Guia de Programação JSensor

Exemplo Flooding

Sumário

1 Introdução

JSensor é um simulador para redes de sensores sem fio em 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).

No arquivo JSensorInstallationGuide.pdf, explicamos passo a passo como realizar a instalação do JSensor. No diretório **projects/default**, encontram-se as classes base para criação de novos modelos que o desenvolvedor queira implementar.

Exemplos de modelos de simulações para o JSensor podem ser encontrados no diretório **jsensor/src/projects**. Neste documento, será demonstrado um exemplo que utiliza o modelo flooding para envio de mensagens, com distribuição aleatória de nós pela área de simulação e sem mobilidade. Tal modelo encontra-se no diretório **jsensor/src/projects/flooding**.

Cada seção deste documento abordará uma das classes que compõem o modelo flooding.

2 FloodingNode

Essa classe representa um nó do tipo flooding e deve estender a classe **Node**. Nesta nova classe, utilizamos um campo do tipo LinkedList<Long> para representar os identificadores das mensagens já recebidas por este nó, desde sua criação.

```
17  public class FloodingNode extends Node{
18      public LinkedList<Long> messagesIDs;
```

A classe **FloodingNode** implementa o método **onCreation()** da classe **Node**, que é responsável por inicializar um novo nó no JSensor, permitindo que o usuário decida o que fazer na fase de inicialização dos nós.

```
51      @Override
52      public void onCreation() {
53          //inicializa a lista de mensagens recebidas pelo nó
54          this.messagesIDs = new LinkedList<Long>();
55
56          //envia uma mensagem caso o nó seja um dos selecionados
57          if(this.ID < 10){
58              int time = 1;
59              FloodingTimer ft = new FloodingTimer();
60              ft.startRelative(time, this);
61          }
62      }
```

É possível definir a regra utilizada para envio de mensagens. No exemplo de código acima, apenas nós cujo número do identificador (ID) são menores que 10 irão criar eventos de envio de mensagem. Eventos são criados a partir de objetos do tipo **timer**, que serão detalhados na seção 3. Neste exemplo, um evento representa o envio de uma mensagem, mas o usuário poderá definir o evento que lhe for mais conveniente.

```
20      @Override
21      public void handleMessages(Inbox inbox) {
22          while(inbox.hasMoreMessages())
23          {
24              Message message = inbox.getNextMessage();
25
26              if(message instanceof FloodingMessage){
27                  FloodingMessage floodingMessage = (FloodingMessage) message;
28
29                  if(this.messagesIDs.contains(floodingMessage.getID())){
30                      continue;
31                  }
32
33                  this.messagesIDs.add(floodingMessage.getID());
34
35                  if(floodingMessage.getDestination().equals(this)){
36                      Jsensor.log("time: "+ Jsensor.currentTime +
37                                  "\t nodeID: " +this.ID+
38                                  "\t receivedFrom: " +floodingMessage.getSender().getID()+
39                                  "\t hops: "+ floodingMessage.getHops() +
40                                  "\t msg: " +floodingMessage.getMsg().concat(this.ID+""));
41                  }
42                  else{
43                      //adiciona ID do nó na mensagem e repassa, via multicast.
44                      loodingMessage.setMsg(floodingMessage.getMsg().concat(this.ID+ " - "));
45                      this.multicast(message);
46                  }
47              }
48          }
49      }
```

Outro método importante que deve ser implementado na classe **FloodingNode** é o **handleMessages**, cujo papel é realizar o controle das mensagens recebidas pelos nós. Para cada mensagem na caixa de entrada de um nó, **handleMessages()** verifica se ela é do tipo FloodingMessage (linha 26) e se é uma mensagem nova (linha 29), 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 35). Em caso positivo, o recebimento da mensagem é registrado. Caso contrário, a mensagem é encaminhada aos seus vizinhos, via multicast (linha 45).

