Universidade Federal da Paraíba
Centro de Informática

Redes sem Fio

Caio Victor do Amaral Cunha Sarmento - 20170021332

Claudio de Souza Brito - 20170023696

Gabriel Teixeira Patrício - 20170170889

João Pessoa, 2021.

Universidade Federal da Paraíba
Centro de Informática

Projeto de Redes sem Fio

Relatório final sobre a implementação das camadas físicas, de enlace e de rede de uma rede sem fio. Tal projeto foi desenvolvido durante a realização da disciplina de Redes sem Fio, ministrada pelo professor Fernando Menezes Matos, do Centro de Informática da Universidade Federal da Paraíba.

João Pessoa, 2021.

Sumário

Objetivo Geral

Neste projeto a equipe deverá implementar as camadas físicas, de enlace e de rede de uma rede sem fio. O objetivo foi criar uma simulação de rede sem fio onde pacotes deverão sair de uma origem e chegar a um destino, não importando o número de nós intermediários.

Introdução

Inicialmente, na simulação devem existir as entidades roteador, pacote e hospedeiro (host), todas podendo ter quantidades configuráveis, além da possibilidade de existirem outras entidades conforme a necessidade. Somado a isso, deve-se implementar algum protocolo de roteamento de redes sem fio para redes ad hoc (redes de sensores, VANETs, MANETs, FANETs, redes mesh, etc), implementar o controle de acesso ao meio (camada de enlace) e uma forma de transmissão de um pacote entre entidades (função da camada física).

Também é necessário implementar o conceito de encapsulamento do pacote entre as camadas e implementar alguma forma de disposição dos nós em um ambiente (aleatória, matriz, etc). O caminho para cada pacote será definido conforme o protocolo de roteamento escolhido, mostrando por algum método as rotas do encaminhamento dos pacotes. Por fim, a simulação deve gerar log com o resultado do encaminhamento para análise.

O relatório tem como objetivo explicar de forma clara o projeto feito na disciplina de Redes Sem Fio, sendo organizado em tópicos que mostrarão como foi desenvolvido, os resultados obtidos, e a conclusão chegada. A seguir teremos os tópicos de Metodologia, Desenvolvimento, Análise e Resultados, o Link do repositório onde se encontra o código e a Conclusão.

Metodologia

O software foi desenvolvido na linguagem de programação Python em sua versão 3.8 para simular toda a rede de acordo com os objetivos especificados acima no ambiente de desenvolvimento Visual Studio Code. Para conjuntos específicos de funções, como plotar o gráfico dos hospedeiros, foram necessárias bibliotecas adicionais, como *matplotlib*, *numpy*, *uuid*, *random* e *json*. Além disso, ferramentas externas foram utilizadas no decorrer do desenvolvimento para uso da equipe, como *GitHub* para armazenamento, compartilhamento e controle de versão do código implementado e o *Discord* para reuniões e realização da programação em si.

Protocolos:

Na camada de Enlace, o protocolo escolhido foi o **CSMA/CA**, pois ele evita colisões, mesmo não corrigindo o problema do terminal oculto e exposto, o que é o suficiente para o programa. Sendo assim, sempre antes de mandar algo, o host irá "escutar" o canal, o que é representado no nosso programa por "algum vizinho meu tá transmitindo algo?", caso algum vizinho estiver transmitindo, então o canal é dito como bloqueado e o host tenta novamente no próximo round, evitando, assim, eventuais colisões. Caso esteja liberado, ele pode mandar o pacote e aguarda a confirmação de recebimento do *ack*. Se um host está esperando o *ack*, ele fica "congelado", inclusive ignorando todos os outros pacotes que chegam nele, e espera por até 2 rounds. Caso ele não receba a confirmação, manda o mesmo pacote de novo, ou seja, mesmo se houver colisão e perda de pacote, o host que não recebeu *ack* garante que o pacote não vai ser perdido, enviando-o novamente, evitando e corrigindo a possibilidade de erros e perda de pacote.

