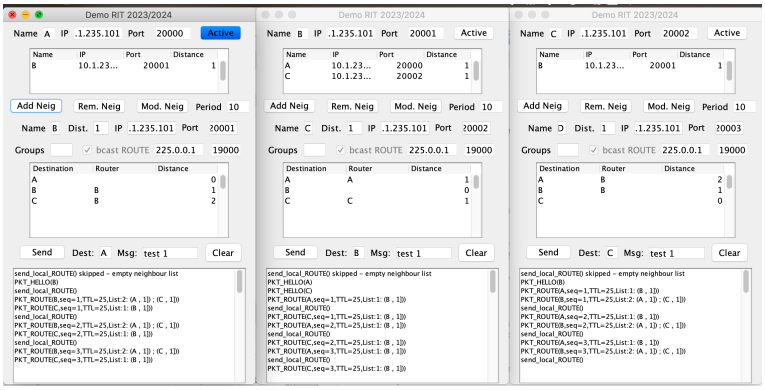
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING



Redes Integradas de Telecomunicações

2023/2024

Trabalho 1:
Encaminhamento dinâmico numa rede em malha

Aulas 2 a 5

Mestrado integrado / Mestrado em
Engenharia Eletrotécnica e de Computadores

https://tele1.deec.fct.unl.pt/rit

Luis Bernardo

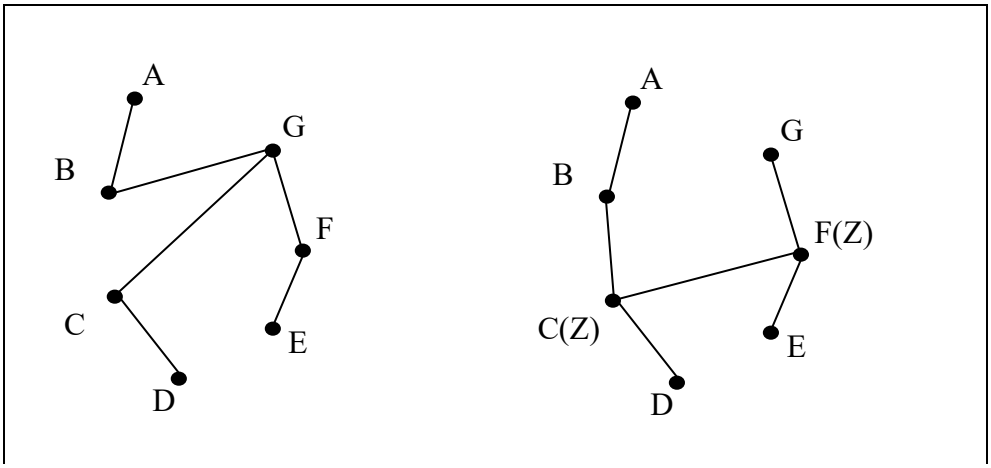
1. Objetivos

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 recetores 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 único: A, B, C, D, etc. Adicionalmente, pode pertencer a um grupo ***anycast***[†], identificado por um nome não único. Embora se esteja a usar uma rede Ethernet no Laboratório, de um ponto de vista lógico as máquinas não podem comunicar diretamente 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 diretos, recebendo informações acerca do resto da rede apenas através do algoritmo de encaminhamento usado – estado de linha. As figuras em baixo ilustram duas hipóteses de redes usando 7 routers: a rede representada à esquerda apenas usa nomes únicos (*unicast*). A rede da direita tem um grupo *anycast* Z, formado pelos nós C e F.



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 https://datatracker.ietf.org/doc/html/rfc2328

[†] Consulte a secção 5.2.9 (pp. 389-390) da 6.ª Edição do livro Computer Networks ou a secção 5.2.9 (pp. 386-387) da 5.ª Edição do livro Computer Networks, para obter uma descrição do funcionamento de um endereço anycast.

