se, em qual unidade de tempo o movimento será iniciado, quanto tempo durará o movimento do nó e o intervalo de repetição de cada movimento. A classe **CellPhoneMobilityTimer** será detalhada na seção 8. Por enquanto, só é necessário saber como instanciá-la e utilizá-la.

Obs.: Como **onCreation()** é executado durante a criação dos nós, não é recomendado fazer referências diretas a outros nós dentro desse método, pois é possível que tais nós ainda nem tenham sido criados, o que causaria erros na simulação. Por isso, utilize objetos do tipo **timer**, devidamente inicializados, para realizar qualquer tipo de operação com outros nós.

```
23      @Override
24      public void handleMessages(Inbox inbox) {
25        while(inbox.hasMoreMessages())
26        {
27            Message m = inbox.getNextMessage();
28
29            if(m instanceof SMSMessage){
30                if(this.messagesIDs.contains(((SMSMessage)m).getID())){
31                    continue;
32                }
33                else
34                    this.messagesIDs.add(((SMSMessage) m).getID());
35
36                if(((SMSMessage) m).getDestination().getID() == this.ID){
37                    // registra recebimento da mensagem em arquivo
38                    Jsensor.log("time: "+ Jsensor.currentTime +" nodeID: " +this.ID+
39                                " receivedFrom: " +((SMSMessage)m).getSender().getID()+
40                                " hops: "+ ((SMSMessage)m).getHops());
41
42                    // probabilidade de responder mensagem recebida é de 0.5
43                    if(this.getRandom().nextDouble() < 0.5){
44                          int time = 1 + this.getRandom().nextInt(50);
45                          SMSTimer ct = new SMSTimer();
46                          ct.startRelative(time, this);
47                    }
48                }
49            }
50        }
51      }
```

Outro método importante que deve ser sobrescrito por **CellPhone** é **handleMessages()**, cujo papel é realizar o controle das mensagens recebidas pelo nó. Esse método possui um único parâmetro, a inbox do nó, ou seja, as mensagens recebidas por esse nó e que devem ser processadas. Para cada mensagem na caixa de entrada de um nó, **handleMessages()** verifica se ela é do tipo SMSMessage (linha 29) e se é uma mensagem nova (linha 30), ou seja, ainda não faz parte da lista de mensagens recebidas pelo nó. Caso a mensagem seja aprovada nesses testes, **handleMessages()** faz um processamento adicional, que tem por objetivo determinar se o destinatário desta mensagem é ou não o nó corrente (linha 36). Em caso positivo, o recebimento da mensagem é registrado em um arquivo de log.

O método **Jsensor.log()**, conforme visto na linha 38, serve para popular o arquivo de log do simulador JSensor. A geração desse arquivo de log é opcional e deve ser feita pelo usuário (definir o parâmetro **log** como **true**, no arquivo de configuração do modelo) antes do início da execução do modelo. Caso o usuário opte pela geração de log, tais logs serão criados no diretório **results**.

```
58      public void findAntenna()
59      {
60          if(this.getNeighbours().getNodesList().isEmpty())
61              myAntennaID = -1;
62          else
63              myAntennaID = this.getNeighbours().getNodesList().first().getID();
64      }
```

O método **findAntenna()** define à qual das antenas um nó irá se conectar.

```
53      public int getMyAntenna()
54      {
55          return myAntennaID;
56      }
```

E, por fim, o método **getMyAntenna()**, que retorna à qual das antenas um nó está conectado.

4 Antenna

A classe **Antenna** define um nó que representa as antenas de uma rede de telefonia móvel e, assim como **CellPhone**, deve estender a classe **Node** do JSensor. Ela possui apenas um atributo, **messagesIDs**, que armazena a lista de identificadores de mensagens já recebidas pelo nó.

```
15  public class Antenna extends Node{
16
17      private LinkedList<Long> messagesIDs;
```

Neste exemplo, a única função do método **onCreation()**, sobrescrito por **Antenna** é instanciar o atributo que representa a lista de mensagens recebidas pelo nó.

```
46      @Override
47      public void onCreation()
48      {
49          //inicializa a lista de mensagens recebidas pelo nó
```

```
50              this.messagesIDs = new LinkedList<Long>();
51          }
```

O método **handleMessages()**, conforme visto na seção anterior, realiza o controle das mensagens recebidas pelo nó. O primeiro teste a ser feito aqui é se a antena em questão está ativa, isto é, se a célula na qual a antena está localiza não foi atingida por um raio ou não está coberta pela água. Para isso, o atributo **isActive** (da classe **Node** do JSensor), deve ser verificado.

```
19          @Override
20          public void handleMessages(Inbox inbox) {
21              if (!this.isActive())
22                  return;
23
24              SetFactCell setThunder = new SetFactCell();
25              setThunder.addCell(this, this.position);
26
27              while(inbox.hasMoreMessages())
28              {
29                  Message m = inbox.getNextMessage();
30
31                  if(m instanceof SMSMessage)
32                  {
33                      if(this.messagesIDs.contains(((SMSMessage)m).getID())){
34                          continue;
35                      }else
36                      {
37                          this.messagesIDs.add(((SMSMessage)m).getID());
38                          ((SMSMessage) m).setMessage(((SMSMessage) m).getMessage().concat(Integer.
39                              toString(this.ID)));
40                          this.multicast(m);
41                      }
42                  }
43              }
44          }
```

Então, caso a antena não esteja em boas condições de funcionamento, não há motivo para **handleMessages()** realizar processamento algum e ele simplesmente termina sua execução. Caso contrário, será possível verificar sua caixa de entrada, em busca de novas mensagens que devem ser encaminhadas. Porém, antes de realizar tal verificação, decidimos fazer com a célula que abriga esta antena seja alterada por um fato (evento). Para isso, é feita uma chamada ao método addCell(), da classe **SetFactCell**, explicada na próxima seção do documento. Alterações nas células que formam a área de simulação devem ser feitas dessa forma. Isso é necessário para que o simulador possa manter a reprodutibilidade das alterações realizadas nas células. Caso o usuário alterasse diretamente uma célula, essa característica poderia ser perdida. Então, ao invés de realizar uma alteração imediatamente, a chamada ao método citado logo acima estipula a ordem de alteração das células, com a ajuda do uso de uma estrutura de dados.

5 SetFactCell

Classe que estende **HandleCells** e implementa seu método **fire()**, cujo objetivo é verificar se a célula que abriga o nó que foi recebido como parâmetro foi atingida por um raio ou se seu nível de água está acima de um determinado valor. Em caso positivo, o nó que percebeu essa alteração na condição da

célula é desativado, por algumas unidades de tempo. Neste exemplo, apenas o primeiro nó que percebeu a incidência de raio na célula é desativado, já que o valor do campo **thunder** é alterado para **false**.

```
8   public class SetFactCell extends HandleCells {
9
10      @Override
11      public void fire(Node node, CellModel cell) {
12          cell = (MyCell) Jsensor.getCell(node.getPosition());
13          if(((MyCell)cell).getThunder()){
14              ((MyCell)cell).setThunder(false);
15              Jsensor.setCell(cell);
16              node.deactivate(node.getRandom().nextInt(100));
17
18              Jsensor.log(">>>>> Antenna "+node.getID()+" struck by Thunder!");
19          }
20
21          if(((MyCell)cell).getWater() > 120){
22              node.deactivate(node.getRandom().nextInt(100));
23
24              Jsensor.log(">>>>> Antenna "+node.getID()+" is full of water!");
25          }
26      }
27
28  }
```

6 SMSMessage

As mensagens enviadas pelos nós são objetos do tipo **SMSMessage**.

Essa classe possui cinco atributos, que representam os nós remetente e destinatário da mensagem, seu texto, a quantidade de saltos realizadas por ela e um identificador, único, de mensagem.

```
10  public class SMSMessage extends Message{
11
12      private Node sender;
13      private Node destination;
14      private int hops;
15      private String message;
16      public static int idNumbers = 0;
```

Métodos de acessos são fornecidos para cada um dos quatro primeiros atributos citados.

```
34      public Node getDestination() {
35          return destination;
36      }
37
38      public void setDestination(Node destination) {
39          this.destination = destination;
40      }
41
42      public int getHops() {
43          return hops;
44      }
45
46      public void setHops(int hops) {
47          this.hops = hops;
48      }
```

```
49
50      public Node getSender() {
51          return sender;
52      }
53
54      public void setSender(Node sender) {
55          this.sender = sender;
56      }
57
58      public String getMessage() {
59          return message;
60      }
61
62      public void setMessage(String message) {
63          this.message = message;
64      }
```

Nos modelos criados para este simulador, classes que implementam mensagens devem estender a classe **Message** do JSensor e sobrescrever seu método **clone**.

```
66      @Override
67      public Message clone() {
68          return new SMSMessage(sender, destination, hops+1,this.getID(), this.message);
69      }
```

Note que o método **clone()** faz uso do construtor de cópia da classe **SMSMessage** (segundo dos construtores disponibilizados por ela), criando uma cópia idêntica da mensagem. Isso é importante para que não haja problemas de concorrência entre os processos, durante a manipulação de mensagens.

```
18      public SMSMessage(Node sender, Node destination, int hops, short chunck) {
19          super(chunck);
20          this.sender = sender;
21          this.destination = destination;
22          this.hops = hops;
23          this.message = "";
24      }
25
26      public SMSMessage(Node sender, Node destination, int hops, long ID, String message) {
27          this.setID(ID);
28          this.sender = sender;
29          this.destination = destination;
30          this.hops = hops;
31          this.message = message;
32      }
```

7 SMSTimer

Para realizar o envio de uma mensagem, o modelo Mobile Phone faz uso de objetos do tipo SMSTimer. Como trata-se de um timer, é necessário estender a classe **TimerEvent** do JSensor e sobrescrever seu método **fire()**.

Na seção 3 (CellPhone), foi dito que a criação de um evento de timer ocorre quando, dada um instância de SMSTimer, o método startRelative() é invocado. Conforme visto naquela seção, este método admite dois argumentos.

```
75                SMSTimer st = new SMSTimer();
76                st.startRelative(time, this);
```

O primeiro deles indica o tempo no qual o evento será disparado, enquanto o segundo indica qual o nó que está vinculado ao evento em questão.

Uma vez que o evento tenha sido criado, quando o tempo utilizado em sua criação for alcançado, o método **fire()** será acionado. Este método será responsável por realizar o envio da mensagem.

```
13  public class SMSTimer extends TimerEvent{
14
15      @Override
16      public void fire() {
17          CellPhone aux = (CellPhone) this.node;
18          aux.findAntenna();
19          if(aux.getMyAntenna() != -1){
20              Node n;
21              n = this.node.getRandomNode("CellPhone");
22              while(true){
23                  if(n == null) {
24                      n = this.node.getRandomNode("CellPhone");
25                      continue;
26                  }
27
28                  if(this.node == n){
29                      n = this.node.getRandomNode("CellPhone");
30                      continue;
31                  }
32                  break;
33              }
34
35              SMSMessage p = new SMSMessage(node, n, 0, this.node.getChunk());
36              p.setMessage("This is the message number " + this.node.getChunk() +
37                      " created by the node " + node.getID() + " path");
38              Jsensor.log("* time: " + Jsensor.currentTime + " nodeID: " + this.node.getID() +
39                      " sendTo: " + n.getID() + " byAntenna: "+
40                      aux.getMyAntenna());
41              this.node.unicast(p, Jsensor.getNodeByID(aux.getMyAntenna()));
42          }
43          else{
44              Jsensor.log("[No signal] time: " + Jsensor.currentTime +
45                      " nodeID: " + this.node.getID());
46              Jsensor.log("[No signal] position: " + this.node.getPosition().getPosX() +
47                      ", " + this.node.getPosition().getPosY());
48          }
49      }
50  }
```

Inicialmente, é verifica se o nó que deseja enviar a mensagem possui sinal, ou seja, está conectado a um nó do tipo **Antenna**. Em caso positivo, o nó destinatário é escolhido, aleatoriamente (linha 21), a mensagem a ser enviada é preparada (linhas 35 e 36) e é então enviada. O método também registra, em arquivo de log, o envio dessa mensagem (linhas 38, 39 e 40). Caso contrário, o método apenas registra, também no arquivo de log, o fato de que o nó não possui sinal, assim como sua localização.