Já na camada de Redes, foi usado o protocolo **DSR**, que é responsável por enviar a requisição (RREQ) a partir do hospedeiro de origem via *broadcast* para todos os hospedeiros vizinhos na rede, com o objetivo de encontrar uma rota para o destino, que por sua vez, deverá retornar uma resposta (RREP), também em *broadcast*, informando que está pronto para receber o pacote. Sendo assim, se o host de origem receber o RREP, ele envia o pacote, caso contrário, ele congela até receber um RREP, mas mesmo congelado, repassa um RREQ/RREP vindo de outro host. É uma maneira eficiente de adquirir rotas, e é simples de implementar.

Desenvolvimento

A seguir, será explicado o fluxo de funcionamento do projeto implementado, separado por camadas e funções, no qual cada camada é uma classe no código do programa. O fluxo do programa é dado por *Rounds*, que foi a técnica escolhida para representar o tempo para cada tarefa ser executada. Cada *round* possui três fases: a primeira é quando os hosts recebem os pacotes enviados no *round* anterior, na segunda é possível visualizar os pacotes em cada host e na terceira, são mostradas as ações executadas por cada host. Para ilustrar, a figura abaixo explica de forma visual a trajetória do pacote pelas funções e em seguida será explicado como cada função funciona.

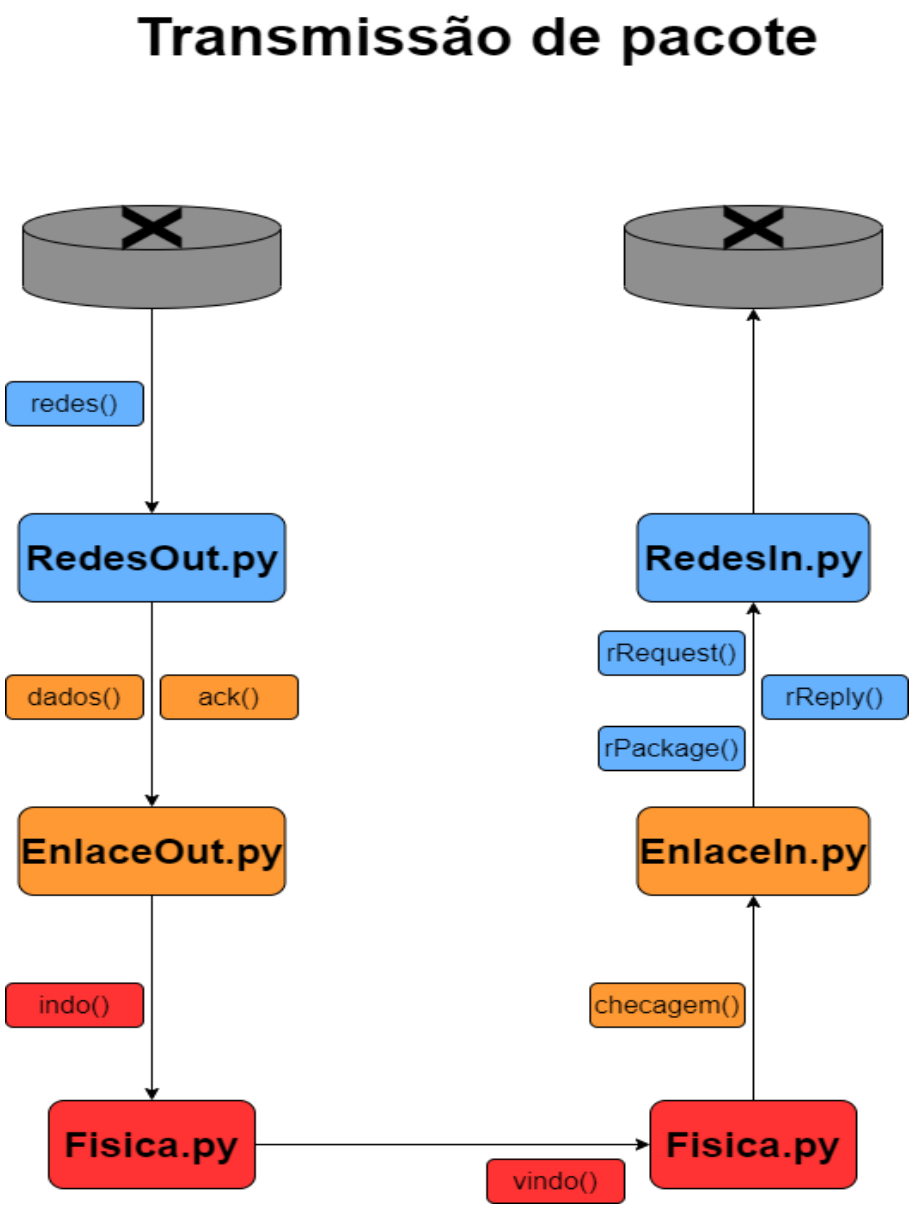

Figura 1: Esquema do fluxo de transmissão do pacote no nosso programa.

Estrutura do programa:

- 📂**Entradas:**

  ┗ **hospedeiros.txt**

  Arquivo que contém a quantidade, posição e raio de alcance de cada hospedeiro, que é organizado de acordo com o mostrado abaixo:

  [dimensão_inicial] [dimensão_final]
  [numero_de_hosts]
  {“x”: [host[i].x], “y”: [host[i].y], “range”: [host[i].range]}
  …

  Exemplo:

  0 150
  4
  {"x": 10, "y": 20, "range": 30}
  {"x": 10, "y": 58, "range": 30}
  {"x": 15, "y": 70, "range": 30}
  {"x": 30, "y": 41, "range": 30}

  ┗ **pacotes.txt**

  Arquivo que contém a origem, destino e a mensagem de cada pacote, que é organizado de acordo com o mostrado abaixo:

  {"origin":[host_origem], "destino":[host_destino], "message":"<mensagem>"}
  …

  Exemplo:

  {"origin":1, "destino":2, "message":"<mensagem>"}
  {"origin":0, "destino":3, "message":"<mensagem>"}
  {"origin":0, "destino":1, "message":"<mensagem>"}
  {"origin":2, "destino":0, "message":"<mensagem>"}

- **Graph.py**

Arquivo que contém a função de criar e plotar o gráfico de disposição dos hospedeiros, juntamente com seus alcances, que é salva como "hospedeiros no plano.png" na pasta "resultados", de acordo com o gráfico mostrado abaixo:

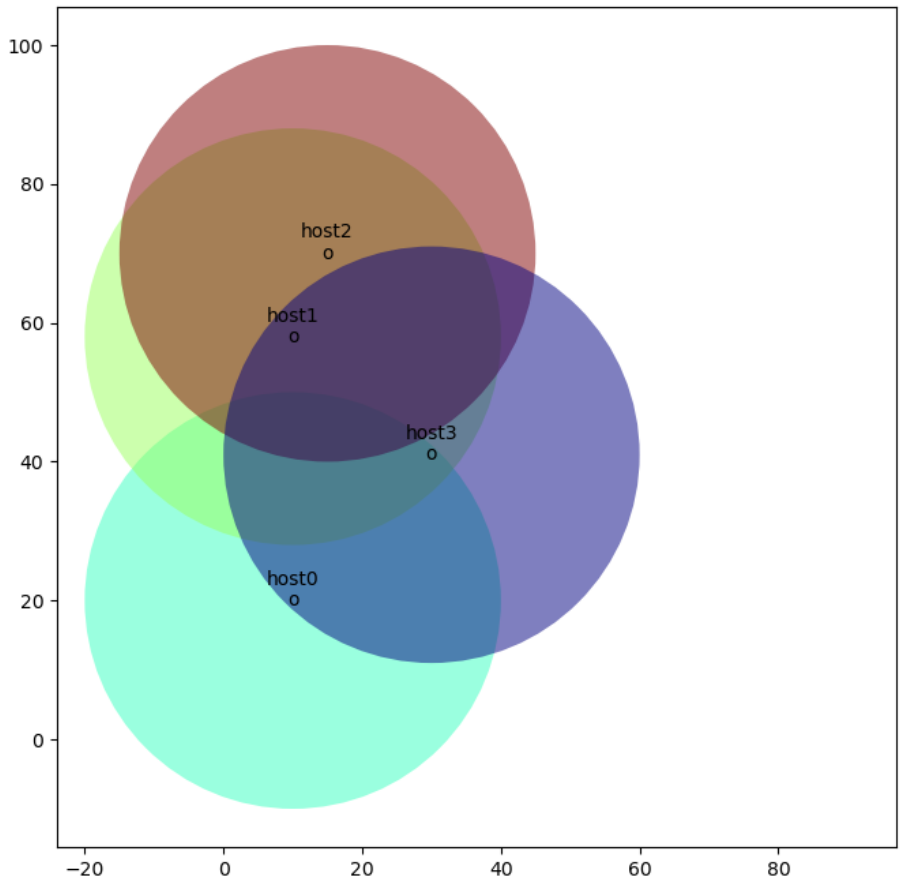

Figura 2: Gráfico de hospedeiros que foi utilizado para um dos exemplos de testes.

- **RedeOut.py**

O arquivo redeOut.py é responsável pelas funções Distance(), Distance_Matrix(), Broadcast() e redes(), que serão explicadas a seguir:

➢ Distance (*origem, destino*):

A função é responsável por calcular a distância entre um host de origem e um de destino passados como parâmetro.

➢ Distance_Matrix (lista*Host*):

A função utiliza a função Distance() para construir uma matriz de distâncias entre todos os hospedeiros.

➢ Broadcast (*REQ, origem, destino, listaPacotes, listaHost*):

A função recebe os parâmetros acima para enviar os *broadcasts* tanto de requisições como o envio em si dos pacotes.

➢ Redes (*origem, listaPacotes*):

A função só é chamada caso o canal não esteja bloqueado e o host tem mensagem para enviar, e é a função que decide a ação que o host vai tomar, de acordo com o esquema abaixo:

