UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
REDES SEM FIO

SIMULAÇÃO - REDES AD HOC

ANTONIO JONAS GONÇALVES DE OLIVEIRA 2016021023

THIAGO ALVES DE ARAUJO 2016019787

JOÃO PESSOA – PB
OUTUBRO 2019

1 – Introdução

O projeto consiste em desenvolver a simulação de uma rede sem fio de topologia Ad Hoc. O projeto foi inteiramente desenvolvido na linguagem python e o protocolo de roteamento escolhido foi o *Ad hoc On-demand distance vector routing* (AODV).

2 – Camadas

Dentre as cinco camadas da rede apresentadas no modelo hibrido do *Tanenbaum* (modelo de referência adotado pelo livro) foram implementadas as seguintes camadas: Camada de rede, Camada de enlace e Camada física. Abaixo falaremos brevemente sobre cada camada, a abordagem utilizada e as soluções implementadas.

2.1 – Camada de Rede

A camada de rede é responsável por controlar as operações da rede. Uma das suas principais funções é fornecer o roteamento para que um pacote possa chegar ao destino desejado (mesmo que isso signifique percorrer diversos nós intermediários).

Como dito anteriormente, o protocolo de roteamento escolhido foi o AODV, protocolo que é destinado para redes móveis e muito indicado para redes de alta mobilidade.

Como o AODV se trata se um protocolo de roteamento reativo, ele utiliza *pacotes de roteamento* para realizar a descoberta as rotas desejadas.

O pacote *Route request* (Rreq) é responsável por solicitar rota para um determinado destino. Ele é difundido (inundado) por toda a rede até

que o destinatário o receba e mande uma resposta utilizando um pacote *Route reply* (Rrep) para o remetente do pacote.

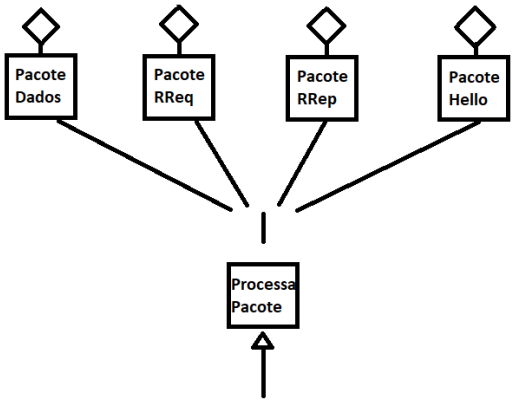
Outro pacote de roteamento utilizado pelo protocolo AODV é o pacote *Hello*. Ele é enviado periodicamente entre os vizinhos e serve para a manutenção das rotas.

A maior parte do projeto está localizada na camada de rede tendo em vista que as demais camadas foram implementadas utilizando uma abordagem mais simples e direta. Abaixo está descrito o funcionamento das principais funções implementadas.

2.1.1 – Processa pacote

Função responsável por receber um pacote (seja ele um pacote de dados o pacote de roteamento) da camada de enlace e encaminha para as funções responsáveis por examinar cada tipo especifico de pacote.

```
def Processa_Pacote(self, Pacote):
```



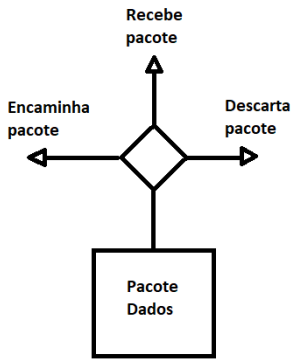
2.1.2 – Recebe pacote de dados

Esta função é responsável por receber um pacote de dados. Ao receber um pacote ela pode tomar três decisões: receber, encaminhar ou descartar.

Caso o pacote que chegou seja para o nó que recebeu (ou seja, o destino do pacote seja o Nó) a função envia este pacote para a camada superior. Como a cama de transporte não foi implementada, o pacote passa direto para o Nó.

Caso o destino não seja o Nó que recebeu, ele identifica o destino, procura o "próximo passo", altera o campo next_step e envia para a camada de enlace. Caso o Nó não tenha uma rota para o destino desejado, ele envia um RReq solicitando rota para este destino. Caso o número de sequência do pacote já esteja entre a lista o pacote é descartado.

```
def Recebe_Dados(self, PacDados): ...
```

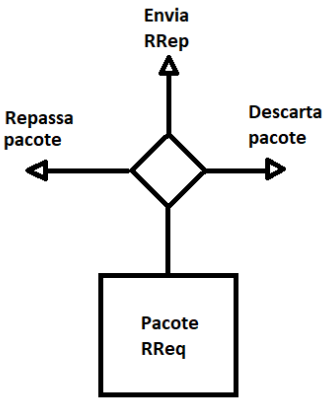


2.1.3 – Recebe pacote RReq

Esta função é responsável por receber o pacote de roteamento RReq. Ao receber um pacote ela pode tomar três decisões: Enviar um RRep, repassar para os vizinhos ou descartar.