8 CellPhoneMobilityTimer

A classe **CellPhoneMobilityTimer** define o modelo de mobilidade utilizado pelo Mobile Phone. Sendo assim, essa classe deve estender a classe **MobilityModel** do JSensor e implementar seus métodos **clone()** e **getNextPosition()**. Este, retorna qual deve ser a próxima posição ocupada por um nó (parâmetro do método). Já aquele, retorna uma cópia do nó corrente. Com esses métodos sobrescritos, será possível criar qualquer modelo de mobilidade.

```
22      @Override
23      public MobilityModel clone() {
24          return new CellPhoneMobilityTimer(this.nodes);
25      }
26
27      @Override
28      public Position getNextPosition(Node n)
29      {
30          return nextMoviment(n);
31      }
```

No exemplo desse documento, o movimento de um dado nó é calculado por **nextMoviment()**, método acionado por **getNextPosition()**.

```
33      public Position nextMoviment(Node n){
34          int x = n.getPosition().getPosX();
35          int y = n.getPosition().getPosY();
36          int direction, newDirection = 0;
37
38          if(nodes.containsKey(n.getID())){
39              direction = nodes.get(n.getID());
40
41              //if is stopped.
42              if(direction == 8){
43                  //probability of remain stopped = 0.8
44                  if(n.getRandom().nextDouble() < 0.8)
45                      return n.getPosition();
46              }
47              //if is walking.
48              else{
49                  //probability of stop = 0.2
50                  if(n.getRandom().nextDouble() < 0.2){
51                      nodes.put(n.getID(), 8);
52                      return n.getPosition();
53                  }
54              }
55
56              //probability of change the moviment
57              if(n.getRandom().nextDouble() > 0.8){
58                  do{
59                      newDirection = n.getRandom().nextInt(8);
60
61                  }while(direction == newDirection);
62              }
63          }
64
65          direction = needChange(n, x, y, newDirection);
66
67          nodes.put(n.getID(), direction);
68
69          switch(direction){
70              case 0:{
71              return new Position(n.getPosition().getPosX(), n.getPosition().getPosY() + 1);}
```

```
72                case 1:{
73                return new Position(n.getPosition().getPosX() + 1, n.getPosition().getPosY() + 1);}
74                case 2:{
75                return new Position(n.getPosition().getPosX() + 1, n.getPosition().getPosY());}
76                case 3:{
77                return new Position(n.getPosition().getPosX() + 1, n.getPosition().getPosY() - 1);}
78                case 4:{
79                return new Position(n.getPosition().getPosX(), n.getPosition().getPosY() - 1);}
80                case 5:{
81                return new Position(n.getPosition().getPosX() - 1, n.getPosition().getPosY() - 1);}
82                case 6:{
83                return new Position(n.getPosition().getPosX() - 1, n.getPosition().getPosY());}
84                case 7:{
85                return new Position(n.getPosition().getPosX() - 1, n.getPosition().getPosY() + 1);}
86            }
87            return null;
88        }
```

Nesse modelo, um nó pode movimentar-se em oito direções diferentes. Esse movimento será determinado pelo valor da variável **direction**, de acordo com a relação abaixo:

1. O valor da coordenada x do nó é mantido e o valor da coordenada y é incrementado em uma unidade
2. O valor de ambas as coordenadas é incrementado em uma unidade
3. O valor da coordenada x do nó é incrementada em uma unidade e o valor da coordenada y é mantido
4. O valor da coordenada x do nó é incrementada em uma unidade e o valor da coordenada y é decrementada em uma unidade
5. O valor da coordenada x do nó é mantido e o valor da coordenada y é decrementado em uma unidade
6. O valor de ambas as coordenadas é decrementado em uma unidade
7. O valor da coordenada x é decrementado em uma unidade e o valor da coordenada y é mantido
8. O valor da coordenada x é decrementado em uma unidade e o valor da coordenada y é incrementado em uma unidade

Existe ainda o valor 8, que indica que o nó em questão está parado.

Objetos do tipo **CellPhoneMobilityTimer** possuem um atributo que mapeia um número identificador de um nó à direção do último movimento do nó. Caso o nó recebido como parâmetro já faça parte dessa estrutura de dados, sua direção é obtida. Nesse modelo, nós que estejam parados (linha 42) possuem 80% de chance de permanecer parado (linha 44). Nesse caso, o método **nextMoviment** apenas retorna a posição atual do nó (linha 45). Caso o nó esteja em movimento, possui, nesse exemplo, apenas 20% de cessar seu movimento (linha 50). O bloco de código definido entre as linhas 58 e 61 é responsável por determinar, de forma aleatória (mas limitada a um ser intervalo), um novo valor para **direction**.

Caso uma nova direção tenha sido determinada, a nova posição para o nó deverá ser calculada e então retornada pelo método, mas, antes disso, algumas precauções devem ser tomadas, já que o nó não poderá mover-se para um ponto fora da área de simulação. O método **needChange()** é responsável por realizar o controle dessa movimentação, ajustando a nova direção do movimento, quando necessário.

```
90      public int needChange(Node n, int x, int y, int direction){
91          //checks if the position have to change
92          if(x == 0){
93              if(y == 0){
94                  direction = n.getRandom().nextInt(3);
95              }
96              else if(y == Jsensor.getDimY()){
97                  direction = 3 + n.getRandom().nextInt(3);
98              }
99              else{
100                 if(direction >= 5 && direction <=7){
101                     direction = n.getRandom().nextInt(5);
102                 }
103             }
104         }
105         else if(x == Jsensor.getDimX()){
106             if(y == Jsensor.getDimY()){
107                 direction = 4 + n.getRandom().nextInt(3);
108             }
109             else{
110                 if(direction >= 1 && direction <= 3){
111                     direction = n.getRandom().nextInt(5);
112                     if(direction != 0)
113                         direction += 3;
114                 }
115             }
116         }
117
118         if(y == 0){
119             if(x == Jsensor.getDimX()){
120                 direction = n.getRandom().nextInt(3);
121                 if(direction != 0)
122                     direction += 5;
123             }
124             else{
125                 if(direction >= 3 && direction <= 5){
126                     direction = n.getRandom().nextInt(5);
127                     if(direction == 3)
128                         direction += 3;
129                     else if(direction == 4)
130                         direction += 3;
131                 }
132             }
133         }
134         else if(y == Jsensor.getDimY()){
135             if(direction ==0 || direction == 1 || direction == 7){
136                 direction = 2 + n.getRandom().nextInt(5);
137             }
138         }
139         return direction;
140     }
```

O método **needChange** realizada a verificação em duas etapas. A primeira delas verificará inicialmente o valor da coordenada x da posição atual do nó (linha 92). Já a segunda, verificará o valor da coordenada y da posição atual do nó (linha 118). Na primeira etapa, o método determinará quais serão as direções poderão ser assumidas pelo nó. Existem três casos aqui: (0, 0), (0, limite da área simulada), (0, qualquer y diferente dos outros dois casos anteriores). No primeiro caso (linha 93), o nó

poderá apenas movimentar-se para direções que façam com que os valores de x e y não decresçam (linha 94). No segundo (linha 96), o nó poderá movimentar-se apenas para direções que façam com o valor da coordenada x não decresça e o valor da coordenada y decresça (linha 97). No terceiro caso (linha 99), o valor da coordenada y da posição do nó poderá crescer ou diminuir, já que este não encontra-se no limite da área de simulação. Caso o valor da coordenada x seja igual ao valor que determina o vértice inferior direito da área simulada, o mesmo tipo de validação também deverá ser realizada. Um raciocínio análogo ao apresentado acima é utilizada na segunda etapa do método **needChange()**, que é quando o valor da coordenada y é analisado inicialmente.

Com a nova direção determinada, o processamento retorna ao método **nextMoviment()**, que vinculará esse novo valor de direção ao nó em questão (linha 67) e retornará uma referência a um novo objeto do tipo **Position**, construído de acordo com o valor determinada pela variável **direção** (linha 69).

9 CellPhoneAntennaConectivity

Classe que deve estender ConectivityModel do JSensor e sobrescrever seu método **isConnected()**, responsável por verificar a conexão entre os nós. O modelo Mobile Phone conta com dois tipos de nó: **CellPhone**, que representam os aparelhos celulares e **Antenna**, que, por sua vez, representam as antenas da rede de telefonia. Mobile Phone define que nós do tipo celular conectam-se apenas à antenas, enquanto elas podem conectar-se a outras antenas ou à celulares.

```
12  public class CellPhoneAntennaConectivity extends ConnectivityModel
13  {
14      @Override
15      public boolean isConnected(Node from, Node to)
16      {
17          if((from instanceof Antenna) && (to instanceof CellPhone))
18                  return true;
19          else if ((from instanceof CellPhone) && (to instanceof Antenna))
20                  return true;
21          else if((from instanceof Antenna) && (to instanceof Antenna))
22                  return true;
23          else
24                  return false;
25      }
26  }
```

10 ClimateAbstractFact

Classe que estende EventModel e permite que o programador crie eventos de aplicação, isto é, uma abstração de fenômeno natural ou qualquer outro tipo de evento que seja capaz de interagir com o ambiente.

Alterações no ambiente de simulação devem ser realizadas em suas células, já que o JSensor divide a área simulada em células. Através do método **setValue()**, que deve ser implementado pela classe, o programador poderá aplicar as modificações desejadas no ambiente de simulação. Este método possui

um único argumento, que representa, justamente, uma célula do JSensor.

```
30      @Override
31      public CellModel setValue(CellModel cell) {
32          Float evaporationUnit;
33          Float rainUnit;
34          double x;
35          Climates climate = Climates.sun;
36
37          if(previousClimate == Climates.sun && previousTimeClimate < 60){
38              previousTimeClimate ++;
39              //0(inclusive) e 10 (exclusive)
40              x = getRandom(cell.getChunk()).nextInt(10);
41
42              if(x < 5){
43                  climate = Climates.sun;
44              }else if(x >= 5 && x < 7){
45                  climate = Climates.cloud;
46                  previousTimeClimate = 60;
47              }else if(x >= 7 && x < 9){
48                  climate = Climates.rain;
49                  previousTimeClimate = 60;
50              }else{
51                  climate = Climates.rainWithThunders;
52                  previousTimeClimate = 60;
53              }
54          }else if(previousClimate == Climates.cloud && previousTimeClimate < 90){
55              previousTimeClimate ++;
56              x = getRandom(cell.getChunk()).nextInt(10);
57
58              if(x < 5){
59                  climate = Climates.cloud;
60              }else if(x >= 5 && x < 7){
61                  climate = Climates.rain;
62                  previousTimeClimate = 90;
63              }else if(x >= 7 && x < 9){
64                  climate = Climates.sun;
65                  previousTimeClimate = 0;
66              }else{
67                  climate = Climates.rainWithThunders;
68                  previousTimeClimate = 90;
69              }
70          }else if(previousClimate == Climates.rain && previousTimeClimate < 95){
71              previousTimeClimate ++;
72              x = getRandom(cell.getChunk()).nextInt(10);
73
74              if(x < 5){
75                  climate = Climates.rain;
76              }else if(x >= 5 && x < 7){
77                  climate = Climates.cloud;
78                  previousTimeClimate = 60;
79              }else if(x >= 7 && x < 9){
80                  climate = Climates.rainWithThunders;
81              }else{
82                  climate = Climates.sun;
83                  previousTimeClimate = 0;
84              }
85          }else {
86              previousTimeClimate = 0;
87              previousClimate = Climates.sun;
88              x = getRandom(cell.getChunk()).nextInt(10);
89
90              if(x < 5)
91                  climate = Climates.sun;
92              else if(x >= 5 && x < 7)
93                  climate = Climates.cloud;
94              else if(x >= 7 && x < 9)
```

```
95                      climate = Climates.rain;
96                  else climate = Climates.rainWithThunders;
97              }
98
99              switch (climate){
100               case sun:
101                   //calcula taxa de evaporação
102                   evaporationUnit = new Float(getRandom(cell.getChunk()).nextInt(20));
103                   //altera o nível de água na célula, baseado na taxa de evaporação
104                   if(((MyCell)cell).getWater() - evaporationUnit > 0)
105                       ((MyCell)cell).setWater(((MyCell)cell).getWater() - evaporationUnit);
106                   else ((MyCell)cell).setWater(new Float(0));
107                   break;
108               case cloud:
109                   //calcula taxa de evaporação
110                   evaporationUnit = new Float(getRandom(cell.getChunk()).nextInt(10));
111                   //altera o nível de água na célula, baseado na taxa de evaporação
112                   if(((MyCell)cell).getWater() - evaporationUnit > 0)
113                       ((MyCell)cell).setWater(((MyCell)cell).getWater() - evaporationUnit);
114                   else ((MyCell)cell).setWater(new Float(0));
115                   break;
116               case rain:
117                   //calcula taxa de chuva
118                   rainUnit = new Float(getRandom(cell.getChunk()).nextInt(20));
119                   //altera o nível de água na célula, baseado na taxa de chuva
120                   ((MyCell)cell).setWater(((MyCell)cell).getWater() + rainUnit);
121                   break;
122               case rainWithThunders:
123                   //calcula taxa de chuva
124                   rainUnit = new Float(getRandom(cell.getChunk()).nextInt(25));
125                   //altera o nível de água na célula, baseado na taxa de chuva
126                   ((MyCell)cell).setWater(((MyCell)cell).getWater() + rainUnit);
127                   //caso a taxa de chuva seja alta, indica que célula foi atingida por um raio
128                   if(rainUnit > 23)
129                       ((MyCell)cell).setThunder(true);
130                   break;
131               default:
132                   System.out.println("Invalid climate!");
133           }
134
135           return cell;
136       }
137   }
```

Com base no clima anterior, determinamos a probabilidade de mudança do clima da célula em questão (linha 37 à 97). Nesse exemplo, há 4 tipos de clima: *ensolarado*, *nublado*, *chuvoso* e *chuvoso com incidência de raios*. A estratégia usada aqui faz com que a probabilidade de mudança de clima de uma célula seja igual a 50%, valor este que é distribuído entre os climas possíveis. Uma vez que o novo clima da célula tenha sido definido, variáveis ajudam a simular as eventuais taxa de evaporação e taxa de chuva na célula corrente. Os valores dessas variáveis, que, nesse exemplo, são definidos de maneira aleatória, atualizam os valores dos campos **thunder** e **water** da célula em questão.