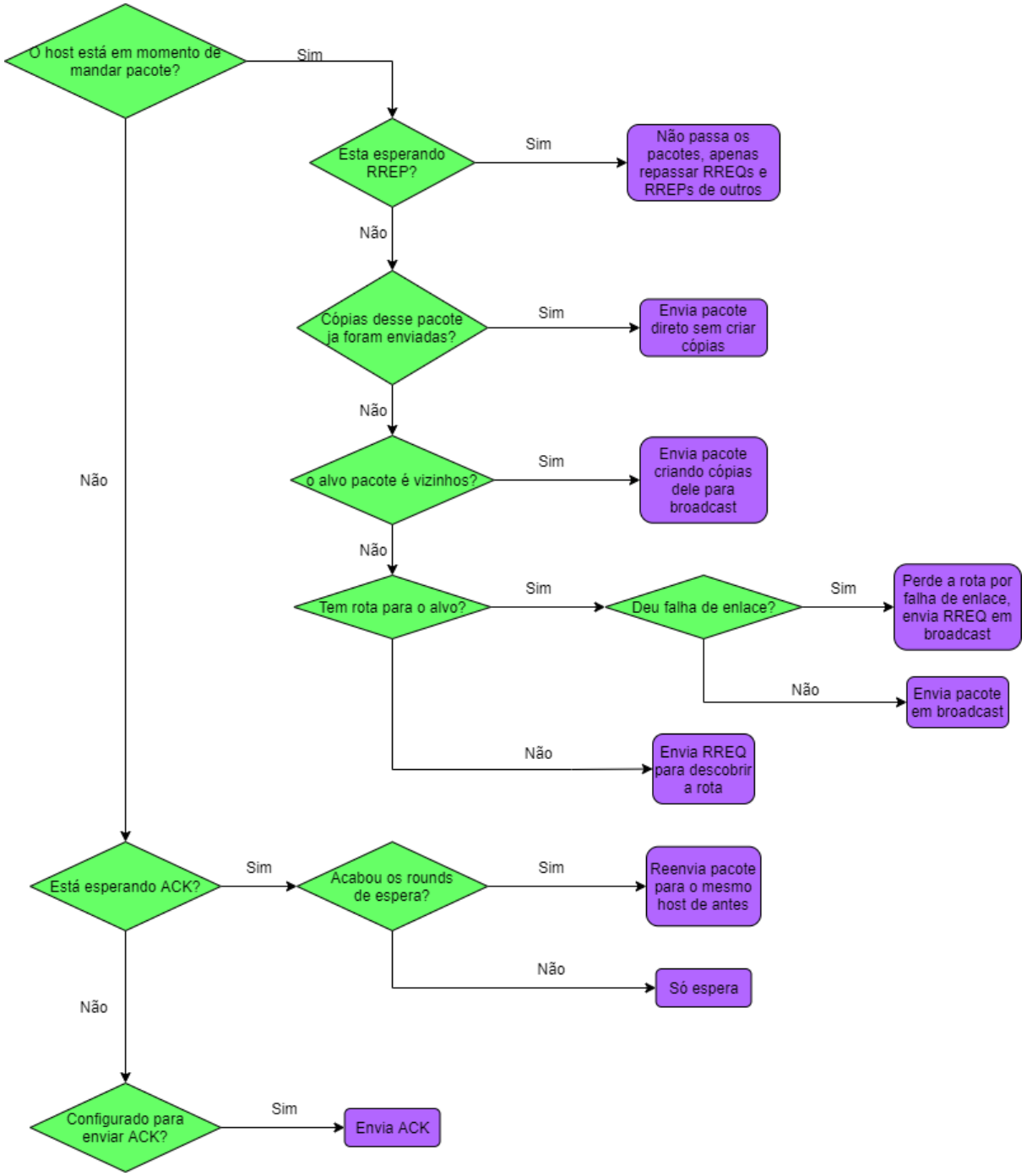

Figura 3: Tomada de decisão da função redes()

- **EnlaceOut.py**

O arquivo enlaceOut.py é responsável pelas funções dados() e ack(), que serão explicadas a seguir:

➢ dados (*hostOrigin, pacote, idDestino*):
Função que envia os pacotes de acordo com o host de origem e o de destino.

➢ ack (*hostOrigin*):
Função que envia os *acks* de acordo com o host de origem.

- **Fisica.py**

O arquivo fisica.py é responsável por fazer o caminho dos pacotes entre nós, chamando as funções indo() e vindo():

➢ indo(*pacote, hostAlvoId*):
A função vai fazer os pacotes percorrerem os nós em direção ao destino.

➢ vindo(*pacote, hostChegada, listaHost, listaPacotes*):
A função vai fazer os pacotes percorrerem os nós em direção do destino a origem.

- **EnlaceIn.py**

O arquivo enlaceIn.py é responsável pela função Checagem():

➢ Checagem(*pacote*, *hostChegada*, listaHost, *listaPacotes*):
Função responsável por verificar o tipo de pacote que chegou ao host de chegada.

- **RedeIn.py**

O arquivo redeIn.py é responsável pelas funções rRequest(), rReply() e rPackage():

➢ rRequest(*pacote*, *hostChegada*, *listaPacotes*):
Função responsável por receber os RREQ. Armazena rotas no host se necessário, e prepara para encaminhá-lo se necessário

- ➢ rReply(*pacote*, *hostChegada*, *listaPacotes*):
  Função responsável por receber os RREP. Armazena rotas no host se necessário, e prepara para encaminhá-lo se necessário

- ➢ rPackage(*pacote*, *hostChegada*, *listaHost*):
  Função responsável por receber os pacotes que não sejam RREQ, RREP ou *ACK*. Decide se eles serão encaminhados, ou se serão dados como "resolvidos".

- 📂 **Resultados**

  ╚ **descricao.log**

  Arquivo que mostra os logs (resultados) do programa, que será mostrado na seção abaixo.