O método **Jsensor.log()**, conforme visto na linha 36, serve para popular o arquivo de log do simulador JSensor. A geração desse arquivo de log é opcional e deve ser definida pelo usuário no arquivo de configurações. Caso o usuário opte pela geração de log, tais logs serão criados no diretório **results**.

3 FloodingTimer

A classe **FloodingTimer** é a responsável por implementar o evento de timer do modelo flooding e, por isso, deve estender a classe TimerEvent.

```
14  public class FloodingTimer extends TimerEvent{
15
16      @Override
17      public void fire() {
18
19          Node destination = this.node.getRandomNode("FloodingNode");
20          while(true){
```

```
21                  if(destination == null) {
22                      destination = this.node.getRandomNode("FloodingNode");
23                      continue;
24                  }
25
26                  if(this.node == destination){
27                      destination = this.node.getRandomNode("FloodingNode");
28                      continue;
29                  }
30                  break;
31              }
32
33              FloodingMessage message = new FloodingMessage(this.node, destination, 0, "" +
34                                                            this.node.getID(),
35                                                            this.node.getChunk());
36
37              String messagetext = "Created by the sensor: "+Integer.toString(this.node.getID()) +
38                                   " Path: ";
39
40              message.setMsg(messagetext);
41              Jsensor.log("time: "+ Jsensor.currentTime +"\t nodeID: "+this.node.getID()+
42                          "\t sendTo: " +destination.getID());
43
44              this.node.multicast(message);
45          }
46      }
```

Essa classe deverá sobrescrever o método **fire()**, já que este é o método acionado quando o evento é disparado.

No exemplo deste documento, um evento consiste em um envio de mensagens a partir de um certo nó, via multicast. Para realizar esse envio, o método **fire()** seleciona, aleatoriamente, um nó válido, isto é, não nulo, que será o nó de destino da mensagem (linha 19). Aqui, o tipo do nó precisa ser **FloodingNode**. Também é importante observar que nós origem e destino devem ser diferentes. Após essas verificações, uma nova mensagem é criada e seus atributos são definidos (nó de origem, nó de destino, número de saltos e identificador da mensagem). Essa mensagem é enviada, via multicast.

O nó retornado por **Node.getRandomNode()** pode ser obtido através do uso da semente padrão do JSensor ou de uma definida pelo usuário.

4 FloodingMessage

FloodingMessage implementa as mensagens utilizadas no modelo e, portanto, deve estender a classe **Message** do JSensor e sobrescrever o método **clone()**, que deverá realizar uma cópia completa do objeto (deep copy).

```
73      @Override
74      public Message clone() {
75          return new FloodingMessage(msg, sender, destination, hops+1,this.getID());
76      }
```

No exemplo deste documento, a classe **FloodingMessage** possui os seguintes atributos (e seus respectivos métodos de acesso) e construtores.

```
12      private String msg;
13      private Node sender;
14      private Node destination;
15      private int hops;
16      short chunk;
17
18      public FloodingMessage(Node sender, Node destination, int hops, String message, short chunk) {
19          //cria um novo ID
20          super(chunk);
21          this.msg = message;
22          this.sender = sender;
23          this.destination = destination;
24          this.hops = hops;
25          this.chunk = chunk;
26      }
27
28      private FloodingMessage(String msg, Node sender, Node destination, int hops, long ID) {
29          //o ID é definido com base no valor recebido como parâmetro
30          this.setID(ID);
31          this.msg = msg;
32          this.sender = sender;
33          this.destination = destination;
34          this.hops = hops;
35      }
```

5 RandomDistribution

Esta classe implementa a forma como os nós serão distribuídos na área coberta pela de simulação e deve estender a classe **DistributionModelNode**.

```
12  public class RandomDistribution extends DistributionModelNode
13  {
14      @Override
15      public Position getPosition(Node s) {
16          return new Position(s.getRandom().nextInt(Configuration.dimX),
17                              s.getRandom().nextInt(Configuration.dimY));
18      }
19  }
```

Classes que implementam um modelo de distribuição de nós precisam sobrescrever o método **getPosition()**, pois este retorna um objeto do tipo **Position**, que indica as coordenadas do nó que será criado. A classe **Position** possui apenas dois atributos, X e Y.