Ao final, o método retorna essa mesma célula, que substitui o respectivo espaço do ambiente de simulação. Isso é necessário para que o simulador mantenha sua consistência em cenários cujo processamento é paralelo.

11 Centralized

Para indicar a localização da ocorrência de um fato, utilizamos a classe **Centralized**. Essa classe deve estender a classe **DistributionModelEvent** do JSensor e implementar o método **getPosition()**.

```
 8  public class Centralized extends DistributionModelEvent{
 9
10          @Override
11          public Position getPosition(Event e) {
12                  return new Position(Configuration.dimX/2, Configuration.dimY/2);
13          }
14
15  }
```

Neste caso, o fatos sempre ocorrerão num ponto que representa o centro da área coberta pela simulação.

12 CustomGlobal

Classe que encapsula várias funcionalidades que permitirão ao programador do modelo ter mais controle sobre sua execução. **CustomGlobal** deve estender a classe **AbsCustomGlobal** do JSensor e sobrescrever os métodos mostrados abaixo.

```
 9  public class CustomGlobal extends AbsCustomGlobal{
10
11      @Override
12      public boolean hasTerminated() {
13          return false;
14      }
15
16      @Override
17      public void preRun() {
18      }
19
20      @Override
21      public void preRound() {
22      }
23
24      @Override
25      public void postRound() {
26      }
27
28          @Override
29          public void postRun() {
30          }
31
32  }
```

O método **hasTerminated()** pode ser usado para interromper uma simulação em andamento, caso seu retorno seja **true**. Os métodos **preRun()** e **postRun()** oferecem ao programador a possibilidade de realizar algum tipo de processamento antes e após a execução da simulação, respectivamente. Os outros dois métodos, **preRound()** e **postRound()**, funcionam de maneira semelhante. A diferença aqui é que eles são acionados antes e após cada round da simulação.

13 Roteamento

O JSensor também oferece ao usuário a possibilidade de utilizar os protocolos de roteamento *DSR* e *AODV* nos modelos criados. Nas próximas seções, explicaremos as classes que implementam esses protocolos e como o usuário poderá fazer como que seu modelo faça uso deles.

Para mais informações sobre os protocolos DSR e AODV, https://en.wikipedia.org/wiki/Dynamic_Source_Routing e https://en.wikipedia.org/wiki/Ad_hoc_On-Demand_Distance_Vector_Routing

Obs.: Caso o usuário queria fazer uso do Mobile Phone com DSR, algumas classes e métodos deverão ser implementados de forma diferente. Tais diferenças serão detalhadas nas seções a seguir.

13.1 Dynamic Source Routing (DSR)

13.1.1 DSRRoute

Classe que representa uma rota do protocolo DSR e possui dois campos: **route** e **ttl**. O primeiro deles, route, é usado para representar as rotas propriamente ditas. No JSensor, tais rotas são listas que armazenam os números identificadores (ID) de cada um dos nós que formam a rota. Já ttl, indica o tempo de validade da rota em questão.

```
7   public class DSRRoute {
8       private LinkedList<Integer> route;
9       private long ttl;
```

Além disso, **DSRRoute** conta com três construtores (o construtor *default*, o construtor de cópia e um construtor que recebe dois parâmetros, indicando a rota usada na construção do objeto e o tempo de validade dessa rota), os métodos de acesso aos campos mencionados acima e com sua própria versão do método **toString()**.

```
11      public DSRRoute() {
12          this.route = new LinkedList<Integer>();
13      }
14
15      public DSRRoute(List<Integer> route, long ttl) {
16          this.route = new LinkedList<Integer>(route);
17          this.ttl = ttl;
18      }
19
20      public DSRRoute(DSRRoute route) {
21          this.route = new LinkedList<Integer>(route.getRoute());
22          this.ttl = route.getTtl();
23      }
24
25      public DSRRoute(ConcurrentLinkedDeque<Integer> route) {
26          this.route = new LinkedList<Integer>(route);
27      }
28
29      public LinkedList<Integer> getRoute() {
30          return route;
31      }
32
33      public void setRoute(DSRRoute route) {
34          this.route = new LinkedList<Integer>(route.getRoute());
35      }
36
37      public long getTtl() {
38          return ttl;
39      }
40
41      public void setTtl(long ttl) {
42          this.ttl = ttl;
43      }
```

13.1.2 DSRRouteCache

Em uma rede que utiliza o protocolo de roteamento DSR, os nós devem ser capazes de armazenar as rotas com as quais já tiveram contato. A esse cache de rotas, damos o nome de tabela de roteamento

do nó, que é representada pela classe **DSSRouteCache** e seu único atributo, **routeCache**.

```
5   public class DSRRouteCache {
6       private ConcurrentHashMap<Integer, DSRRoute> routeCache;
```

Trata-se de um mapeamento, relacionando o identificador de um nó a uma rota. Devido a característica assíncrona da simulação, o cache de rotas de um nó pode ser acessado, simultaneamente, por vários nós. Por este motivo, o atributo **routeCache** é do tipo **ConcurrentHashMap**.

Além dos construtores padrão e de cópia e os métodos de acesso ao atributo descrito acima, **DSRRouteCache** possui métodos para inserir, remover e verificar a existência de uma rota para o determinado destino, que é recebido como parâmetro.

```
8       public DSRRouteCache() {
9           this.routeCache = new ConcurrentHashMap<Integer, DSRRoute>();
10      }
11
12      public DSRRouteCache(DSRRouteCache routeCache) {
13          this.routeCache = new ConcurrentHashMap<Integer, DSRRoute>(routeCache.getRouteCache());
14      }
15
16      public ConcurrentHashMap<Integer, DSRRoute> getRouteCache() {
17          return routeCache;
18      }
19
20      public DSRRoute getRoute(int destination) {
21          return routeCache.get(destination);
22      }
23
24      public void storeRoute(int destination, DSRRoute route) {
25          DSRRoute r = routeCache.get(destination);
26
27          if (r == null || r.getRoute().size() > route.getRoute().size())
28              routeCache.put(destination, new DSRRoute(route));
29      }
30
31      public void removeRoute(int destination) {
32          if (destination >= 0)
33              routeCache.remove(destination);
34      }
35
36      public boolean containsRoute(int destination) {
37          return routeCache.contains(destination);
38      }
```

13.1.3 DSRNode

Classe que representa um nó participante do protocolo de roteamento DSR. Observe que como essa classe estende a classe **Node** do JSensor, caso seja da vontade do usuário fazer uso do modelo Mobile Phone com roteamento via DSR, os nós desse modelo deverão estender **DSRNode** ao invés de **Node**.

```
7   public abstract class DSRNode extends Node {
8       protected DSRAlgorithm dsr;
9       private ThreadLocal<LinkedList<Integer>> forwardRoute;
10      private DSRRouteCache routeCache;
11      protected boolean enableForwarding;
```

O campo **dsr** provê acesso aos algoritmos utilizados pela implementação do protocolo de roteamento DSR no JSensor. Tais algoritmos serão detalhados mais adiante.

O campo **routeCache** tem como objetivo armazenar as rotas com as quais o nó entrou em contato, servindo, portanto, como a tabela de roteamento do nó.

Já **forwardRoute**, é o campo que representa os identificadores dos nós pelos quais o processo de descoberta de rota seguiu até chegar ao nó atual. Note que esse atributo é do tipo **ThreadLocal**. Isso é necessário para que a integridade da rota que esteja em processo de descoberta não seja afetada por outra descoberta, ocorrendo, simultaneamente, em outra thread.

O campo **enableForwarding**, por sua vez, pode ser utilizado para indicar se o nó vai ou não participar do processo de descoberta de rota. Esse controle é útil, por exemplo, no modelo Mobile Phone, já que este conta com dois tipos de nó: aqueles que representam uma antena da rede de telefonia e aqueles que representam os aparelhos celulares. Neste caso, para evitar que as rotas descobertas possuam números identificadores intermediários pertencentes a celulares, o valor desse campo deve ser definido como **false** para todos os nós desse tipo.

Além dos métodos de acesso aos campos descritos acima, **DSRNode** possui também um método que indica se ele é ou não vizinho de um outro nó, recebido como parâmetro.

```
25      public DSRAlgorithm getDsr() {
26          return dsr;
27      }
28
29      public DSRRouteCache getRouteCache() {
30          return routeCache;
31      }
32
33      public ThreadLocal<LinkedList<Integer>> getForwardRoute() {
34          return forwardRoute;
35      }
36
37      public void setForwardRoute(DSRRoute forwardRoute) {
38          this.forwardRoute.set(new LinkedList<Integer>(forwardRoute.getRoute()));
39      }
40
41      public boolean isEnableForwarding() {
42          return enableForwarding;
43      }
44
45      public void setEnableForwarding(boolean enableForwarding) {
46          this.enableForwarding = enableForwarding;
47      }
48
49      public boolean isNeighbour(DSRNode node) {
50          return this.getNeighbours().getNodesList().contains(node);
51      }
```

13.1.4 DSRMessage

Essa classe representa as mensagens enviadas por nós participantes do protocolo de roteamento DSR. Como **DSRMessage** estende a classe **Message** do JSensor, caso seja da vontade do usuário fazer uso

do modelo Mobile Phone com roteamento via DSR, as mensagens desse modelo deverão estender **DSRMessage** ao invés de **Message**.

```
5   public class DSRMessage extends Message {
6       private int hops;
7       private DSRNode sender, destination;
8       private DSRRoute route;
```

Os quatro atributos dessa classe representam, respectivamente, o nó remetente da mensagem, o nó destinatário da mensagem, o número de saltos realizados pela mensagem e a rota pela qual a mensagem deverá seguir.

13.1.5 DSRAlgorithm

Classe que oferece métodos que realizam ou ajudam a realizar a tarefa principal do protocolo de roteamento DSR, a descoberta de uma rota para um dado nó.

O método **routeDiscovery()**, como o nome sugere, é responsável por determinar, caso seja possível, uma rota para um dado nó. Esse método método possui dois parâmetros, **sourceNode** e **destNode**, representando, respectivamente, os nós de origem e destino.

```
17  public class DSRAlgorithm {
18      public DSRRoute routeDiscovery(DSRNode sourceNode, int destNodeID) {
```

A partir desses dois dados, o processo de descoberta de rota já pode ser realizado. No JSensor, a descoberta de rotas baseia-se no fato de que um nó tem acesso a sua lista de vizinhos. Desta forma, a ideia aqui é fazer, com o auxílio de uma fila, uma busca pelo nó de destino na lista de vizinhos do nó (busca por largura), começando pelo nó de origem. É importante ressaltar que, no caso do modelo Mobile Phone, a fim de evitar a geração de rotas inconsistentes, isto é, rotas do tipo *celular origem*, *antena 1*, *antena 2*, *celular intermediário*, *antena 3*, *celular destino*, o nó de origem deve ser a antena ao qual o celular está conectado. Para garantir que um mesmo nó não seja visitado mais de uma vez, utilizamos a variável **visitedNodes**.