  ╚ **hospedeiros no plano.png**

  Imagem gerada do gráfico de disposição dos hospedeiros no **graph.py.**

Nota sobre a falha de enlace:

Foi feita para simular uma perda de rota, algo que acontece na vida real por causa da mobilidade ou energia em nós. Como alternativa de implementação, usamos uma probabilidade de 20% disso ocorrer.

Análise e Resultados

Com os testes feitos mudando tanto a informação do pacote, o nó de origem e o nó de destino, ainda foi feito testes com a quantidade de nós, a posição de cada nó e seus alcances. Após algumas versões, foi possível satisfazer todos os requisitos de funcionamento, possibilitando um funcionamento exemplar do código, para detalhar mais o resultado é possível ser visto no arquivo de logs.

Nas figuras abaixo foi utilizado algumas nomenclaturas de encapsulamento, e para melhor entendimento da figura, antes será mostrado o significado de algumas nomenclaturas que estão presente nas figuras:

- **RREQ:** (*host que criou o rreq*)**-**(*host alvo do rreq*)**:**(*rota atual do rreq até agora separado por vírgulas*).
  Ex.: RREQ:0-2:0,3,1.
  Tradução: RREQ de host 0 a host 2 com rota armazenada de 0-3-1.

- **RREP:** (*host que criou o rrep*)**-**(*host alvo do rrep*)**:**(*rota atual do rrep até agora separado por vírgulas*).
  Ex.: RREP:0-2:0,3,1,2.
  Tradução: RREP de host 2 ao 0 com rota armazenada de 0-3-1-2.
  Nota: A rota armazenada é a mesma do RREQ pois importa para o que criou o RREQ.

- **Pacotes:** (*host origem*)**-**(*host destino*)/(*mensagem*).
  Ex.: 2-1/algo.
  Tradução: Mensagem “algo” enviada do host 2 para o host 1.
  Nota: Esse encapsulamento é feito para manter as informações básicas durante o processo.

```
 Round 4
     Recebimentos:
(FISICA)         Pacote 1-2/<mensagem> enviado por host 1 chegou no host 3
(ENLACE)             Host entrando em modo de ACK
(ENLACE)             Incapaz de receber qualquer outro pacote esse round
(REDES)                      Pacote 1-2/<mensagem> sera ignorado pelo host pois nao faz parte da rota
     Filas de pacotes nos hosts:
         Host 0 possui os seguintes pacotes para enviar:
             RREQ:0-1:0
             0-1/<mensagem>
         Host 1 possui os seguintes pacotes para enviar:
             1-2/<mensagem>
             1-2/<mensagem>
         Host 2 possui os seguintes pacotes para enviar:
             RREQ:2-0:2
             2-0/<mensagem>
         Host 3 possui os seguintes pacotes para enviar:
             RREQ:0-1:0,3


     Movimentos:
(REDES)          Host 0 esperando ack
(REDES)          Host 1 esperando ack
(REDES)          Host 2 esperou ack e nao recebeu, planeja reenviar pacote para host 1, pois eh vizinho
(ENLACE)                 Prepara para enviar pacote: RREQ:2-0:2 para host 1
(FISICA)                     Enviando pacote RREQ:2-0:2 para host 1
(ENLACE)                 Entra em modo de espera ack
(ENLACE)                 Bloqueia o canal para seus vizinhos
(REDES)          Host 3 planeja enviar confirmacao de pacote ACK de volta para host 1
(ENLACE)                 Prepara para enviar pacote ack
(FISICA)                     Enviando pacote ACK para host 1
(ENLACE)                 Bloqueia o canal para seus vizinhos
```

Figura 4: Print de parte do arquivo de Log no Visual Code.

Comentários:

O pacote 1-2/<*mensagem*> chega no host 3 pois a transmissão funciona por broadcast, mas como ele não faz parte da rota do pacote, o host ignora, por isso não aparece na fila de pacotes do host 3. Podemos perceber que os hosts 0 e 1 não realizam nenhuma ação pois esperam a confirmação ACK. Host 2 também estava aguardando, mas o tempo de espera acabou e o host voltou a mandar o pacote. Host 3 envia o ACK que o host 1 estava aguardando.