Casa o destino do RReq seja o Nó que está recebendo, a função chama a função responsável por criar o pacote RRep e não repassa mais o pacote. Caso o destino não seja o Nó que recebeu, a função simplesmente encaminha o pacote para a camada de enlace. Por fim, caso o pacote possua um número de sequência já conhecido anteriormente a função simplesmente descarta o pacote.

```
def Recebe_Request(self, PacReq):
```



2.1.4 – Recebe pacote RRep

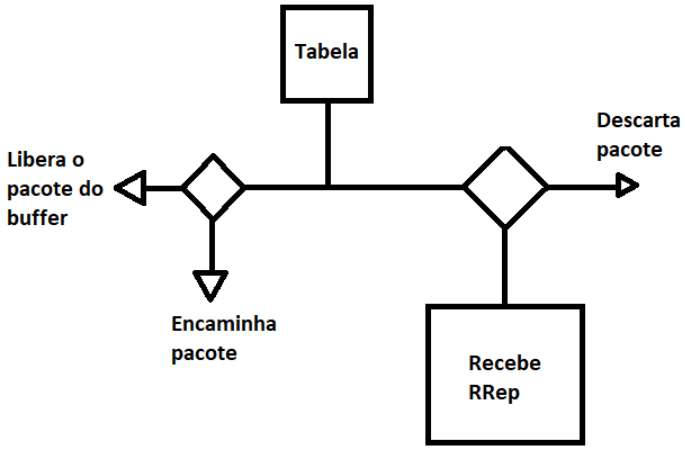
Esta função é responsável por receber o pacote de roteamento RRep. Ao receber um pacote ela pode tomar três decisões: Descartar, repassar ou liberar um pacote do buffer.

Caso o destino do pacote seja o nó que recebeu, ele cria a nova rota recebida e libera do buffer o pacote que estava aguardando aquela rota.

Caso o pacote não seja para o nó que recebeu, ele envia para o próximo passo. Antes de encaminhar, a função incrementa um salto no pacote. O total de saltos que o RRep realiza é utilizado para que o remetente do RReq saiba o custo da nova rota que está sendo criada.

É valido ressaltar que neste ponto do projeto foi adotado que toda rota é bilateral, ou seja, se o ponto A pode enviar para o ponto B, o inverso é valido. Logo, o pacote RRep retorna pela mesma rota que o RReq tomou.

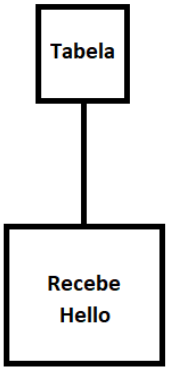
```
def Recebe_Reply(self, PacResp):
```



2.1.5 – Recebe pacote Hello

Esta função é responsável por receber o pacote Hello. Ao receber o pacote a função atualiza a tabela com a rota recebida.

```
def Recebe_Hello(self, Pacote_Hello):
```



2.1.6 – Envia Hello

Função responsável por criar um pacote hello. O pacote não possui um destino especifico e contém um campo de dados com as seguintes informações: [meu MAC, meu MAC, 1 salto]. Isso significa dizer que *"se o destino sou eu, seu próximo passo sou eu e o custo de envio é de um salto"*.

```
def Envia_Hello(self):
```

2.1.7 – Envia Reply

Função responsável por criar o pacote RRep. Ela recebe como parâmetro o destino (Origem do RReq) e uma rota (Rota percorrida pelo RReq).

```
def Envia_Reply(self, Destino, Rota):
```