Por padrão, distribuições de nós de forma aleatória e em grade já são fornecidas pelo JSensor. Para utilizá-las, basta atribuir o valor *Random* ou *Gride* ao do parâmetro **distribution_model** (arquivo de configuração do modelo).

Neste exemplo, **getPosition()** obtém as coordenadas de forma aleatória (o que não é necessário, uma vez que o simulador já oferece tal tipo de distribuição), mas o usuário é livre para implementar qualquer outra estratégia de distribuição mais adequada a sua necessidade.

6 FloodingNodeConectivity

É a classe responsável pela conexão entre os nós e deve estender a classe **ConnectivityModel** do JSensor. O método **isConnected()** deve ser obrigatoriamente sobrescrito, caso contrario, não haverá comunicação entre nenhum nó. Já **isNear()** é opcional e, caso não seja sobrescrito será utilizado o método default (UDG).

```
11  public class FloodingNodeConectivity extends ConnectivityModel
12  {
13      @Override
14      public boolean isConnected(Node from, Node to) {
15          return true;
16      }
17
18
19      @Override
20      public boolean isNear(Node n1, Node n2) {
21          return super.isNear(n1, n2);
22      }
23
24  }
```

O método **isConnected()** indica se um nó (from) está conectado a outro (to) de forma lógica. Neste modelo de exemplo, todos os nós podem se conectar e, por isso, seu retorno é sempre **true**.

Para verificar a conexão física dos nós utilizamos o método **isNear()**. JSensor implementa por padrão o modelo *Unit Disk Graph*(UDG), onde o nó *n2* está próximo e, portanto, conectado ao nó *n1* caso *n2* esteja dentro do raio de comunicação de *n1*. O usuário pode definir outros modelos de conexão implementando esta classe .

7 MessageTransmission

MessageTransmission deve estender a classe **MessageTransmissionModel** do JSensor e sobrescrever seu método **timeToReach**. Seu objetivo definir o número de unidades de tempo necessárias para que uma mensagem enviada por um nó alcance o nó destino.

```
7   public class DefaultMessageTransmissionModel extends MessageTransmissionModel{
8
9           @Override
10          public float timeToReach(Node startSensor, Node endSensor, Message msg) {
11                  return 9;
12          }
13
14  }
```

Neste exemplo, qualquer mensagem enviada gastará 9 unidades de tempo para chegar ao destino. O usuário é livre para determinar uma estratégia que lhe seja mais conveniente, podendo, inclusive, variar o tempo com base nos nós envolvidos na troca de mensagem (ou com base na própria mensagem enviada).

8 ReliableDelivery

Vários fatores podem afetar a estabilidade de uma rede, fazendo com que os pacotes enviados por ela cheguem ou não até seu destino. A classe **ReliableDelivery**, que estende a classe **ReliabilityModel** do JSensor, representa a confiabilidade da rede simulada.

```
6   public class ReliableDelivery extends ReliabilityModel{
7
8           @Override
9           public boolean reachesDestination(Packet p) {
10                  return true;
11          }
12
13  }
```

A estratégia que ditará tal confiabilidade deverá ser definida em **reachesDestination()**, que determina a chance de sucesso de cada pacote enviado pela rede. Este é o único que método deve ser sobrescrito por classes que desejam implementar modelos de confiabilidade. Para cada pacote enviado pela rede, esse método é acionado e deve decidir se o pacote será ou não derrubado.

9 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.

Após a criação do modelo, é hora de executar a simulação. Para isso, certifique-se de que o modelo criado esteja no diretório **jsensor/src/projects** e leia os guias JSensorInstallationGuide e JSensorUserGuide.