Inicialmente, a variável **forwardRoute**, que armazena os identificadores dos nós pelos quais o processo de descoberta de rota passou, simulando o envio de um pacote *Route Request* é atualizada com o número que identifica o próprio nó de origem. Após isso, o nó de origem é marcado como já visitado e inserido na fila que controla a ordem na qual os nós serão visitados.

```
26          sourceNode.getForwardRoute().get().add(sourceNode.getID());
27
28          // Indica que nó de origem já foi visitado
29          visitedNodes.add(sourceNode);
30          // Insere nó de origem na fila
31          queue.add(sourceNode);
```

A partir daí, tem início o laço principal do método. O objetivo desse método é simular o envio de pacotes do tipo *Route Request* aos vinzinhos de um nó.

```
33          while (queue.size() > 0) {
34              DSRNode node = queue.poll();
35
36              if (destNodeID == node.getID()) {
37                  // Nó retirado da fila é o nó de destino
38                  route = node.getForwardRoute().get();
39
40                  // Retorna rota encontrada
41                  return new DSRRoute(route, DSRMessageTransmissionModel.timeToReach() *
42                                      DSRConstants.getActiveRouteTime() + Jsensor.currentTime);
43              } else if (node.enableForwarding == true) {
44                  // Procura pela rota no cache do nó removido da fila
45                  DSRRoute cachedRoute = ((DSRNode) node).getRouteCache().getRoute(destNodeID);
46
47                  if (cachedRoute != null) {
48                      // Nó removido da fila possui rota para o nó de destino
49                      route = node.getForwardRoute().get();
50                      // Para evitar duplicações ao combinar as rotas, remove o último ID de route
51                      route.removeLast();
52                      /* Concatena rota que representa caminho até o nó 'node' com
53                      a rota encontrada no cache*/
54                      route.addAll(cachedRoute.getRoute());
55
56                      if (hasDuplicates(new DSRRoute(route, 0L)) == false) {
57                          // Atualiza valor do campo que indica tempo de vida da rota
58                          cachedRoute.setTtl(cachedRoute.getTtl() +
59                                             DSRMessageTransmissionModel.timeToReach() *
60                                             DSRConstants.getActiveRouteTime());
61
62                          // Retorna rota encontrada
63                          return new DSRRoute(route, ttl);
64                      } else {
65                          route = null;
66                      }
67                  } else {
68                      neighbourList = node.getNeighbours().getNodesList();
69
70                      for (Node neighbour : neighbourList) {
71                          if (visitedNodes.contains(neighbour) == false) {
72                              visitedNodes.add((DSRNode) neighbour);
73                              queue.add((DSRNode) neighbour);
74
75                              ((DSRNode) neighbour).getForwardRoute().get().clear();
76
77                              // Atualiza variável que representa o caminho da descoberta
78                              ttl += Jsensor.currentTime;
79                              DSRRoute fwRoute = new DSRRoute(node.getForwardRoute().get(), ttl);
80                              fwRoute.getRoute().add(neighbour.getID());
81                              ((DSRNode) neighbour).setForwardRoute(fwRoute);
82
83                              // Atualiza cache de rotas do vizinho com rota para o nó de origem
84                              DSRRoute r = new DSRRoute(revertRoute(fwRoute));
85                              ((DSRNode) neighbour).getRouteCache().storeRoute(sourceNode.getID(), r);
86                              // Define evento de timer para remoção da rota
87                              setRouteTtlTimer((DSRNode) neighbour, sourceNode.getID());
88                          }
89                      }
90                  }
91              }
92          }
```

A cada iteração, é verificado se o nó removido da fila é o nó procurado (linha 36). Em caso positivo, a rota armazenada no atributo *forwardRoute* representará a rota completa até o destino (no caso do Mobile Phone, o primeiro nó da rota será a antena à qual o nó de origem está conectado). Neste caso, a rota é retornada (linhas 43 e 44). Caso contrário, será verificado se nó removido da fila possui, em seu

cache de rotas, uma rota para o nó de destino (linha 45). Porém, essa verificação só será realizada caso o atributo **enableForwarding** no nó em questão seja igual a **true**.

Na linha 47, verificamos se a variável **cachedRoute** é diferente de *null*, o que indica que o nó removido da fila possui uma rota para o nó de destino. Em caso positivo, será preciso concatenar a rota contida no campo **forwardRoute** com a rota encontrada no cache de rotas do nó. A fim de evitar que o ID do nó que possui a rota fique duplicado após a concatenação das rotas (linha 54), será preciso remover o último ID contido em **forwardRoute** (linhas 51). Porém, mesmo após esse ajuste, é possível que a nova rota gerada tenha algum ID duplicado. Nesses casos, o protocolo de roteamento DSR diz que essa rota não poderá ser considerada como válida, fazendo com que o processo de descoberta de rotas continue.

```
44                  // Procura pela rota no cache do nó removido da fila
45                  DSRRoute cachedRoute = ((DSRNode) node).getRouteCache().getRoute(destNodeID);
46
47                  if (cachedRoute != null) {
48                      // Nó removido da fila possui rota para o nó de destino
49                      route = node.getForwardRoute().get();
50                      // Para evitar duplicações ao combinar as rotas, remove o último ID de route
51                      route.removeLast();
52                      /* Concatena rota que representa caminho até o nó 'node' com
53                      a rota encontrada no cache*/
54                      route.addAll(cachedRoute.getRoute());
55
56                      if (hasDuplicates(new DSRRoute(route, 0L)) == false) {
57                          // Atualiza valor do campo que indica tempo de vida da rota
58                          cachedRoute.setTtl(cachedRoute.getTtl() +
59                                             DSRMessageTransmissionModel.timeToReach() *
60                                             DSRConstants.getActiveRouteTime());
61
62                          // Retorna rota encontrada
63                          return new DSRRoute(route, ttl);
64                      } else {
65                          route = null;
66                      }
```

Então, caso o nó removido da fila não seja o nó de destino ou não possua uma rota para ele, o processo de descoberta continuará, verificando, um a um, se algum dos vizinhos desse nó é ou possui uma rota para o nó de destino.

```
68                  neighbourList = node.getNeighbours().getNodesList();
69
70                  for (Node neighbour : neighbourList) {
71                      if (visitedNodes.contains(neighbour) == false) {
72                          visitedNodes.add((DSRNode) neighbour);
73                          queue.add((DSRNode) neighbour);
74
75                          ((DSRNode) neighbour).getForwardRoute().get().clear();
76
77                          // Atualiza variável que representa o caminho da descoberta
78                          ttl += Jsensor.currentTime;
79                          DSRRoute fwRoute = new DSRRoute(node.getForwardRoute().get(), ttl);
80                          fwRoute.getRoute().add(neighbour.getID());
81                          ((DSRNode) neighbour).setForwardRoute(fwRoute);
82
83                          // Atualiza cache de rotas do vizinho com rota para o nó de origem
84                          DSRRoute r = new DSRRoute(revertRoute(fwRoute));
85                          ((DSRNode) neighbour).getRouteCache().storeRoute(sourceNode.getID(), r);
86                          // Define evento de timer para remoção da rota
```

```
87                                  setRouteTtlTimer((DSRNode) neighbour, sourceNode.getID());
88                              }
89                          }
```

Na linha 68, a lista de vizinhos do nó é acessada. Cada vizinho, ainda não visitado, do nó removido da fila é o adicionado à fila (linha 73) e é então marcado como já visitado (linha 72). Após isso, atualizamos a variável que armazena o caminho percorrido pelo processo de descoberta da rota com o número identificador do vizinho corrente, indicando que a mensagem *Route Request* foi recebida por este vizinho (linhas 79 à 81). Dessa forma, simulamos o envio de mensagens de um nó a seus vizinhos, pois cada vizinho possuirá uma variável que representa o caminho pelo qual o processo de descoberta da rota seguiu até chegar a ele. No momento da criação da variável que representa o caminho percorrido, definimos o valor da variável **ttl** (linha 78), com base no tempo atual da simulação, no tempo gasto para que uma mensagem vá de um ao outro e no valor definido para **activeRouteTime**, detalhado mais adiante. A rota utilizada para atualizar o campo *forwardRoute* do nó vizinho, quando invertida, gera uma rota que leva ao nó de origem, que também é incluída no cache de rotas do nó vizinho (linhas 84 e 85). Por fim, criamos um evento de timer para verificar a necessidade de remove a rota, no devido tempo (linha 87).

Esse laço repete-se até que uma rota seja encontrada ou que a fila se esvazie e uma rota não seja encontrada.

Com a rota definida, a mensagem partirá do nó de origem, passará por cada um dos nós que formam a rota e será entregue ao nó de destino. Durante esse tráfego, é possível que um nó x não consiga realizar o envio da mensagem ao próximo no da lista, y. Nesses casos, o nó que percebeu o problema na rota acionará o método **routeError()**, que será responsável por remover tal rota, agora inválida, dos caches de rotas (tabela de roteamento) dos nós que participaram do envio dessa mensagem por essa rota.

```
165     public void routeError(int lastValidLink, DSRRoute route) {
166         long ttl;
167         int index = route.getRoute().indexOf(lastValidLink);
168 
169         DSRNode node = null;
170 
171         for (int i = index - 1; i >= 0; i--) {
172             node = ((DSRNode) Jsensor.getNodeByID(route.getRoute().get(i)));
173 
174             if (route.getRoute().getLast() != null)
175                 node.getRouteCache().removeRoute(route.getRoute().getLast());
176 
177             ttl = DSRMessageTransmissionModel.timeToReach() * DSRConstants.getActiveRouteTime() +
178                     Jsensor.currentTime;
179             DSRRoute newRoute = new DSRRoute(route.getRoute().subList(i, index + 1), ttl);
180             node.getRouteCache().storeRoute(lastValidLink, newRoute);
181             // Define evento de timer para remoção da rota
182             setRouteTtlTimer(node, lastValidLink);
183         }
184     }
```

Além dos métodos descritos acima, a classe **DSRAlgorithm** conta com alguns métodos auxiliares tais como **revertRoute()**, **hasDuplicates()** e **sortRoute()**, que são inclusive utilizados durante o processo de descoberta da rota e **nextNode()**, **previousNode()**, **sendMessage()**, **newRouteDiscovery()** e **setRouteTtlTimer()**. Os métodos **nextNode()** e **previousNode()**, dado uma rota e o

número identificador do nó (ID), retornam, respectivamente, o próximo e o nó anterior. Já **sendMessage()**, realiza o envio de uma mensagem, via *unicast*, de um nó para outro, caso os nós envolvidos no envio sejam vizinhos. Por sua vez, **newRouteDiscovery()** é usado quando há necessidade de realizar uma nova descoberta de rota para um destino, fazendo isso através da criação de um evento de timer.

O cache de rotas dos nós são úteis, pois podem evitar que novas descobertas de rota sejam feitas sem necessidade. Porém, com o intuito de evitar que essas estruturas de dados contenham rotas que já não são utilizadas há algum tempo ou que elas ocupem muito espaço em memória, é necessário utilizar algum mecanismo que controle a quantidade de rotas nos caches. O método **setRouteTtlTimer()** é o método responsável por realizar tal controle.

Rotas possuem um campo (ttl), que indica seu tempo de vida. Quando uma nova rota é criada, seu tempo de vida é definido como o tempo atual da simulação + o valor de **activeRouteTime** (seção anterior). Sempre que uma nova rota é incluída (em um cache de rotas) ou utilizada, um evento de timer é criado. Esse evento será responsável por verificar se determinada rota pode ser considerada inativa (não foi mais acessada após sua inclusão no cache) e, em caso positivo, por removê-la do cache (linha 198). No momento da criação desse tipo de evento, é utilizado o método startRelative(), da classe **TimerEvent**, significando que o tempo em que ele será disparado é baseado no tempo atual da simulação + activeRouteTime unidades de tempo (linha 199).

```
97      public DSRRoute revertRoute(DSRRoute route) {
98          Iterator<Integer> ite = route.getRoute().descendingIterator();
99
100         DSRRoute revertedRoute = new DSRRoute();
101
102         while (ite.hasNext() == true)
103             revertedRoute.getRoute().add(ite.next());
104
105         return revertedRoute;
106     }
107
108     private boolean hasDuplicates(DSRRoute route) {
109         DSRRoute sortedRoute = new DSRRoute(sortRoute(route));
110
111         for (int i = 0; i < sortedRoute.getRoute().size() - 1; i++) {
112             if (sortedRoute.getRoute().get(i + 1).intValue() == sortedRoute.getRoute().get(i).
113                 intValue())
114                 return true;
115         }
116
117         return false;
118     }
119
120     private DSRRoute sortRoute(DSRRoute route) {
121         DSRRoute sortedRoute = new DSRRoute(route);
122         sortedRoute.getRoute().sort(new Comparator<Integer>() {
123             @Override
124             public int compare(Integer var, Integer var2) {
125                 if (var.intValue() < var2.intValue())
126                     return -1;
127                 else if (var.intValue() > var2.intValue())
128                     return 1;
129                 else
130                     return 0;
131             }
132         });
133
```

```
134             return sortedRoute;
135         }
136 
137         public DSRNode nextNode(int nodeID, DSRRoute route) {
138             int index;
139 
140             index = route.getRoute().indexOf(nodeID) + 1;
141 
142             return (DSRNode) Jsensor.getNodeByID(route.getRoute().get(index));
143         }
144 
145         public DSRNode previousNode(int nodeID, DSRRoute route) {
146             int index;
147 
148             index = route.getRoute().indexOf(nodeID) - 1;
149 
150             return (DSRNode) Jsensor.getNodeByID(route.getRoute().get(index));
151         }
152 
153         public boolean sendMessage(DSRNode from, DSRNode to, DSRMessage msg) {
154             if (from.isNeighbour(to)) {
155                 from.unicast(msg, to);
156 
157                 // Mensagem foi enviada
158                 return true;
159             } else {
160                 // Mensagem não foi enviada
161                 return false;
162             }
163         }
```

```
190         public void newRouteDiscovery(DSRNode node, DSRMessage msg) {
191             DSRResendTimer rt = new DSRResendTimer();
192             rt.setMessage(msg);
193             rt.startRelative(1, node);
194         }
195 
196         public void setRouteTtlTimer(DSRNode node, int destination) {
197             DSRRemoveRouteEntryTimer timer = new DSRRemoveRouteEntryTimer();
198             timer.setEntryToRemove(destination);
199             timer.startRelative(DSRConstants.getActiveRouteTime(), node);
200         }
```

13.1.6 DSRResendTimer

A classe **DSRResendTimer** representa um evento de timer que deve ser usado quando a tentativa de envio de uma mensagem resultar em falha. Isso pode acontecer, por exemplo, quando o nó que seria o próximo a receber a mensagem encontra-se fora do alcance do nó remetente.

