FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Engenharia Electrotécnica

REDES INTEGRADAS DE TELECOMUNICAÇÕES I

2015 / 2016

Mestrado Integrado em Engenharia Electrotécnica
e Computadores

4º ano
7º semestre

1º Trabalho Prático:
Encaminhamento dinâmico numa rede em malha

http://tele1.dee.fct.unl.pt

Luis Bernardo

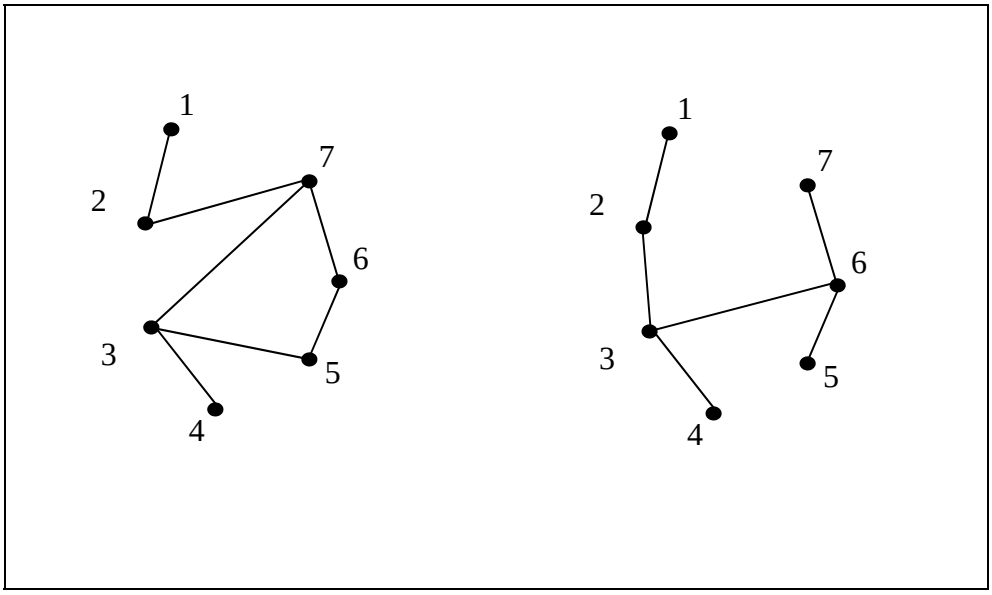
1. Objectivos

Familiarização com o funcionamento de encaminhadores (*routers*) numa rede em malha. O trabalho consiste na comunicação entre componentes interligados numa rede em malha. Neste trabalho imagina-se que as estações estão a usar uma rede em malha por cima da rede física Ethernet. O trabalho consiste no desenvolvimento de um programa ***router*** que pode comunicar com os *routers* vizinhos, oferecendo um serviço de encaminhamento baseado num **algoritmo de estado de linha**, resultante da simplificação do algoritmo OSPFv2 (*Open Shortest Path First*)*, usado nas redes TCP/IP.

Em sistemas reais, os encaminhadores interligam máquinas pertencentes a várias redes locais. Neste trabalho vai-se simular este ambiente, com algumas simplificações: Os *routers* são também, simultaneamente, os emissores e receptores das mensagens. Ao gerar pacotes a partir dos vários *routers* para todos os outros *routers*, o trabalho vai permitir testar o comportamento do serviço de encaminhamento perante modificações na rede.

2. Especificações

Cada processo *router* na rede é identificado por um nome: A, B, C, D, etc. Embora se esteja a usar uma rede Ethernet no Laboratório, de um ponto de vista lógico as máquinas não podem comunicar directamente umas com as outras, mas têm um circuito para o fazer. A topologia de cada rede em malha é definida pelo conjunto de relações de vizinhança introduzidas localmente na interface gráfica de cada *router*. Cada *router* tem apenas conhecimento dos vizinhos directos, recebendo informações acerca do resto da rede apenas através do algoritmo de encaminhamento usado – estado de linha. As figuras em baixo mostram duas hipóteses de redes usando 7 routers.