Nota: O nome da camada no início de cada linha do Log indica em que camada a ação ocorreu.

```
 Round 5
     Recebimentos:
(FISICA)         Pacote RREQ:2-0:2 enviado por host 2 chegou no host 1
(ENLACE)                 Mas foi bloqueado pois o mesmo estava esperando um ACK
(FISICA)         Pacote ACK enviado por host 3 chegou no host 1
(ENLACE)             Ack recebido, eliminando pacote do topo da lista
     Filas de pacotes nos hosts:
         Host 0 possui os seguintes pacotes para enviar:
             RREQ:0-1:0
             0-1/<mensagem>
         Host 1 possui os seguintes pacotes para enviar:
             1-2/<mensagem>
         Host 2 possui os seguintes pacotes para enviar:
             RREQ:2-0:2
             2-0/<mensagem>
         Host 3 possui os seguintes pacotes para enviar:
             RREQ:0-1:0,3


     Movimentos:
(REDES)          Host 0 esperou ack e nao recebeu, planeja reenviar pacote para host 3, pois eh vizinho
(ENLACE)                 Prepara para enviar pacote: RREQ:0-1:0 para host 3
(FISICA)                     Enviando pacote RREQ:0-1:0 para host 3
(ENLACE)                 Entra em modo de espera ack
(ENLACE)                 Bloqueia o canal para seus vizinhos
(REDES)          Host 1 proximo pacote ja foi disparado na rede, nao precisa clonar de novo
(ENLACE)                 Prepara para enviar pacote: 1-2/<mensagem> para host 2
(FISICA)                     Enviando pacote 1-2/<mensagem> para host 2
(ENLACE)                 Entra em modo de espera ack
(ENLACE)                 Bloqueia o canal para seus vizinhos
                 Host 2 esta esperando ack, canal bloqueado
                 Host 3 quer enviar RREQ:0-1:0,3 com destino a 1 mas canal bloqueado
```

Figura 5: Print 2 de parte do arquivo de Log no Visual Code.

Comentários:

Pode-se perceber que quando o pacote RREQ:2-0:2 chega no host 1, ele vai para o topo da fila, isso acontece pois os pacotes RREQ e RREP são prioridade, diferente dos outros pacotes que vão para o fim da fila. Além disso, o pacote 1-2/<*mensagem*> que estava no host 1 desaparece, isso acontece porque ele recebeu a confirmação ACK que a transmissão foi um sucesso, então ele pode apagar a memória daquele pacote. Host 0 reenvia o pacote depois de esperar 2 *rounds* pelo ACK. Host 1 envia um pacote que já foi "clonado", como a transmissão é broadcast, os pacotes são clonados para serem enviados a todos os vizinhos. No round passado ele enviou ao vizinho host 3, agora ele envia para o 2. Host 2 espera ACK. Host 3 quer enviar um pacote, mas está incapacitado pois o canal foi bloqueado pelos seus vizinhos 0 e 1, evitando colisão.