Essa classe possui apenas um atributo, **message**, que representa a mensagem que deve ser reenviada

```
8   public class DSRResendTimer extends TimerEvent {
9       private DSRMessage message;
```

Como **DSRResendTimer** estende a classe **TimerEvent** do JSensor, deve implementar seu método **fire()**, pois ele será responsável por tentar encontrar uma nova rota para o nó de destino da mensagem.

```
15      @Override
16      public void fire() {
17          Jsensor.log("[Resend message] time: " + Jsensor.currentTime + " node " +
18                  this.node.getID() + " will resend the message to node " +
19                  message.getSender().getID());
20
21          DSRRoute sourceRoute = ((DSRNode) this.node).getDsr().routeDiscovery((DSRNode) this.node,
22                                  message.getDestination().getID());
23
24          if (sourceRoute != null) {
25              // Adiciona rota descoberta ao cache do nó corrente
26              ((DSRNode) this.node).getRouteCache().storeRoute(message.getDestination().getID(),
27                                                               sourceRoute);
28              // Cria evento de timer para remoção da rota inserida no cache do nó
29              ((DSRNode) this.node).getDsr().setRouteTtlTimer(((DSRNode) this.node),
30                                                              message.getDestination().getID());
31              message.setRoute(sourceRoute);
32              ((DSRNode) this.node).getDsr().sendMessage((DSRNode) this.node,
33                      (DSRNode) Jsensor.getNodeByID(sourceRoute.getRoute().getFirst()), message);
34          } else {
35              Jsensor.log("[No route] time: " + Jsensor.currentTime + " nodeID: " +
36                      message.getDestination().getID());
37              Jsensor.log("[No route] position: " + message.getDestination().getPosition().getPosX()+
38                      ", " + message.getDestination().getPosition().getPosY());
39          }
40      }
```

Inicialmente (linhas 17 à 19), o reenvio da mensagem é registrado no arquivo de log do JSensor. Na linha 21, o método que realiza a busca por uma rota ao nó de destino é acionado. Caso uma rota seja encontrada, ela é adicionada ao cache do nó que usado como nó de origem da descoberta da rota (linha 26), é criado um evento de timer para remoção da rota (linha 29), é adicionada à mensagem (linha 31) e enviada ao próximo nó (linhas 32 e 33). Caso contrário, esse fato é registrado no arquivo de log do JSensor.

13.1.7 DSRMessageTransmissionModel

Classe que possui apenas um método estático e serve para informar quantas unidades de tempo do JSensor uma mensagem gasta para sair de um nó remetente a e chegar a um nó destinatário b.

```
3   public class DSRMessageTransmissionModel {
4       private static final int time = 1;
5
6       public static int timeToReach() {
7           return time;
8       }
9   }
```

13.1.8 DSRUtils

DSRUtils é uma simples classe, que tem como objetivo encapsular classes utilitárias para o AODV. No momento, há apenas uma classe interna, **DSRConstants**. Seu único atributo, **activeRouteTime** indica o tempo de vida de cada rota, em unidades de tempo do JSensor.

```
3   public class DSRUtils {
4       public static class DSRConstants {
5           private static final int activeRouteTime = 3000;
6
7           public static int getActiveRouteTime() {
8               return activeRouteTime;
9           }
10      }
11  }
```

13.1.9 DSRRemoveRouteEntryTimer

A classe **DSRRemoveRouteEntryTimer** representa um evento de timer e deve estender a classe **TimerEvent** do JSensor. **DSRRemoveRouteEntryTimer** deve ser usada para verificar a necessidade de remover uma determinada rota do cache de rotas de um dado nó.

```
6   public class DSRRemoveRouteEntryTimer extends TimerEvent {
7       private int entryToRemove;
8
9       @Override
10      public void fire() {
11          DSRRoute re = ((DSRNode) this.node).getRouteCache().getRoute(entryToRemove);
12
13          if (re != null && re.getTtl() == Jsensor.currentTime )
14              ((DSRNode) this.node).getRouteCache().removeRoute(entryToRemove);
15      }
16
17      public void setEntryToRemove(int entryToRemove) {
18          this.entryToRemove = entryToRemove;
19      }
20  }
```

Essa classe possui apenas um atributo, **entryToRemove**, que indicará a rota a ser removida do cache do nó questão. Como **TimerEvent** é sua super classe, **DSRRemoveRouteEntryTimer** deverá implementar o método **fire()**, já que esse é o método que será chamado quando o evento for acionado. Na linha 11, procuramos pela rota que leva até o destino indicado pela variável entryToRemove. Caso a rota exista e o valor de seu campo **ttl** seja igual ao tempo atual da simulação (linha 13), ela será removida do cache (linha 14).

Obs.: É importante ressaltar que, para que o protocolo de roteamento DSR funcione no modelo Mobile Phone, os exemplos de implementação demonstrados anteriormente deverão ser ajustados. Nas próximas seções, mostraremos como esse ajuste deverá ser feito.

13.1.10 SMSMessage

Como citado na seção relativa à classe **DSRMessage** (13.1.4), classes que representam as mensagens enviadas no protocolo de roteamento DSR deverão estender **DSRMessage**.

```
16      public SMSMessage(DSRNode sender, DSRNode destination, int hops, short chunck) {
17          super(sender, destination, hops, chunck);
18
19          this.message = "";
20      }
21
```

```
22      public SMSMessage(DSRNode sender, DSRNode destination, int hops, long ID,
23                        String message, DSRRoute route) {
24          super(ID, sender, destination, hops, route);
25
26          this.message = message;
27      }
```

Nesse exemplo, **SMSMessage** possui apenas um atributo, **message**, que contém a mensagem propriamente dita e dois construtores. Note que o tipo dos nós origem e destino usado nos construtores é **DSRNode** e não mais **Node**.

13.1.11 SMSTimer

A primeira modificação a ser feita nessa classe está relacionada ao tipo dos nós utilizados pelo protocolo de roteamento. Quando a mensagem for criada, no método **fire()**, será necessário fazer uma conversão explícita de tipos, já que o tipo esperado pelo construtor da classe **SMSMessage** agora é **DSRNode**.

```
29              SMSMessage msg = new SMSMessage((DSRNode) this.node, destNode, 0,
30                                              this.node.getChunk());
```

Além disso, chamada ao método que realiza a descoberta da rota é feita pelo método **fire()**. Como o método **routeDiscovery()** retorna a rota, uma variável do tipo **DSRRoute** deverá ser criada para armazenar esse retorno.

```
41              DSRRoute sourceRoute = ((DSRNode) this.node).getDsr().routeDiscovery(sourceNodesAntenna,
42                                                                                   destNode.getID());
```

Importante.: No Mobile Phone, o argumento que indica o nó de origem para o método de descoberta da rota deverá ser o nó que representa a antena do nó que deseja enviar a mensagem. Tal restrição é necessária para que rotas inconsistentes não sejam geradas, isto é, rotas do tipo *celular origem*, *antena 1*, *antena 2*, *celular intermediário*, *antena 3*, *celular destino*. A linha 38 mostra como é possível obter o nó que representa a antena de um outro nó.

```
38              DSRNode sourceNodesAntenna = (DSRNode) Jsensor.getNodeByID(((CellPhone) this.node).
39                                                                         getMyAntenna());
```

Uma vez que a rota tenha sido descoberta, será necessário incluí-la no cache de rotas do nó que deseja enviar a mensagem (linha 46), bem como no cache de rotas do nó que representa sua antena (linha 48). Além disso, para cada uma das rotas adicionadas, é criado um evento de timer que será responsável por removê-la do cache, caso esteja inativa (linhas 50 e 52). Por fim, basta incluir a rota no objeto que representa a mensagem a ser enviada ao nó seguinte (linha 54) e fazer o envio da mensagem (linha 56), de acordo o retorno do método **nextNode()**.

```
45                  // Adiciona a rota descoberta ao cache do nó de origem
46                  ((DSRNode) this.node).getRouteCache().storeRoute(destNode.getID(), sourceRoute);
47                  // Adiciona a rota descoberta ao cache da antena conectado ao nó de origem
48                  sourceNodesAntena.getRouteCache().storeRoute(destNode.getID(), sourceRoute);
49                  // Cria evento de timer para remoção da rota inserida no cache do nó de origem
50                  ((DSRNode) this.node).getDsr().setRouteTtlTimer(((DSRNode) this.node),
51                      destNode.getID());
52                  // Cria evento de timer para remoção da rota inserida no cache da antena
53                  sourceNodesAntenna.getDsr().setRouteTtlTimer(sourceNodesAntenna, destNode.getID());
54
55                  msg.setRoute(sourceRoute);
56
57                  ((DSRNode) this.node).getDsr().sendMessage(((DSRNode) this.node), ((DSRNode)
58                      this.node).getDsr().nextNode(this.node.getID(), msg.getRoute()), msg);
```

13.1.12 Antenna

O método **handleMessages()**, que é o método responsável por realizar o processamento das mensagens recebidas pelo nó, deverá ser alterado para que contemple as características do protocolo de roteamento DSR.

```
37                  currentNodeIndex = smsMessage.getRoute().getRoute().indexOf(this.ID);
38                  routeSize = smsMessage.getRoute().getRoute().size();
39                  DSRRoute partialRoute;
40
41                  // Atualiza cache do nó corrente com a rota inversa
42                  ttl += Jsensor.currentTime;
43                  partialRoute = new DSRRoute(smsMessage.getRoute().getRoute().subList(0,
44                      currentNodeIndex + 1), ttl);
45                  this.getRouteCache().storeRoute(smsMessage.getRoute().getRoute().getFirst(),
46                      this.dsr.revertRoute(partialRoute));
47                  // Timer to remove route
48                  this.dsr.setRouteTtlTimer(this, smsMessage.getRoute().getRoute().getFirst());
```

Caso a mensagem em processamento seja nova, o nó corrente deverá encaminhá-la ao próximo nó da rota. Porém, antes de fazer isso, este nó salva, em seu próprio cache de rotas, a rota invertida que leva até o nó remetente da mensagem (linha 45). Para isso, faz uso da variável **currentNodeIndex**, que o auxiliam a delimitar o intervalo útil de rota que será armazenado no cache (linha 43). Na linha 46, é criado o evento de timer responsável pela remoção da rota, caso esteja inativa.

```
50                  int lastAntennaID = smsMessage.getRoute().getRoute().get(routeSize - 2);
51
52                  if (this.dsr.destinationNodeFound(this.getID(), lastAntennaID) == true) {
53                      if (this.getNeighbours().getNodesList().contains(smsMessage.
54                                                                  getDestination())) {
55                          // Antena ainda pode alcançar nó de destino
56                          this.unicast(m, Jsensor.getNodeByID(smsMessage.getDestination().
57                                                                  getID()));
58                      } else {
59                          Jsensor.log("[No signal] time: " + Jsensor.currentTime +
60                                  " nodeID: " + this.getID());
61                          Jsensor.log("[No signal] position: "+ this.getPosition().getPosX() +
62                                  ", " + this.getPosition().getPosY());
63                      }
64                  } else {
65                      this.messagesIDs.add(smsMessage.getID());
66                      ((SMSMessage) m).setMessage(smsMessage.getMessage().concat(" - " +
```

```
67                          Integer.toString(this.ID)));
68
69                      // Envia a mensagem para o próximo nó, via unicast
70                      DSRNode nextNode = this.dsr.nextNode(this.ID, smsMessage.getRoute());
71                      boolean sendResult = this.dsr.sendMessage(this, nextNode, smsMessage);
72
73                      if (sendResult == true) {
74                          // Atualiza cache do nó corrente com parte da rota
75                          ttl += Jsensor.currentTime;
76                          partialRoute = new DSRRoute(smsMessage.getRoute().getRoute().
77                                  subList(currentNodeIndex, routeSize), ttl);
78                          this.getRouteCache().storeRoute(smsMessage.getRoute().getRoute().
79                                  getLast(), partialRoute);
80                          // Timer to remove route
81                          this.dsr.setRouteTtlTimer(this, smsMessage.getRoute().getRoute().
82                                  getLast());
83                      } else {
84                          // Não foi possível enviar a mensagem para o próximo nó da rota
85                          Jsensor.log("[Route error] time: " + Jsensor.currentTime +
86                                  " route to node " + smsMessage.getDestination().getID() +
87                                  " must be rediscovered");
88                          // Rota deve ser removida do cache
89                          this.getRouteCache().removeRoute(nextNode.getID());
90                          this.dsr.routeError(this.getID(), smsMessage.getRoute());
91                          this.dsr.newRouteDiscovery(this, smsMessage);
92                      }
93                  }
```

Após isso, o método verifica se o nó (antena) corrente é a última antena da rota (linha 51), que é a antena responsável por enviar a mensagem para o nó (celular) de destino. A mensagem será enviada ao destinatário caso ele ainda seja vizinho desta antena (linha 57). Caso contrário, entendemos que o celular em questão não possui sinal e a mensagem não pode ser entregue. Decisão esta que é registrada no arquivo de log do JSensor (linhas 61 e 63).