Os vários processos ***router*** de cada máquina comunicam através de *sockets* datagrama.

Um *router* real monitoriza os vizinhos na rede e troca com eles pacotes de controlo (com as tabelas de encaminhamento) periodicamente, ou em resposta a acontecimentos imprevistos. Nesta simulação, os processos *routers* simulam este comportamento através de comandos na interface de utilizador.

* Definido no IETF RFC 2328, disponível em http://www.ietf.org/rfc.html

Em termos de comportamento dos processos *routers*, existe o seguinte cenário:

Um *router* oferece para o utilizador opções de monitorização dos vizinhos:

- permite acrescentar vizinhos (identificados pelo endereço IP e porto do seu *socket*);
- permite mudar a distância a qualquer dos vizinhos;
- permite desligar uma ligação para um vizinho.

Paralelamente, o processo *router* participa no algoritmo de encaminhamento da classe estado de linha. Neste trabalho é usada uma versão simplificada do OSPF, apenas com uma área e com a optimização de difusão dos pacotes com o estado da ligação. Periodicamente o processo *router* desencadeia o envio de pacotes de actualização do estado das ligações locais, e recalcula a tabela de encaminhamento local. Esta sequência de acções também pode ser desencadeada por uma modificação no estado da rede. Os pacotes com o estado da ligação são enviados utilizando-se o algoritmo de inundação especificado no OSPF, até serem distribuídos por todos os encaminhadores. Simplificou-se o algoritmo de inundação ao remover o pacote de confirmação (ACK) e o atraso na retransmissão do pacote, admitindo que é sempre correctamente recebido. Para facilitar a visualização do estado do *router*, apresenta-se o conteúdo da tabela de encaminhamento local na interface gráfica.

Finalmente, o processo *router* oferece na sua interface opções de envio e recepção de pacotes de dados:

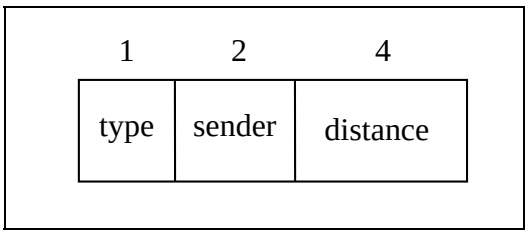
- permite enviar pacotes de dados para qualquer destino;
- recebe e reenvia os pacotes de dados para um *router* vizinho, consoante o conteúdo da tabela de encaminhamento, memorizando no pacote o percurso;
- recebe os pacotes de dados no *router* de destino, apresentando a rota completa percorrida pelo pacote desde a origem até ao destino.

Notar que neste simulador, tal como num sistema real, a topologia da rede pode ser diferente para cada sentido. Por exemplo, numa linha de acesso a uma rede com um servidor *web* muito acedido, o tráfego será mais elevado no sentido originado a partir do servidor web.

2.1. Configuração da rede

Quando o programa *router* arranca, não tem nenhum vizinho. Depois, vai ganhar ou perder vizinhos, à medida que o utilizador usa as operações de acrescentar, remover ou modificar distâncias, ou à medida que outros *routers* se associam a ele. É usado um protocolo para criar e destruir relações de vizinhança entre *routers*. Quando um utilizado acrescenta um vizinho à lista, o programa envia um pacote HELLO para o endereço IP e porto do vizinho. Quando um programa *router* recebe um pacote HELLO acrescenta o emissor do pacote à lista de vizinhos, respondendo também com um pacote HELLO, com igual valor de distância. Admite-se que por omissão a distância é igual nos dois sentidos, embora o utilizador possa modificar em qualquer altura a distância em cada sentido. Caso o vizinho não aceite o pacote HELLO por exceder o número máximo de vizinhos, deverá terminar a relação de vizinhança enviando um pacote BYE (ver adiante).