2.1.8 – Envia Request

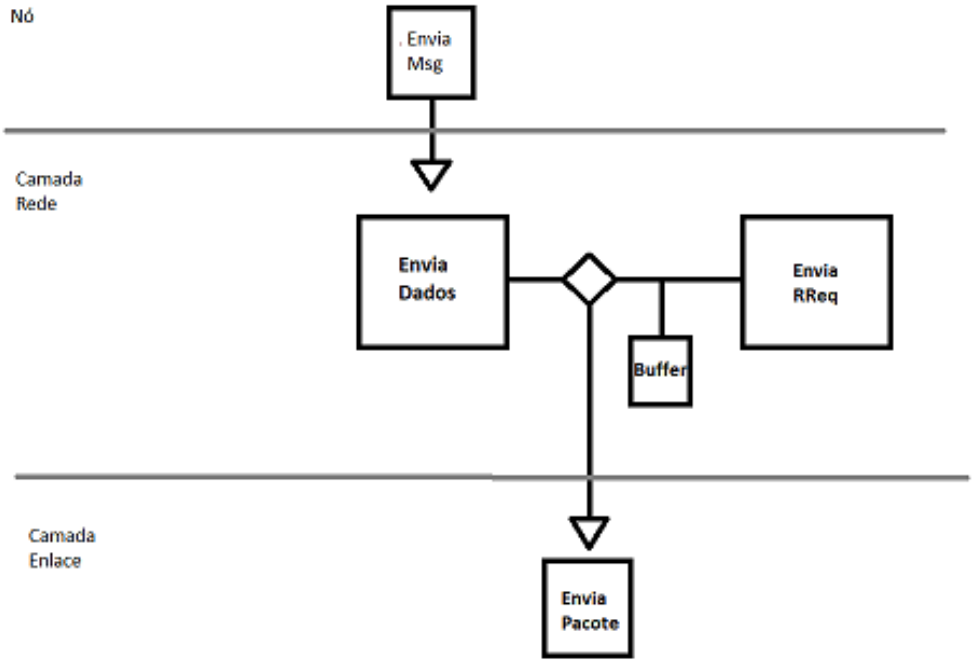
Função responsável por criar o pacote RReq. Ela recebe como parâmetro apenas o destino desejado (rota que está sendo solicitada) e envia o pacote para a camada de enlace.

```
def Envia_Request(self, Destino):
```

2.1.9 – Envia Dados

Função responsável por receber um pacote de dados da camada superior. Como a camada de aplicação não foi implementada, o pacote é criado diretamente na classe Nó. Nela, os dados são inseridos juntamente com a origem e o destino do pacote. Após isso, o pacote é enviado para a camada de rede. Neste ponto, um número de sequência para o é criado e o seu “próximo passo” é definido. Caso o remetente não possua uma rota para o destino desejado, o pacote é armazenado em um buffer e um pacote RReq é criado.

```
def Envia_Dados(self, Pac):
```



3 – Camada de Enlace

A camada de enlace é responsável por realizar o controle de acesso ao meio. Neste projeto foi utilizada uma abordagem simples adotando uma estratégia de acesso aleatório. Quando um pacote que enviar, ele solicita o acesso à classe Controle.

```
def Processa_Envio(self,Pacote):

    #Armazena pacote no buffer
    self._BufferPac.append(Pacote)

    #Avisa para o Controle que quer acessar
    self._Controle.SolicitaAcesso(self)
```

```
def Acesso(self):

    if(self._flagAcesso == False):
        print("MAC:",self._MAC,"nao pode acessar no momento")
        return 0

    else:
        #print("MAC:",self._MAC,"acesso liberado")

        #Obtem o primeiro pacote do buffer
        Pacote = self._BufferPac[0]

        #Remove ele do buffer
        self._BufferPac.pop(0)

        #Recebe o pacte
        self.Envia_Pacote(Pacote)
```

4 – Camada Física

Dentre as três camadas esta é a mais simples. Ela é responsável por enviar os pacotes para os vizinhos. Como o projeto se trata de uma rede sem fio, cada pacote que é enviado chega em todos os nós que se encontrarem em um determinado raio de alcance.

Para determinar como os nós são distribuídos foi utilizado uma classe auxiliar chamada Grafo. Esta classe possui a matriz de adjacência gerada quando iniciamos o programa.

Quando um nó deseja enviar um pacote, ele solicita para a classe grafo quais são os nós que estão no alcance dele. A classe grafo retorna para o nó uma lista com as referências de cada nó alcançável.

```
def Envia_Pacote(self, Pacote):

    #Verifica no grafo os vizinhos alcançaveis
    listaObj = self._Grafo.Obtem_Vizinhos(self._MAC)

    #Envia o pacote para todos os vizinhos(Broadcast)
    for i in range(len(listaObj)):

        listaObj[i]._Camada_Rede.Camada_Enlace._Camada_Fisica.Recebe_Pacote(Pacote)

def Recebe_Pacote(self, Pacote):

    #Envia o pacote para a camada de enlace
    self._Camada_Enlace.Recebe_Pacote(Pacote)
```

5 – Classe Grafo

A classe Grafo possui é responsável por gerar a matriz de adjacência. Ela é utilizada para representar o grafo que representa a distribuição dos nós no espaço. O grafo também possui uma lista com os objetos da classe Nó, bem como o alcance de cada um dos nós, e um identificador para cada objeto.