Caso o nó corrente não seja a última antena da rota, acontece a tentativa de envio da mensagem à próxima antena (linhas 71 e 72). O resultado dessa tentativa é armazenada na variável **sendResult**. Caso seu valor seja **true**, o envio foi realizado com sucesso e então esse nó poderá armazenar, em seu cache, uma rota que parte dele e vai nó destinatário (linhas 77 e 78). Caso contrário, a falha no envio da mensagem será registrada no arquivo de log do JSensor (83, 84 e 85) e o nó corrente removerá, de seu cache de rotas, caso exista, a rota para o próximo nó (linha 89). Além disso, acionará o método **routeError()**, que acessará o cache de rotas de cada um dos nós participantes deste envio de mensagem e removerá a rota para o destino dessa mensagem (linha 91). Por fim, o nó corrente a chamada ao método **newRouteDiscovery()**, que agendará, para a próxima unidade de tempo, um evento de timer responsável por tentar encontrar uma nova rota para o destinatário da mensagem (linha 90).

13.1.13 CellPhone

Analogamente ao nó do tipo Antenna, a alteração aqui também deverá ser feita no método **handleMessages()**.

```
32                      // Atualiza cache do nó corrente com a rota inversa
33                      DSRRoute revertedRoute = new DSRRoute(this.dsr.revertRoute(smsMessage.
34                              getRoute()));
35                      revertedRoute.getRoute().removeFirst();
36                      revertedRoute.getRoute().addLast(smsMessage.getSender().getID());
```

```
37                      this.getRouteCache().storeRoute(smsMessage.getSender().getID(), revertedRoute);
```

Quando uma nova mensagem for recebida pelo nó, este deverá armazenar, em seu cache de rotas, a rota invertida para o nó remetente. No caso do modelo Mobile Phone, o primeiro número identificador numa rota refere-se à antena a qual o nó de origem (remetente) está conectado. Então, após inverter a rota (linha 33), também será necessário remover o primeiro número identificador da rota (linha 34), que agora refere-se ao nó de destino e acrescentar, ao final da rota, o número identificador do nó de origem (linha 35). Após isso, a nova rota criada pode ser inserida no cache de rotas do nó corrente, destinatário da mensagem.

```
41                      Jsensor.log("[Received] time: " + Jsensor.currentTime + "\t sensorID: " +
42                              this.ID + "\t receivedFrom: " + smsMessage.getSender().getID() +
43                              "\t hops: " + smsMessage.getHops() + "\t msg: "+
44                              smsMessage.getMessage().concat(" - " + String.valueOf(this.ID)));
45
46                      // Probabilidade de responder mensagem é igual a 0.8
47                      if (this.getRandom().nextDouble() < 0.8) {
48                          this.findAntenna();
49
50                          if (this.getMyAntenna() > -1) {
51                              SMSMessage replyMessage = new SMSMessage(this, smsMessage.getSender(),
52                                                                       0, this.getChunk());
53                              replyMessage.setRoute(revertedRoute);
54                              DSRNode nextNode = this.dsr.nextNode(this.ID, replyMessage.getRoute());
55                              this.dsr.sendMessage(this, nextNode, replyMessage);
56                          } else {
57                              Jsensor.log("[No signal] time: " + Jsensor.currentTime +
58                                      " node " + this.getID() + " has no signal");
59                          }
60                      }
```

Nas linhas 41-44, o recebimento da mensagem é registrado no arquivo de log do JSensor.

Com o intuito de simular uma troca de mensagens mais intensa, o usuário pode decidir por fazer, neste ponto, com que o nó destinatário da mensagem responda ao remetente. Optamos por fazer isso neste exemplo, tornando o envio da resposta condicionado a um simples cálculo de probabilidade (linha 48). Então, caso o nó destinatário esteja conectado a alguma antena, fará, com a ajuda da rota invertida, o envio de uma mensagem de resposta ao nó remetente (linhas 54 a 57).

Outro detalhe importante e que deve ser observado, é o valor do atributo **enableForwarding**, que deve ser definido como false para nós do tipo **CellPhone**. Apenas para relembrar, isso é necessário para que rotas geradas não fiquem inconsistentes, isto é, uma que contém um nó do tipo celular como nó intermediário.

```
97          /*A fim de evitar rotas do tipo:
98          celular (origem), antena, celular, antena, celular (destino) nós do tipo
99          CellPhone não devem encaminhar a rota contida no atributo forwardRoute*/
100         this.enableForwarding = false;
```

13.2 Ad hoc On-Demand Distance Vector (AODV)

13.2.1 AODVRouteRequest

Classe que representa um pacote do tipo *Route Request*. Essa classe possui três atributos, **sourceID**, **destID** e **hopCount**, que representam, respectivamente, o número identificador do nó de origem, o número identificador do nó de destino e o número de saltos realizados por esse pacote. AODVRouteRequest conta também com dois construtores, o padrão e o que inicializa os atributos de acordo com os argumentos recebidos e com os métodos de acesso a seus atributos.

```
3   public class AODVRouteRequest {
4       private int sourceID, destID, hopCount;
5
6       public AODVRouteRequest(int sourceID, int destID, int hopCount) {
7           this.sourceID = sourceID;
8           this.destID = destID;
9           this.hopCount = hopCount;
10      }
11
12      public AODVRouteRequest() {
13
14      }
15
16      public int getSourceID() {
17          return sourceID;
18      }
19
20      public void setSourceID(int sourceID) {
21          this.sourceID = sourceID;
22      }
23
24      public int getDestID() {
25          return destID;
26      }
27
28      public void setDestID(int destID) {
29          this.destID = destID;
30      }
31
32      public int getHopCount() {
33          return hopCount;
34      }
35
36      public void setHopCount(int hopCount) {
37          this.hopCount = hopCount;
38      }
39  }
```

13.2.2 AODVRouteEntry

Classe que representa uma entrada de rota no cache de rotas de um nó do tipo AODV. Os atributos *destination*, *next*, *hops* e *ttl* representam, respectivamente, o número identificador do nó de destino relacionado àquela entrada de rota, o número identificador do próximo nó para o qual o nó corrente enviará mensagens endereçadas ao nó de destino, o número de saltos que deverão ser realizados para que o a mensagem vá do nó corrente ao nó de destino e o tempo de vida (time to live) daquela entrada de rota.

```
3   public class AODVRouteEntry {
```

```
4       private int destination, next, hops;
5       private long ttl;
```

Além disso, a classe **AODVRouteEntry** possui os métodos de acesso aos atributos citados acima, dois construtores, o padrão e aquele que inicializa os atributos da classe com os argumentos recebidos e uma versão personalizada do método **toString()**.

```
7       public AODVRouteEntry(int destination, int next, int hops, long ttl) {
8           this.destination = destination;
9           this.next = next;
10          this.hops = hops;
11          this.ttl = ttl;
12      }
13
14      public AODVRouteEntry() {
15
16      }
17
18      public int getDestination() {
19          return destination;
20      }
21
22      public void setDestination(int destination) {
23          this.destination = destination;
24      }
25
26      public int getNext() {
27          return next;
28      }
29
30      public void setNext(int next) {
31          this.next = next;
32      }
33
34      public int getHops() {
35          return hops;
36      }
37
38      public void setHops(int hops) {
39          this.hops = hops;
40      }
41
42      public long getTtl() {
43          return ttl;
44      }
45
46      public void setTTL(long ttl) {
47          this.ttl = ttl;
48      }
49
50      @Override
51      public String toString() {
52          return "Destination: " + this.destination + ", Next: " + this.next +
53                  ", Hops: " + this.hops + ", TTL: " + this.ttl + "\n";
54      }
```

13.2.3 AODVNode

Classe que representa um nó participante do protocolo de roteamento AODV. Observe que como essa classe estende a classe **Node** do JSensor, caso seja da vontade do usuário fazer uso do modelo

Mobile Phone com roteamento via AODV, os nós desse modelo deverão estender **AODVNode** ao invés de **Node**.

```
 7  public abstract class AODVNode extends Node {
 8      protected AODVAlgorithm aodv;
 9      private ThreadLocal<AODVRouteRequest> rreq;
10      private AODVRouteCache routeCache;
11      private ThreadLocal<AODVNode> previousNode;
12      protected boolean enableForwarding;
```

O atributo **aodv** provê acesso aos algoritmos utilizados pela implementação do protocolo de roteamento AODV no JSensor. Tais algoritmos serão detalhados mais adiante.

O atributo **rreq** representa o pacote do tipo *Route Request* recebido pelo nó. Note que esse atributo é do tipo **ThreadLocal**. Isso é necessário para que a integridade da rota que esteja em processo de descoberta não seja afetada por outra descoberta, ocorrendo, simultaneamente, em outra thread.

O atributo **routeCache**, por sua vez, tem como objetivo armazenar as rotas com as quais o nó entrou em contato, servindo, portanto, como a tabela de roteamento do nó. Diferentemente do protocolo DSR, entradas dessa tabela não armazenam a rota completa até um nó de destino, mas sim o próximo nó pelo qual a mensagem passará até chegar ao nó de destino.

O atributo **previousNode** armazena o nó pelo qual o processo de descoberta de rota passou imediatamente antes de chegar ao nó corrente. Essa informação será utilizada logo após a descoberta de uma rota para o nó destinatário, pois, diferentemente do protocolo DSR, no momento em que o nó procurado é encontrado, não tem-se a rota completa (origem- destino). Como cada nó conhece apenas o nó referente ao próximo salto, há necessidade de realizar o processo inverso, que irá responder a requisição de rota ao nó de origem, atualizando, pelo caminho, o cache de rotas de todos os nós que fazem parte desta. Note que esse atributo também é do tipo **ThreadLocal**.Isso é necessário para que a integridade da rota que esteja em processo de descoberta não seja afetada por outra descoberta, ocorrendo, simultaneamente, em outra thread.

Já **enableForwarding**, pode ser utilizado para indicar que o nó vai ou não participar do processo de descoberta de rota. Esse controle é útil, por exemplo, no modelo Mobile Phone, já que este conta com dois tipos de nó: aqueles que representam uma antena da rede de telefonia e aqueles que representam os aparelhos celulares. Neste caso, para evitar que as rotas descobertas possuam números identificadores intermediários pertencentes a celulares, o valor desse campo deve ser definido como **false** para todos os nós desse tipo.

```
14      public AODVNode() {
15          super();
16          routeCache = new AODVRouteCache();
17          aodv = new AODVAlgorithm();
18          previousNode = new ThreadLocal<AODVNode>();
19          enableForwarding = true;
20          rreq = new ThreadLocal<AODVRouteRequest>() {
21              @Override
22              protected AODVRouteRequest initialValue() {
23                  return new AODVRouteRequest();
```

```
24                  }
25              };
26          }
```

Além dos métodos de acesso aos atributos descritos acima, **AODVNode** possui um construtor e um método responsável por verificar se um nó, recebido como parâmetro, é ou não vizinho do nó corrente.

13.2.4 AODVRouteCache

Em uma rede que utiliza o protocolo de roteamento AODV, os nós devem ser capazes de armazenar as rotas com as quais já tiveram contato. A esse cache de rotas, damos o nome de tabela de roteamento do nó, que é representada pela classe **AODVRouteCache** e seu único atributo, **routeCache**. Trata-se de um mapeamento, relacionando o identificador de um nó a uma entrada de rota.

```
5   public class AODVRouteCache {
6       private HashMap<Integer, AODVRouteEntry> routeCache;
```

Como métodos, essa classe possui apenas o construtor padrão, sua versão do método **toString()**, um método de acesso a seu atributo e dois métodos para manipulação do cache de rotas, que tem com objetivo armazenar e remover uma rota no cache de rotas do nó.

```
9       public AODVRouteCache() {
10          routeCache = new HashMap<Integer, AODVRouteEntry>();
11      }
12
13      public AODVRouteEntry getRouteEntry(int destination) {
14          return routeCache.get(destination);
15      }
16
17      public void storeRouteEntry(int destination, AODVRouteEntry routeEntry) {
18          routeCache.put(destination, routeEntry);
19      }
20
21      public void removeRoute(int destination) {
22          routeCache.remove(destination);
23      }
24
25      @Override
26      public String toString() {
27          return routeCache.toString();
28      }
```

13.2.5 AODVMessage

Essa classe representa as mensagens enviadas por nós participantes do protocolo de roteamento AODV. Como **AODVMessage** estende a classe **Message** do JSensor, caso seja da vontade do usuário fazer uso do modelo Mobile Phone com roteamento via AODV, as mensagens desse modelo deverão estender **AODVMessage** ao invés de **Message**.

```
5   public class AODVMessage extends Message {
6       private int hops;
7       private AODVNode sender, destination;
```

AODVMessage possui os atributos **sender**, **destination** e **hops**, que representam, respectivamente, o nó remetente da mensagem, o nó destinatário da mensagem e o número de saltos realizados por ela.

13.2.6 AODVAlgorithm

Classe que oferece métodos que realizam ou ajudam a realizar a tarefa principal do protocolo de roteamento AODV, a descoberta de uma rota para um dado nó.