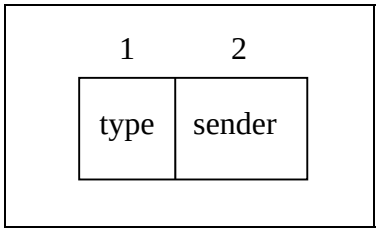
```
Pacote HELLO: { sequência contígua de }
 byte type;   { tipo pacote  -  HELLO = 21 }
 char sender;       { nome router origem }
 int        distance;     { distância ao vizinho }
```



O pacote HELLO também é usado para comunicar a um vizinho que mudou a distância entre dois *routers*. A distância é um valor inteiro compreendido entre 0 e 50. **A distância 50 é reservada e simboliza a não existência de ligação entre dois *routers***.

Quando um *router* quer terminar uma relação de vizinhança envia um pacote BYE para o vizinho. Tanto o emissor como o receptor da mensagem devem terminar imediatamente o envio de pacotes de anúncio de rotas para o ex-vizinho.

```
Pacote BYE: { sequência contígua de }
 byte type;   { tipo pacote  -  BYE = 22 }
 char sender;       { nome router origem }
```



Durante o trabalho, pretende-se testar o comportamento dos algoritmos de estado de linha com o surgir de novos *routers* e com o terminar de ligações ou *routers*. Assim, durante o desenvolvimento do trabalho deverá testar o comportamento do programa desenvolvido perante estes cenários.

2.2. Algoritmo de inundação de pacotes de estado de linha

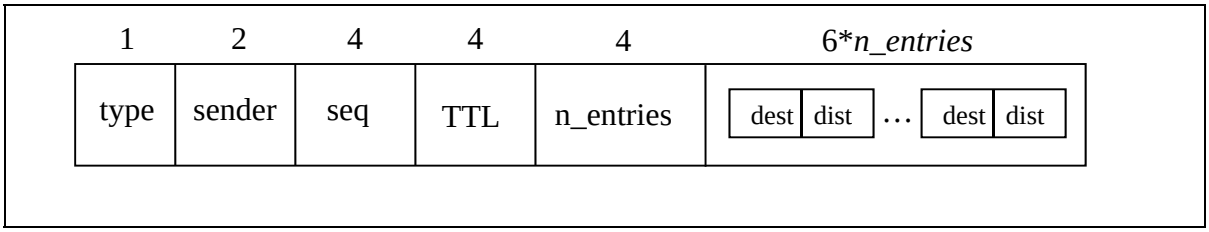
A partir do momento em que fica activo, o programa *router* deve enviar periodicamente para todos os *routers* na rede um pacote ROUTE, com as informações referentes às ligações aos seus vizinhos. Se a opção "*Send if changes*" na janela do programa estiver seleccionada, deverá também enviar após uma mudança nos dados da vizinhança, desde que o envio anterior seja menor do que um intervalo mínimo configurável.

O *router* participa no algoritmo de inundação dos pacotes ROUTE de todos os *routers* para todos os *routers*. Cada *router* mantém uma tabela com o número de sequência do mais recente pacote ROUTE recebido de cada origem. Caso o pacote ROUTE recebido seja mais recente (tenha um número de sequência superior), reenvia-o para todas as ligações, excepto aquela a partir de onde foi recebido. O TTL deve ser decrementado em uma unidade. Caso o número de sequência seja idêntico ou inferior, deve descartar o pacote, evitando uma retransmissão desnecessária do pacote. O pacote ROUTE deve ser guardado até um tempo máximo igual ao período de vida do pacote (TTL) **igual ao período de envio de pacotes ROUTE mais dez segundos/hops**.