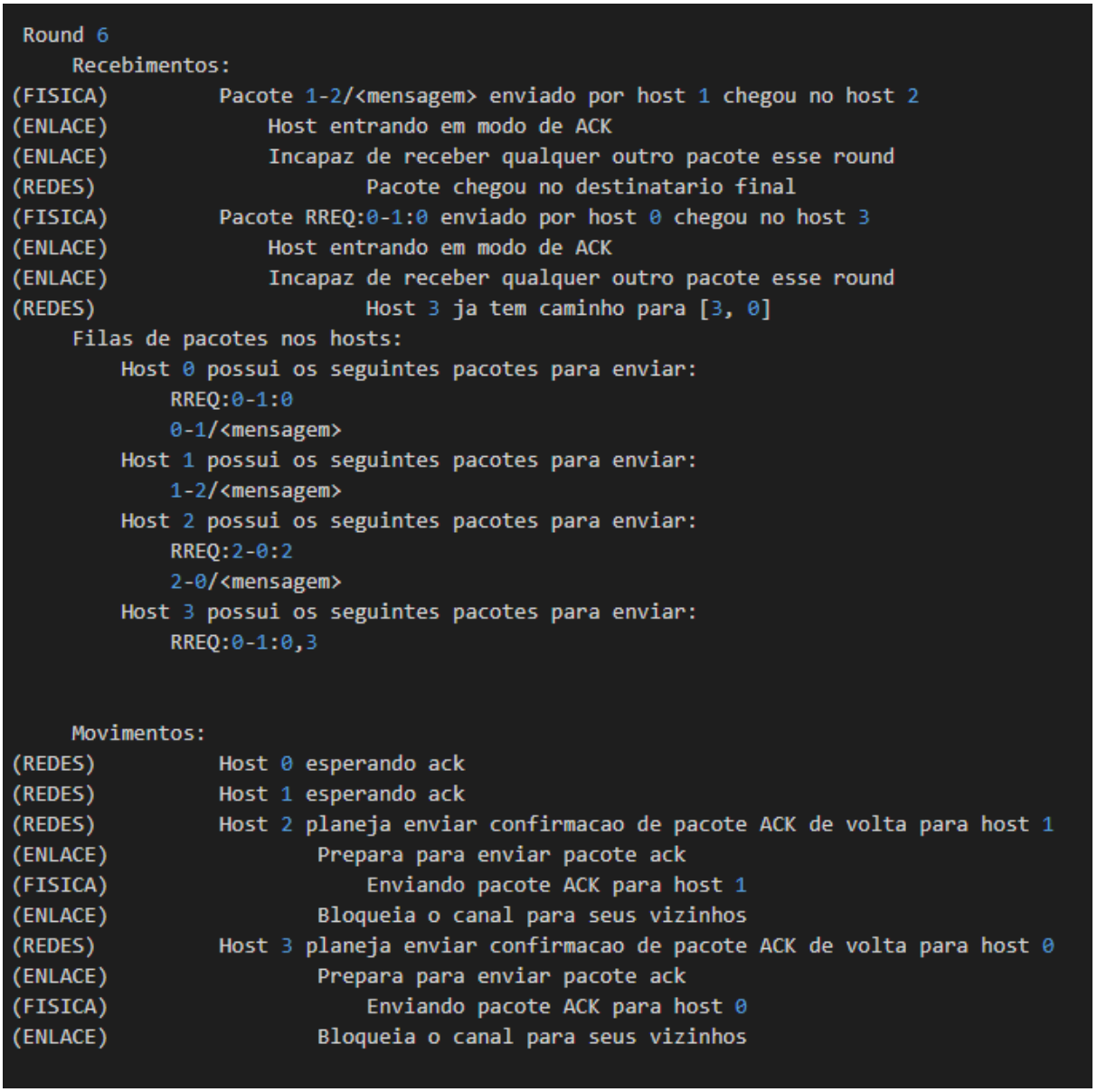

Figura 6: Print 3 de parte do arquivo de Log no Visual Code.

Comentários:

Pode-se perceber que o pacote 1-2/<*mensagem*> chega no host 2, e logo identifica que é o destino final do pacote, de acordo com o cabeçalho. Pacote RREQ:0-1:0 chega no host 3, o qual já tinha esse RREQ antes, e isso acontece pois é a segunda vez que o host 0 envia o pacote. Ele faz isso pois não tinha recebido a confirmação, porém, o host 3 sabe que já recebeu aquilo, e apenas ignora. Hosts 0 e 1 esperam seus ACKs, e host 2 e 1 enviam para eles, podendo ser observado que não ocorreu bloqueio de canal, pois eles não são vizinhos.

Link do Repositório

- https://github.com/claudiosouzabrito/Redes-sem-fio

Conclusão

O projeto foi desenvolvido com êxito e respeitando todas as requisições necessárias, proporcionando não apenas um conhecimento teórico do assunto, mas também de forma prática como o envio de pacotes por redes sem fio tem toda uma complexidade de implementação. Por se tratar de uma simulação, foi necessário replicar algumas funcionalidades físicas com ferramentas de programação, mas acreditamos ter atingido de forma satisfatória o objetivo do projeto.