O método **routeDiscovery()**, como o nome sugere, é responsável por determinar, caso seja possível, uma rota para um dado nó. Esse método método possui dois parâmetros, **sourceNode** e **destNode**, representando, respectivamente, os nós de origem e destino.

```
14   public class AODVAlgorithm {
15       public boolean routeDiscovery(AODVNode sourceNode, int destNodeID) {
```

A partir desses dois dados, o processo de descoberta de rota já pode ser realizado. No JSensor, a descoberta de rotas baseia-se no fato de que um nó tem acesso a sua lista de vizinhos. Desta forma, a ideia aqui é fazer, com o auxílio de uma fila, uma busca pelo nó de destino na lista de vizinhos do nó (busca por largura), começando pelo nó de origem. É importante ressaltar que, no caso do modelo Mobile Phone, a fim de evitar a geração de rotas inconsistentes, isto é, rotas do tipo *celular origem*, *antena 1*, *antena 2*, *celular intermediário*, *antena 3*, *celular destino*, o nó de origem deve ser a antena ao qual o celular está conectado. Para garantir que um mesmo nó não seja visitado mais de uma vez, utilizamos a variável **visitedNodes**.

```
20          // Indica que nó de origem já foi visitado
21          visitedNodes.add(sourceNode);
22          // Insere nó de origem na fila
23          queue.add(sourceNode);
24          // Inicializa variável que representa RREQ
25          sourceNode.setRreq(new AODVRouteRequest(sourceNode.getID(), destNodeID, 0));
```

O nó de origem é marcado como já visitado e inserido na fila que controla a ordem na qual os nós serão visitados. Após isso, o atributo do nó de origem que representa o pacote do tipo *Route Request*, é inicializado, a fim de simular o envio deste tipo de pacote.

```
27          while (queue.size() > 0) {
28              AODVNode node = queue.poll();
29
30              if (destNodeID == node.getID()) {
31                  // Nó removido da fila é o nó de destino
32                  routeReply(sourceNode, (AODVNode) Jsensor.getNodeByID(destNodeID), node);
33
34                  return true;
35              } else if (node.enableForwarding == true) {
36                  // Procura pela rota no cache do nó removido da fila
37                  AODVRouteEntry cachedRoute = ((AODVNode) node).getRouteCache().getRouteEntry(
38                      destNodeID);
39
```

```
40                  if (cachedRoute != null && node.getID() != sourceNode.getID() && cachedRoute.
41                      getNext() != node.getPreviousNode().getID()) {
42                      // Atualiza tempo de vida da entra de rota
43                      long ttl = node.getRouteCache().getRouteEntry(destNodeID).getTtl();
44                      node.getRouteCache().getRouteEntry(destNodeID).setTTL(ttl + AODVConstants.
45                          getActiveRouteTime());
46
47                      routeReply(sourceNode, (AODVNode) Jsensor.getNodeByID(destNodeID), node);
48
49                      return true;
50                  } else {
51                      neighbourList = node.getNeighbours().getNodesList();
52
53                      for (Node neighbour : neighbourList) {
54                          if (visitedNodes.contains(neighbour) == false) {
55                              visitedNodes.add((AODVNode) neighbour);
56                              queue.add((AODVNode) neighbour);
57
58                              // RREQ
59                              AODVRouteRequest rreq = new AODVRouteRequest(sourceNode.getID(),
60                                  destNodeID, node.getRreq().getHopCount() + 1);
61                              ((AODVNode) neighbour).setRreq(rreq);
62
63                              // Atualiza cache de rotas do vizinho com rota para o nó de origem
64                              AODVRouteEntry re = new AODVRouteEntry(sourceNode.getID(),
65                                  node.getID(), node.getRreq().getHopCount() + 1,
66                                  AODVMessageTransmissionModel.timeToReach() * AODVConstants.
67                                  getActiveRouteTime() + Jsensor.currentTime);
68                              ((AODVNode) neighbour).getRouteCache().storeRouteEntry(sourceNode.
69                                  getID(), re);
70
71                              // Define o tempo de vida da rota para o nó de origem
72                              setRouteTtlTimer(((AODVNode) neighbour), sourceNode.getID());
73
74                              ((AODVNode) neighbour).setPreviousNode(node);
75                          }
76                      }
77                  }
78              }
79          }
```

A partir daí, tem início o laço principal do método. A cada iteração, é verificado se o nó removido da fila é o nó procurado. Em caso positivo, o método **routeReply()** é acionado e o método **routeDiscovery()** retorna **true**, indicando que uma rota para o nó de destino foi encontrada. É importante observar que, no AODV, diferentemente do DSR, o retorno de **routeDiscovery()** não é uma rota completa para o nó de destino, já que cada um dos nós armazena apenas qual é o próximo salto. O método **routeReply()** será responsável por simular o envio dos pacotes do tipo *Route Reply* aos nós que constituem a rota descoberta. Este método será explicado mais detalhadamente em breve. Caso contrário, será verificado se nó removido da fila possui, em seu cache de rotas, uma rota para o nó de destino. Porém, essa verificação só será realizada caso o atributo **enableForwarding** no nó em questão seja igual a **true**.

Na linha 40, é verificado se o nó retirado da fila tem, em seu cache, uma rota para o nó de destino. Caso seja encontrada uma rota e tal rota não tenha como próximo salto o nó anterior ao nó corrente, o método **routeReply()** é acionado e o método **routeDiscovery()** retorna true. Caso contrário, o processo de descoberta continuará, verificando, um a um, se algum dos vizinhos do nó corrente é ou possui uma rota para o nó de destino.

```
51                      neighbourList = node.getNeighbours().getNodesList();
```

```
52
53                      for (Node neighbour : neighbourList) {
54                          if (visitedNodes.contains(neighbour) == false) {
55                              visitedNodes.add((AODVNode) neighbour);
56                              queue.add((AODVNode) neighbour);
57
58                              // RREQ
59                              AODVRouteRequest rreq = new AODVRouteRequest(sourceNode.getID(),
60                                  destNodeID, node.getRreq().getHopCount() + 1);
61                              ((AODVNode) neighbour).setRreq(rreq);
62
63                              // Atualiza cache de rotas do vizinho com rota para o nó de origem
64                              AODVRouteEntry re = new AODVRouteEntry(sourceNode.getID(),
65                                  node.getID(), node.getRreq().getHopCount() + 1,
66                                  AODVMessageTransmissionModel.timeToReach() * AODVConstants.
67                                  getActiveRouteTime() + Jsensor.currentTime);
68                              ((AODVNode) neighbour).getRouteCache().storeRouteEntry(sourceNode.
69                                  getID(), re);
70
71                              // Define o tempo de vida da rota para o nó de origem
72                              setRouteTtlTimer(((AODVNode) neighbour), sourceNode.getID());
73
74                              ((AODVNode) neighbour).setPreviousNode(node);
75                          }
76                      }
```

Na linha 51, a lista de vizinhos do nó removido da fila é acessado. Para cada vizinho do nó removido da fila que ainda não foi visitado, o definimos como já visitado (linha 55) e o adicionamos na fila (linha 56). Após isso, atualizamos a variável que representa o pacote do tipo *Route Request*, incrementando, em uma unidade, o valor de seu atributo *hops* (linhas 59 a 61) e armazenamos, no cache do vizinho, uma entrada de rota para o nó de origem (linhas 64 e 69). Por fim, o nó removido da fila é atribuído ao campo **previousNode** do vizinho, indicando que o processo de descobrimento desta rota passou por ele, imediatamente, antes de chegar a esse vizinho

```
109     private void routeReply(AODVNode sourceNode, AODVNode destNode, AODVNode node) {
110         AODVNode previousNode = node.getPreviousNode();
111         int nextNode = node.getID();
112
113         for (int i = 0; i < node.getRreq().getHopCount(); i++) {
114             AODVRouteEntry re = new AODVRouteEntry(destNode.getID(), nextNode, i + 1,
115                 AODVMessageTransmissionModel.timeToReach() * AODVConstants.getActiveRouteTime() +
116                 Jsensor.currentTime);
117             previousNode.getRouteCache().storeRouteEntry(destNode.getID(), re);
118             setRouteTtlTimer(previousNode, destNode.getID());
119             nextNode = previousNode.getID();
120             previousNode = previousNode.getPreviousNode();
121         }
122     }
```

Conforme citado anteriormente, o método **routeReply()** é acionado sempre que uma rota para o nó de destino for encontrada. Sua função é atualizar o cache de rotas dos nós que compõem tal rota, de forma a fazer com que o envio da mensagem (origem-destino) ocorra com sucesso. Baseando-se no número de saltos existente no atributo que representa o pacote do tipo *Route Request* do parâmetro *node* (que pode ser o próprio nó de destino ou um nó que possui uma rota para o nó de destino), esse método consegue fazer o caminho inverso àquele realizado durante a descoberta da rota, ou seja, parte do nó de destino e vai até o nó de origem. Esse método possui duas variáveis locais, que o auxiliam nesse percurso. A primeira, **previousNode**, que, inicialmente, possui o mesmo valor existente no atributo

previousNode do parâmetro *node*. E **nextNode**, que, também inicialmente, representa o número identificador do parâmetro *node*. Já no laço, no primeiro momento, é criada uma variável que representa uma entrada de rota para o nó de destino. Mais especificamente, tal variável representa o processamento que seria feito pelo nó quando o pacote do tipo *Route Reply* fosse recebido por ele (observe os argumentos passados para o construtor de AODVRouteEntry, na linha 114). Na próxima linha, a entrada de rota recém criada é adicionada ao cache do nó representado por **previousNode**. Após isso, os valores das variáveis locais à função são atualizadas, isto é, **nextNode** assume o valor de **previousNode**, enquanto **previousNode** torna-se o nó anterior ao **previousNode**. Com isso, os caches de rotas de todos os nós envolvidos na descoberta de uma determinada rota terão sido devidamente atualizados.

```
84      public boolean sendMessage(AODVNode from, AODVNode to, AODVMessage msg) {
85          if (from.isNeighbour(to)) {
86              from.unicast(msg, to);
87
88              // Mensagem foi enviada
89              return true;
90          } else {
91              // Mensagem não foi enviada
92              return false;
93          }
94      }
95
96      public void newRouteDiscovery(AODVNode node, AODVMessage msg) {
97          AODVResendTimer rt = new AODVResendTimer();
98          rt.startRelative(1, node);
99          rt.setMessage(msg);
100     }
101
102     public void removeRouteFromPrecursors(AODVNode node, int destination) {
103         SortedSet<Node> neighbourList = node.getNeighbours().getNodesList();
104
105         for (Node neighbour : neighbourList)
106             ((AODVNode) neighbour).getRouteCache().removeRoute(destination);
107     }
```

AODVAlgorithm conta também com os métodos **sendMessage()**, **newRouteDiscovery()**, **removeRouteFromPrecursors()**.

Assim, como visto no protocolo DSR, o método sendMessage() realiza o envio de uma mensagem, via *unicast*, de um nó para outro, caso os nós envolvidos no envio sejam vizinhos. Já newRouteDiscovery(), é usado quando é necessário descobrir uma nova rota para um determinado nó de destino, através da criação de um evento de timer. O método **removeRouteFromPrecursors()** remove, do cache de rotas de cada um dos vizinhos de um dado nó, a entrada referente ao destino, seu segundo parâmetro. No Mobile Phone, esse método é acionado por objetos do tipo Antenna, quando estes percebem que há algum problema durante o envio de uma mensagem

O cache de rotas dos nós são úteis, pois podem evitar que novas descobertas de rota sejam feitas sem necessidade. Porém, com o intuito de evitar que essas estruturas de dados contenham rotas que já não são utilizadas há algum tempo ou que elas ocupem muito espaço em memória, é necessário utilizar algum mecanismo que controle a quantidade de rotas nos caches. O método **setRouteTtlTimer()** é o método responsável por realizar tal controle.

Rotas possuem um campo (ttl), que indica seu tempo de vida. Quando uma nova rota é criada, seu tempo de vida é definido como o tempo atual da simulação + o valor de **activeRouteTime** (seção anterior). Sempre que uma nova rota é incluída (em um cache de rotas) ou utilizada, um evento de timer é criado. Esse evento será responsável por verificar se determinada rota pode ser considerada inativa (não foi mais acessada após sua inclusão no cache) e, em caso positivo, por removê-la do cache (linha 126). No momento da criação desse tipo de evento, é utilizado o método startRelative(), da classe **TimerEvent**, significando que o tempo em que ele será disparado é baseado no tempo atual da simulação + activeRouteTime unidades de tempo (linha 127).

13.2.7 AODVResendTimer

A classe **AODVResendTimer** representa um evento de timer que deve ser usado quando a tentativa de envio de uma mensagem resultar em falha. Isso pode acontecer, por exemplo, quando o nó que seria o próximo a receber a mensagem encontra-se fora do alcance do nó remetente.