Pacote ROUTE: { sequência contígua de }

```
 byte type;   { tipo pacote  -  ROUTE = 10 }
 char sender;       { nome router origem }
 int        seq;          { número de sequência }
 int        TTL;          { tempo de vida da informação [s] }
 int        n_entries;    { número de entradas na tabela }
 Entry[]    entries;      { array com n_entries entradas }

class Entry { // Tipo de entrada no array
 char       dest;         { nome de router vizinho }
 int        dist;         { distância até router }
}
```



Para simplificar o programa, deverá admitir que as únicas modificações à topologia da rede (aos vizinhos) ocorrem em resultado de modificações na vizinhança controlada pelo utilizador, ou em resultado da recepção de pacotes HELLO e BYE. Admita ainda que não há perdas na rede local, não sendo portanto recorrer a pacotes ACK no processo de inundação do estado da rede.

2.3. Algoritmo de encaminhamento

A tarefa mais importante do programa *router* é o cálculo da tabela de encaminhamento utilizando o algoritmo de estado de linha. Após cada envio periódico do pacote ROUTE (ou após uma modificação do estado da rede com a opção "*Send if changes*") deverá recalcular a tabela local de encaminhamento aplicando o algoritmo de Dijkstra à informação recebida de todos os *routers*. Este algoritmo deve ser utilizado um número reduzido de vezes devido à sua complexidade algorítmica de ordem $O(n^2)$. Assim, deve ser utilizado para calcular a tabela de encaminhamento, que por sua vez, é usada no encaminhamento dos pacotes de dados.

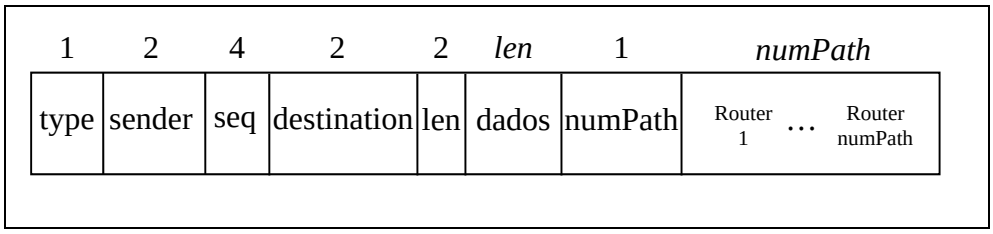
2.3.1 Partes opcionais do algoritmo de encaminhamento e inundação

A) Deixe para o fim a realização da opção "*Send if changes*", para ter respostas rápidas a modificações na topologia. Caso se detecte uma modificação local, ou remota (através do conteúdo dos pacotes ROUTE), deve ser recalculada a tabela de encaminhamento.

2.4. Envio e recepção de dados

Os programas *router* também simulam o papel de emissor e receptor de pacotes de dados. Sempre que o utilizador selecciona a opção de enviar dados, o programa deve criar um pacote do tipo DATA, com o valor de *path* igual ao nome local. Em seguida, o programa deve consultar a tabela de encaminhamento e deve enviar o pacote para o próximo *router* indicado na tabela, ou retornar o código de erro caso o nome de destino esteja inacessível.

```
Pacote DATA: { sequência contígua de }
 byte type;   { tipo pacote  -  DATA = 109 }
 char sender;       { nome router origem }
 int        seq;          { número de sequência }
 char destination;  { nome router destino }
 shortlen;          { número de bytes de dados }
 byte[]     dados[];      { len bytes de dados }
 byte numPath;      { número de routers percorridos }
 byte[]     path;         { sequência de routers percorridos }
```



Cada *router* que receber o pacote deverá acrescentar o seu nome ao campo *path* do pacote, incrementando o valor de *numPath*. Caso seja o *router* de destino de pacote, deve escrever o conteúdo do pacote recebido. Senão, deve consultar a sua tabela de encaminhamento e enviar para o próximo *router*. Caso o tamanho do percurso percorrido pelo pacote atinja a distância máxima (8), então deverá aparecer uma mensagem de erro a sinalizar o facto ao utilizador, e o pacote deve ser tratado como se tivesse atingido o destino. Desta forma, fica-se a saber sempre por onde ela viajou.