A classe possui duas funções principais, sendo elas a Exibe_Grafo, na qual permite ver a distribuição da rede, bem como a função Obtem_Vizinhos, responsável por retornar a lista de vizinhos de um determinado Nó.

```
def Obtem_Vizinhos(self, MAC):

    try:
        n = self._IDs.index(MAC)

    except ValueError:
        print("O MAC nao foi encontrado na lista")
        return -1

    else:
        #Lista de referencia aos objetos vizinhos
        listaObj = []

        for i in range(len(self._IDs)):

            #Se 0 < d <= Alcance, é meu vizinho
            if(self._MatrizAdj[n][i] > 0) and (self._MatrizAdj[n][i] <= self._Alcance):

                listaObj.append(self._nos[i])

        return listaObj
```

6 – Classe Pacote

É importante ressaltar que foi na classe Pacote que foi feito a parte referente ao encapsulamento, usando uma forma explícita, onde cada camada tem acesso as funções que dizem respeito a elas mesmas.

Como a camada de rede utiliza o endeço MAC para realizar o roteamento do pacote, apenas ela pode acessar os campos de origem, destino, next step e número de sequência. O encapsulamento foi feito de forma explicita como pode ser observado abaixo:

```
def GetOrigem(self, Camada):

    tipo = str(type(Camada))

    if(tipo == "<class 'Camada_Rede.Camada_Rede'>"):
        return self._MACOrigem
    else:
        return ''

def GetDestino(self, Camada):

    tipo = str(type(Camada))

    if(tipo == "<class 'Camada_Rede.Camada_Rede'>"):
        return self._MACDestino
    else:
        return ''
```

Os dados do pacote são encapsulados para que apenas a classe Nó possas acessá-los. Além disso, apenas o nó destino posso ter acesso aos dados.

```
def GetDados(self, Camada, MAC):

    tipo = str(type(Camada))

    if(tipo == "<class 'no.no'>") and (MAC == self._MACDestino):
        return self._Dados
    else:
        return ''
```

7 – Decisões Tomadas

Neste tópico iremos abordar, de forma breve, as principais decisões tomadas na criação da lógica do código, como o porquê do protocolo escolhido para a camada de rede, qual o protocolo para acesso ao meio, entre outras decisões gerais feitas.

7.1. Protocolo Ad hoc On-Demand Distance Vector Rounting (AODV)

Este protocolo foi escolhido devido ao fato que ele foi de mais fácil entendimento, e a sua lógica foi bem compreendida pelo grupo.

7.2. Acesso ao meio

Foi feito um acesso de forma aleatória, devido ao fato de ser mais simples de ser implementado.

7.3. Outras decisões gerais

Quando um pacote chega em um nó, a primeira verificação realizada é se o número de sequência daquele pacote já está na minha lista. Isso é feito para amenizar um pouco o processamento em cada nó.

8 – Log

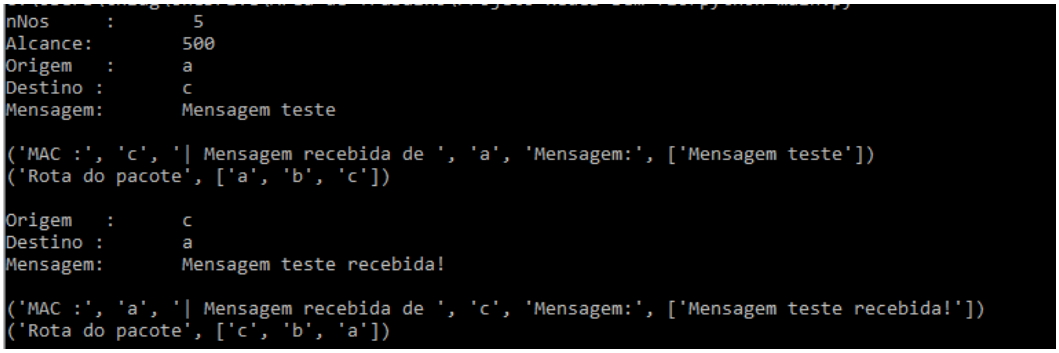
O log foi gerado utilizando as funções básicas de read e write. Cada mensagem possui o seguinte formato:

- Nó que está “falando”

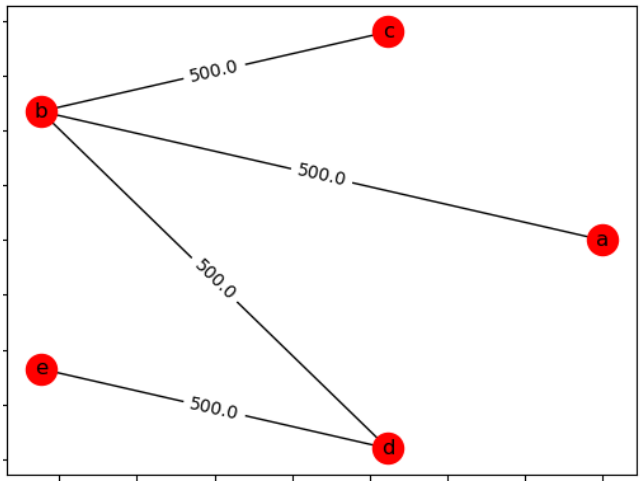
- Função que está sendo executada no momento

- Informação sobre o que está acontecendo

Abaixo podemos ver um exemplo de execução utilizando como entrada 5 nós e o alcance de 500



Abaixo podemos ver o grafo gerado



Inicialmente, cada nó não possui nenhum vizinho

```
MAC: a []

MAC: b []

MAC: c []

MAC: d []

MAC: e []
```

Após isso, cada nó envia um pacote Hello para descobrir os vizinhos

```
MAC: a| (Envia_Hello) Enviando pacote hello

MAC: b| (Envia_Hello) Enviando pacote hello

MAC: c| (Envia_Hello) Enviando pacote hello

MAC: d| (Envia_Hello) Enviando pacote hello

MAC: e| (Envia_Hello) Enviando pacote hello
```

Abaixo podemos ver cada nó recebendo o pacote Hello dos seus demais vizinhos.

```
T4:
MAC: b| (Recebe_Hello) Pacote hello recebido
MAC: b| (Recebe_Hello) Adicionando nova rota para
T5:
MAC: b| (Recebe_Hello) Pacote hello recebido
MAC: b| (Recebe_Hello) Adicionando nova rota
MAC: e| (Recebe_Hello) Pacote hello recebido
MAC: e| (Recebe_Hello) Adicionando nova rota
```

Após isso, os nós descobrem seus vizinhos. Abaixo podemos ver as rotas de cada Nó após o Hello

```
MAC: a [['b', 'b', 1]]
MAC: b [['a', 'a', 1], ['d', 'd', 1], ['c', 'c', 1]]
MAC: c [['b', 'b', 1]]
MAC: d [['e', 'e', 1], ['b', 'b', 1]]
MAC: e [['d', 'd', 1]]
```

Como o nó A não conhece rota para C, ele envia um RReq para descobrir uma rota

```
MAC: a| (Envia_Dados) Preparando pacote de dados. Destino = c
MAC: a| (Envia_Dados) Nenhuma rota encontrada para c. Enviando Route Request!
MAC: a| (Envia_Request) Solicitando rota para c. Enviando RRequest
MAC: a| (Envia_Dados) Preparando pacote de dados. Destino = c
MAC: a| (Envia_Dados) Nenhuma rota encontrada para c. Enviando Route Request!
MAC: a| (Envia_Request) Solicitando rota para c. Enviando RRequest
```

Quando o pacote Req chega no Nó C ele envia o RRep para A

```
MAC: c| (Recebe_Request) Pacote de request recebido. Origem = a

MAC: c| (Recebe_Request) Enviando Reply

MAC: c| (Envia_Reply) Enviando Reply para a
```

Após isso, a nova rota é aprendida e o pacote é enviado

```
MAC: a [['b', 'b', 1], ['c', 'b', 2]]

MAC: b [['a', 'a', 1], ['d', 'd', 1], ['c', 'c', 1]]

MAC: c [['b', 'b', 1], ['a', 'b', 2]]

MAC: d [['e', 'e', 1], ['b', 'b', 1]]

MAC: e [['d', 'd', 1]]
```