Essa classe possui apenas um atributo, **message**, que representa a mensagem que deve ser reenviada

```
8   public class AODVResendTimer extends TimerEvent {
9       private AODVMessage message;
```

Como **AODVResendTimer** estende a classe **TimerEvent** do JSensor, deve implementar seu método **fire()**, pois ele será responsável por tentar encontrar uma nova rota para o nó de destino da mensagem.

```
15      @Override
16      public void fire() {
17          boolean routeFound = ((AODVNode) this.node).getAodv().
18              routeDiscovery((AODVNode) this.node, message.getDestination().getID());
19
20          if (routeFound == true) {
21              AODVRouteEntry re = ((AODVNode) this.node).getRouteCache().
22                  getRouteEntry(message.getDestination().getID());
23              ((AODVNode) this.node).aodv.sendMessage((AODVNode) this.node,
24                  (AODVNode) Jsensor.getNodeByID(re.getNext()), message);
25          } else {
26              Jsensor.log("[No route] time: " + Jsensor.currentTime + " nodeID: " +
27                  message.getDestination().getID());
28              Jsensor.log("[No route] position: " + message.getDestination().
29                  getPosition().getPosX() + ", " + message.getDestination().
30                  getPosition().getPosY());
31          }
32      }
```

Na linha 17, o método que realiza a busca por uma rota ao nó de destino é acionado. Caso uma rota seja encontrada, será possível acessá-la através do cache de rotas do nó que solicitou a descoberta (conforme explicado na seção anterior - routeReply()). A rota então é acessada (linha 21) e uma referência a ela é atribuída à variável auxiliar **re**. Na linha 23, a mensagem é reenviada ao próximo no da rota. Caso a rota não seja encontrada, esse fato é registrado no arquivo de log do JSensor.

13.2.8 AODVMessageTransmissionModel

Classe que possui apenas um método estático e serve para informar quantas unidades de tempo do JSensor uma mensagem gasta para sair de um nó remetente a e chegar a um nó destinatário b.

```
3  public class AODVMessageTransmissionModel {
4      private static final int time = 1;
5
6      public static int timeToReach() {
7          return time;
8      }
9  }
```

13.2.9 AODVUtils

AODVUtils é uma simples classe, que tem como objetivo encapsular classes utilitárias para o AODV. No momento, há apenas uma classe interna, **AODVConstants**. Seu único atributo, **activeRouteTime** indica o tempo de vida de cada rota, em unidades de tempo do JSensor.

```
3  public class AODVUtils {
4      public static class AODVConstants {
5          private static final int activeRouteTime = 3000;
6
7          public static int getActiveRouteTime() {
8              return activeRouteTime;
9          }
10     }
11 }
```

13.2.10 AODVRemoveRouteEntryTimer

A classe **AODVRemoveRouteEntryTimer** representa um evento de timer e deve estender a classe **TimerEvent** do JSensor. **AODVRemoveRouteEntryTimer** deve ser usada para verificar a necessidade de remover uma determinada rota do cache de rotas de um dado nó.

```
6  public class AODVRemoveRouteEntryTimer extends TimerEvent {
7      private int entryToRemove;
8
9      @Override
10     public void fire() {
11         AODVRouteEntry re = ((AODVNode) this.node).getRouteCache().getRouteEntry(entryToRemove);
12
13         if (re != null && re.getTtl() == Jsensor.currentTime )
14             ((AODVNode) this.node).getRouteCache().removeRoute(entryToRemove);
15     }
16
17     public void setEntryToRemove(int entryToRemove) {
18         this.entryToRemove = entryToRemove;
19     }
20 }
```

Essa classe possui apenas um atributo, **entryToRemove**, que indicará a rota a ser removida do cache do nó questão. Como **TimerEvent** é sua super classe, **AODVRemoveRouteEntryTimer** deverá

implementar o método **fire()**, já que esse é o método que será chamado quando o evento for acionado. Na linha 11, procuramos pela rota que leva até o destino indicado pela variável **entryToRemove**. Caso a rota exista e o valor de seu campo **ttl** seja igual ao tempo atual da simulação (linha 13), ela será removida do cache (linha 14).

Obs.: É importante ressaltar que, para que o protocolo de roteamento AODV funcione no modelo Mobile Phone, os exemplos de implementação demonstrados anteriormente deverão ser ajustados. Nas próximas seções, mostraremos como esse ajuste deverá ser feito.

13.2.11 SMSMessage

Como citado na seção relativa à classe **AODVMessage** (13.2.5), classes que representam as mensagens enviadas no protocolo de roteamento AODV deverão estender **AODVMessage**.

```
10  public class SMSMessage extends AODVMessage{
11      private String message;
12
13      public SMSMessage(AODVNode sender, AODVNode destination, int hops, short chunck) {
14          super(sender, destination, hops, chunck);
15
16          this.message = "";
17      }
18
19      public SMSMessage(AODVNode sender, AODVNode destination, int hops, long ID,
20                        String message) {
21          super(ID, sender, destination, hops);
22
23          this.message = message;
24      }
25
26      public String getMessage() {
27          return message;
28      }
29
30      public void setMessage(String message) {
31          this.message = message;
32      }
33
34      @Override
35      public SMSMessage clone() {
36          return new SMSMessage(this.getSender(), this.getDestination(),
37                                this.getHops() + 1, this.getID(), this.message);
38      }
39  }
```

Nesse exemplo, **SMSMessage** possui apenas um atributo, **message**, que contém a mensagem propriamente dita e dois construtores. Note que o tipo dos nós origem e destino usado nos construtores é **AODVNode** e não mais **Node**.

13.2.12 SMSTimer

A primeira modificação a ser feita nessa classe está relacionada ao tipo dos nós utilizados pelo protocolo de roteamento. Quando a mensagem for criada, no método **fire()**, será necessário fazer uma conversão explícita de tipos, já que o tipo esperado pelo construtor da classe **SMSMessage** agora é **AODVNode**.

```
35                  SMSMessage msg = new SMSMessage((AODVNode) this.node, destNode, 0,
36                                                  this.node.getChunk());
```

O método que realiza a descoberta da rota é feita pelo método **fire()**.

Importante.: No Mobile Phone, o argumento que indica o nó de origem para o método de descoberta da rota deverá ser o nó que representa a antena do nó que deseja enviar a mensagem. Tal restrição é necessária para que rotas inconsistentes não sejam geradas, isto é, rotas do tipo *celular origem*, *antena 1*, *antena 2*, *celular intermediário*, *antena 3*, *celular destino*. A linha 28 mostra como é possível obter o nó que representa a antena de um outro nó.

Caso a rota tenha sido descoberta, **routeDiscovery()** retornará true. Nesse momento, o cache de rotas dos nós que formam tal rota estarão preenchidos com as entradas de rota necessárias para alcançar o nó destinatário, exceto pelo cache do nó remetente, pelo motivo descrito no parágrafo anterior. Na linha 46, é criada uma entrada de rota para o nó destinatário. Na próxima linha, essa rota é armazenada no cache de rotas do nó remetente. Sendo assim, o envio da mensagem já pode então ser realizado (linha 50).

13.2.13 Antenna

O método **handleMessages()**, que é o método responsável por realizar o processamento das mensagens recebidas pelo nó, também deverá ser alterado para que contemple as características do protocolo de roteamento AODV.

```
30                  if (!this.messagesIDs.contains(smsMessage.getID())) {
31                      this.messagesIDs.add(smsMessage.getID());
32                      smsMessage.setMessage(smsMessage.getMessage().concat(" - " + Integer.
33                          toString(this.ID)));
34
35                      AODVRouteEntry routeEntry = this.getRouteCache().getRouteEntry(smsMessage.
36                          getDestination().getID());
37
38                      if (routeEntry != null) {
39                          // Envia mensagem para o próximo nó, via unicast
40
41                          if (routeEntry.getNext() == smsMessage.getDestination().getID()) {
42                              if (this.getNeighbours().getNodesList().contains(smsMessage.
43                                  getDestination())) {
44                                  // Antena ainda pode alcançar nó de destino
45                                  this.unicast(m, Jsensor.getNodeByID(smsMessage.getDestination().
46                                      getID()));
47                              } else {
48                                  Jsensor.log("[No signal] time: " + Jsensor.currentTime +
49                                          " nodeID: " + this.getID());
50                                  Jsensor.log("[No signal] position: "+ this.getPosition().getPosX()
51                                      + ", " + this.getPosition().getPosY());
52                              }
53                          } else {
54                              AODVNode nextNode = (AODVNode) Jsensor.getNodeByID(routeEntry.
55                                  getNext());
56                              boolean sendResult = this.aodv.sendMessage(this, nextNode, smsMessage);
57
58                              if (sendResult == false) {
59                                  Jsensor.log("[Route error] time: " + Jsensor.currentTime +
60                                          " route to node " + smsMessage.getDestination().getID() +
```

```
61                                          " must be rediscovered");
62
63                                      this.aodv.removeRouteFromPrecursors(this, nextNode.getID());
64                                      this.aodv.newRouteDiscovery(this, smsMessage);
65                                  }
66                              }
67                          } else {
68                              this.aodv.newRouteDiscovery(this, smsMessage);
69                          }
70                      }
```

Caso a mensagem em processamento seja nova, o nó corrente deverá encaminhá-la ao próximo nó da rota. Para isso, o nó corrente verificará se possui uma rota para o próximo nó (linhas 35 e 36). Em caso positivo, verificará se o campo que indica o identificador do próximo dessa rota possui o identificador do nó destinatário da mensagem (linha 41). De acordo com esse teste, a mensagem será enviada ao nó destinatário da mensagem (linha 45) ou ao nó que representa a próxima antena da rota (linhas 54 a 56). Caso a mensagem não possa ser enviada ao nó destinatário, **handleMessages()** registrará tal fato no arquivo de log do JSensor (linhas 48 a 51). Caso a mensagem não possa ser encaminhada à próxima antena da rota, **handleMessages()**, após registrar tal fato no arquivo de log do simulador (linhas 59 a 61), acionará os métodos **removeRouteFromPrecursors()** (linha 63) e newRouteDiscovery() (linha 64), responsáveis por atualizar o cache de rotas dos nós vizinhos ao nó corrente e por realizar uma nova tentativa de descoberta de rota e envio da mensagem, respectivamente.

Caso o nó corrente não possua uma entrada de rota para o próximo nó, em seu cache, uma nova tentativa de descoberta de rota será realizada (linha 68).

13.2.14 CellPhone

Nessa classe, a diferença entre o modelo Mobile Phone padrão e o Mobile Phone com roteamento via AODV, ocorrerá no momento em que o nó de destino precisar enviar uma mensagem de volta ao remetente (a partir da linha 37), simulando assim uma troca de mensagens mais intensa.

```
36                      // Probabilidade de responder mensagem é igual a 0.8
37                      if (this.getRandom().nextDouble() < 0.8) {
38                          this.findAntenna();
39
40                          if (this.getMyAntenna() > -1) {
41                              SMSMessage replyMessage = new SMSMessage(this, smsMessage.getSender(),
42                                  0, this.getChunk());
43                              smsMessage.setMessage("This is the message number " + this.getChunk()
44                                  + " created by the node " + this.getID() + " path " + this.getID());
45                              AODVRouteEntry routeEntry = this.getRouteCache().getRouteEntry(
46                                  smsMessage.getSender().getID());
47
48                              if (routeEntry != null) {
49                                  AODVNode nextNode = (AODVNode) Jsensor.getNodeByID(routeEntry.
50                                      getNext());
51                                  boolean sendResult = this.aodv.sendMessage(this, nextNode,
52                                      replyMessage);
53
54                                  if (sendResult == false) {
55                                      this.aodv.removeRouteFromPrecursors(this, nextNode.getID());
56                                      this.aodv.newRouteDiscovery((AODVNode) Jsensor.getNodeByID(
57                                          this.getMyAntenna()), replyMessage);
58                                  }
```

```
59                                  } else {
60                                      this.aodv.newRouteDiscovery((AODVNode) Jsensor.getNodeByID(this.
61                                          getMyAntenna()), replyMessage);
62                                  }
63                              } else {
64                                  Jsensor.log("[No signal] time: " + Jsensor.currentTime +
65                                          " node " + this.getID() + " has no signal");
66                              }
67                          }
```

Caso o nó corrente, que agora vai agir como nó remetente, esteja conectado a uma antena (linha 40), uma mensagem de resposta é criada (linhas 41 e 42) e é também verificado se o nó corrente possui uma rota para o nó de destino dessa mensagem recém criada. Em caso positivo, a mensagem será encaminhada ao próximo nó dessa rota (linhas 49 a 52). Se esse envio falhar, isto é, caso o próximo nó da rota não possa ser alcançado (devido à mobilidade), essa rota é removida do cache de todos os precursores (linha 55) e uma nova tentativa de envio da mensagem será então realizada, através de uma chamada ao método **newRouteDiscovery()**, da classe **AODVAlgorithm** (linhas 56 e 57).

Caso o nó corrente não possua uma rota para o nó de destino da mensagem recém criada, uma nova tentativa envio da mensagem também será realizada (linhas 60 e 61).

Outro detalhe importante e que deve ser observado, é o valor do atributo **enableForwarding**, que deve ser definido como false para nós do tipo **CellPhone**. Apenas para relembrar, isso é necessário para que rotas geradas não fiquem inconsistentes, isto é, uma que contém um nó do tipo celular como nó intermediário.

```
106             /*A fim de evitar rotas do tipo:
107             celular (origem), antena, celular, antena, celular (destino) nós do tipo
108             CellPhone não devem encaminhar a rota contida no atributo forwardRoute*/
109             this.setEnableForwarding(false);
```