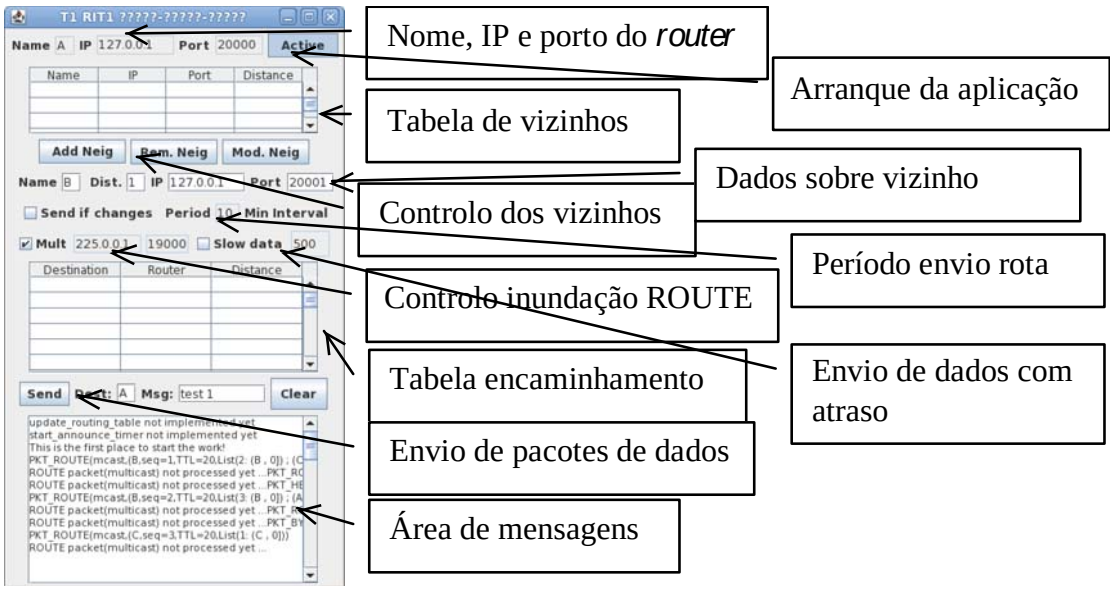
2.4.1 Partes opcionais do envio e recepção de dados

A) Deixe para o fim a realização da opção "Slow data", onde o objectivo é reter o envio dos dados durante o número de milisegundos indicado na caixa, antes de retransmitir o pacote para o próximo nó. Desta forma, é possível que as tabelas de encaminhamento sejam modificadas durante o percurso de um pacote.

3. Desenvolvimento do programa

3.1. Código fornecido

Para facilitar o desenvolvimento do programa e tornar possível o desenvolvimento do programa durante as seis horas previstas, é fornecido juntamente com o enunciado um programa *router* incompleto, com a interface gráfica representada abaixo, que já realiza parte das funcionalidades pedidas. Cada grupo pode fazer todas as modificações que quiser ao programa base, ou mesmo, desenhar uma interface gráfica de raiz. No entanto, recomenda-se que invistam o tempo na correcta realização do algoritmo de encaminhamento e no envio dos dados.



O programa fornecido é composto por sete classes:

- *Entry.java* (completa) – Descritor de elemento no vector de um pacote ROUTE;
- *neighbour.java* (completa) – Descritor de vizinho, usado no controlo de vizinhança e no envio de pacotes;
- *neighbourList.java* (completa) – Descritor de lista de vizinhos, usado no controlo de vizinhança e no envio de pacotes para nós vizinhos;
- *MulticastDaemon.java* (completa) – Classe que suporta comunicação multicast (envio/recepção de pacotes);
- *routing.java* (**a completar**) – Classe responsável pelo envio e recepção de pacotes ROUTE, pelo cálculo da tabela de encaminhamento, e pelo tratamento dos pacotes DATA;
- *RouteEntry.java* (**a completar**) – Descritor incompleto de elementos da tabela de encaminhamento;
- *router.java* (completa) – Classe principal com interface gráfica (herda de JFrame), que faz a gestão de sincronismo dos vários objectos usados.

O programa fornecido inclui todos os ficheiros completos excepto os ficheiros *routing.java* e *RouteEntry.java*, que devem ser completados pelos alunos. A activação do socket e da *thread* de recepção de dados e a gestão de vizinhança já é feita no programa fornecido. Falta apenas realizar os aspectos relacionados com o algoritmo de encaminhamento (controlo do envio e processamento de pacotes ROUTE, cálculo da tabela de encaminhamento), a consulta da tabela de encaminhamento e o atraso durante o processamento dos pacotes de dados. Para se poder desenvolver o algoritmo de encaminhamento antes do algoritmo de inundação de pacotes ROUTE é fornecida a classe *MulticastDaemon* que suporta o envio através de *sockets multicast* de pacotes ROUTE. Este modo de transmissão de pacotes deve ser abandonado após a realização do algoritmo de inundação.

A classe *routing* define um conjunto inicial de variáveis com os dados da interface gráfica, inicializados no construtor da classe:

```
private char local_name;       // Local name
private neighbourList neig;    // Neighbour list
private int period;            // Routing update period
private int min_interval;      // Minimum interval between sends
private int data_delay;        // DATA packets' delay time
private router win;            // Main window
```

```
  private DatagramSocket ds;     // Unicast datagram socket
  private JTable tabela;         // window with routing table
```

A classe *router* define um conjunto de métodos que permitem saber o estado das check boxes da interface gráfica:

```
  public boolean is_multicast();     // Use multicast communication
  public boolean is_SendIfChanges(); // React to topology changes
  public boolean is_SlowData();      // Delay DATA packet's
```

A classe *routing* define ainda um conjunto de variáveis locais, que pode e deve ser estendido:

```
  private int local_TTL;       // ROUTE TTL value
  private int route_seq;       // Sequence number for ROUTE packets
  private int data_seq;        // Sequence number for DATA packets
  private Route_queue queue;   // Queue for storing ROUTE packets
  private MulticastDaemon mdaemon;   // Multicast communication
                                     //    support
```

Durante o trabalho devem ser definidas ou completados os seguintes métodos da classe *routing*:

```
/** Construtor da classe routing – iniciar variáveis adicionadas */
public routing(char local_name, neighbourList neig,
    boolean sendIfChanges, int period, int min_interval, int ttl_tx,
    int ttl_ack, boolean use_multi, String multi_addr, int multi_port,
    router win,  DatagramSocket ds, JTable tabela) {…}

/** Arranca o relógio que envia periodicamente o pacote ROUTE */
public boolean start_announce_timer() { … }

/** Pára o relógio que envia periodicamente o pacote ROUTE */
public boolean stop_announce_timer() { … }

/** Chamada quando uma ligação para um vizinho muda – pode ser chamada
     por outros motivos no meio do código a desenvolver */
public void network_changed(boolean network_changed) { … }

/** Termina todas as tarefas de encaminhamento após premir Active */
public void stop() { … }

/** Inicia um pacote ROUTE local – só falta o suporte de unicast */
public boolean send_local_ROUTE(boolean use_multicast)

/** Descodifica pacotes ROUTE e processa-os. É fornecido o código
    Que descodifica o pacote. Falta o processamento do pacote,
    E o seu reenvio, em inundação por unicast. */
public boolean process_ROUTE(char sender, DatagramPacket dp,
            String ip, DataInputStream dis) { … }

/** Calcula a tabela de encaminhamento utilizando o algoritmo de
    Dijkstra, que pode e deve ser realizado noutro método/classe */
private synchronized void update_routing_table() { … }

/** Actualiza conteúdo da tabela de encaminhamento na janela */
public void update_routing_window() { … é fornecida sugestão … }
```

```
/** Envia o pacote DATA com msg para o destino dest – falta realizar
       envio diferido no tempo */
public void send_data_packet(char sender, int seq, char dest,
                    String msg, String path) {…}

/** Consulta tabela de encaminhamento e devolve próx. nó para
      destino dest */
public char next_Hop(char dest) {…}
```

Durante o trabalho pode e deve utilizar os métodos que já são disponibilizados pelas várias classes do trabalho, para enviar pacotes para vizinhos (método *send_packet* da classe *neighbour* ou método *send_packet* da classe *neighbourList*), obter um vector com todos os vizinhos (método *local_vec* da classe *neighbourList*), etc. Desta forma, antes de iniciar o trabalho, o aluno deve ler atentamente todo o código fornecido, de forma a poder tirar total proveito dele.

Os alunos vão poder testar os programas desenvolvidos com uma realização de teste do programa *router*, disponibilizado apenas nas aulas de laboratório.

3.2 Metas

Uma sequência para o desenvolvimento do programa poderá ser:

1. Programar os métodos *start_announce_timer* e *stop_announce_timer*, que respectivamente lançam e param um temporizador que invoca a rotina para enviar o pacote ROUTE. Use inicialmente o socket multicast, para enviar o pacote ROUTE;
2. Definir uma estrutura de dados para guardar os dados dos pacotes ROUTE relativos a cada *router*, incluindo, os vectores *Entry[]*, TTL (validade), o número de sequência, a data de recepção, tudo organizado como uma lista (recomenda-se a utilização de um *HashMap* de uma classe definida pelo aluno (*Router_info*) indexado pelo nome do *router*).
3. Programar a geração da tabela de encaminhamento, e completar as funções *update_routing_table()* e *update_routing_window*(). Esta é a parte mais IMPORTANTE do trabalho. Deve programar o algoritmo de Dijkstra, prevendo situações de erro comuns (partições na rede, letras não sequenciais (A, F, Z), etc.). Defina uma estrutura de dados para guardar a tabela de encaminhamento. A função *update_routing_table()* deve chamada após o envio do pacote ROUTE;
4. Programar a consulta à tabela de encaminhamento, na função *next_Hop*;
5. Programar o algoritmo de inundação de pacotes ROUTE, utilizando agora o socket unicast;
6. Programar o envio diferido de pacotes de Dados;
7. Programar a reacção a modificações na topologia, incluindo a detecção de mudança de topologia através da mudança do conteúdo dos pacotes ROUTE; caso esta modificação ocorra antes de *min_interval*, o relógio de envio periódico deve ser reprogramado para disparar precisamente após *min_interval* msec, voltando à periodicidade normal, após isso.

TODOS os alunos devem tentar concluir **pelo menos a fase 4**. Na primeira semana do trabalho é feita uma introdução geral do trabalho, começando-se a fase 1 do trabalho. No fim da segunda semana devem estar a meio da fase 3. No fim da terceira semana devem ter concluído o passo 5. No fim da quarta e última semana devem tentar realizar o máximo de fases possível, tendo sempre em conta que é preferível fazer menos e bem (a funcionar e sem erros), do que tudo e nada funcionar.

Postura dos Alunos

Cada grupo deve ter em consideração o seguinte:

- Não perca tempo com a estética de entrada e saída de dados
- Programe de acordo com os princípios gerais de uma boa codificação (utilização de indentação, apresentação de comentários, uso de variáveis com nomes conformes às suas funções...) e
- Proceda de modo a que o trabalho a fazer fique equitativamente distribuído pelos dois membros do grupo.