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 difusão dos pacotes com o estado da ligação. Periodicamente o processo *router* desencadeia o envio de pacotes de atualização do estado das ligações locais, e recalcula a tabela de encaminhamento local. Esta sequência de ações 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 corretamente 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 receção de pacotes de dados:

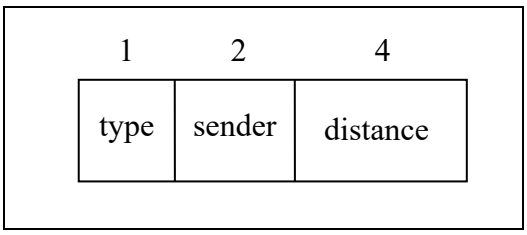
- permite enviar pacotes de dados para qualquer destino;
- recebe e reenvia os pacotes de dados para um *router* vizinho, consoante o conteúdo da tabela de encaminhamento, memorizando no pacote o percurso;
- recebe os pacotes de dados no *router* de destino, apresentando a rota completa percorrida pelo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

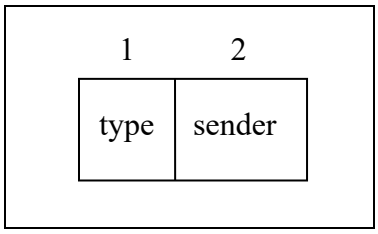
```
Pacote HELLO: { sequência contígua de }
 byte       type;         { tipo pacote  -  HELLO = 21 }
 char       sender;       { nome router origem }
 int        distance;     { distância ao vizinho }
```



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 25. **A distância 25 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 recetor 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 ativo, o programa *router* deve enviar periodicamente para todos os *routers* na rede um pacote ROUTE, com as informações referentes às ligações aos seus vizinhos.

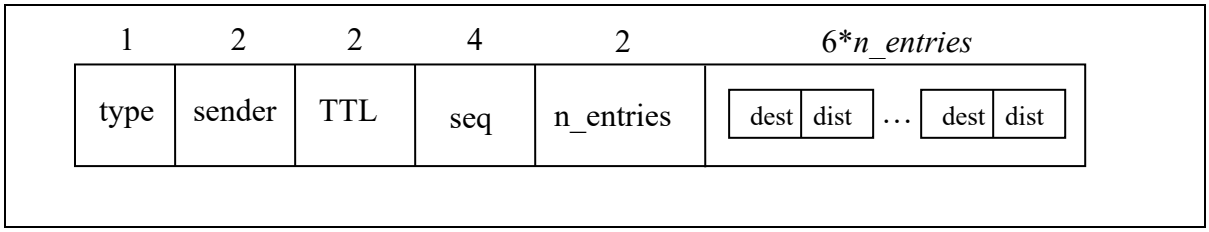
O *router* participa no algoritmo de inundação dos pacotes ROUTE de todos os *routers* para todos os *routers*. Cada *router* mantém uma tabela com o número de sequência do mais recente pacote ROUTE recebido de cada origem. Caso o pacote ROUTE recebido seja mais recente (tenha um número de sequência superior), reenvia-o para todas as ligações, exceto 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 e usado até um tempo máximo igual ao período de vida do pacote (TTL). O valor inicial de TTL é calculado como sendo igual à duração do período de envio de pacotes ROUTE mais quinze segundos

.

Pacote ROUTE: { sequência contígua de }

```
byte       type;          { tipo pacote  -  ROUTE = 31 }
char       sender;        { nome router origem }
short      TTL;           { tempo de vida da informação [s] }
 int       seq;           { número de sequência }
short      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 receção de pacotes HELLO e BYE. Admita ainda que não há perdas na rede local, não sendo necessário, portanto, recorrer a pacotes ACK no processo de inundação do estado da rede.

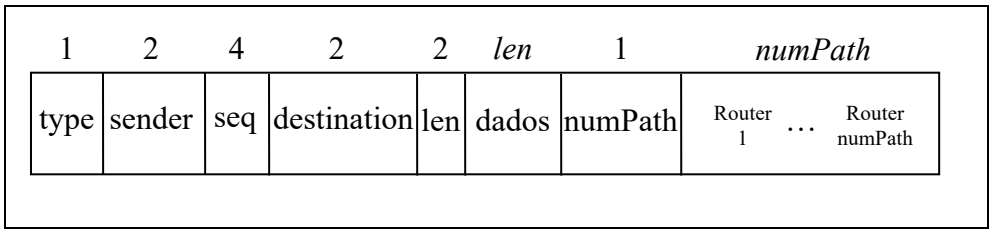
2.3. Algoritmo de encaminhamento

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 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.

2.4. Envio e receção de dados

Os programas *router* também simulam o papel de emissor e recetor de pacotes de dados. Sempre que o utilizador seleciona 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 = 4 }
 char       sender;       { nome router origem }
 int        seq;          { número de sequência }
 char       destination;  { nome router destino }
 short      len;          { 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 (por ser o endereço

local *unicast* ou um dos *anycast*), 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.

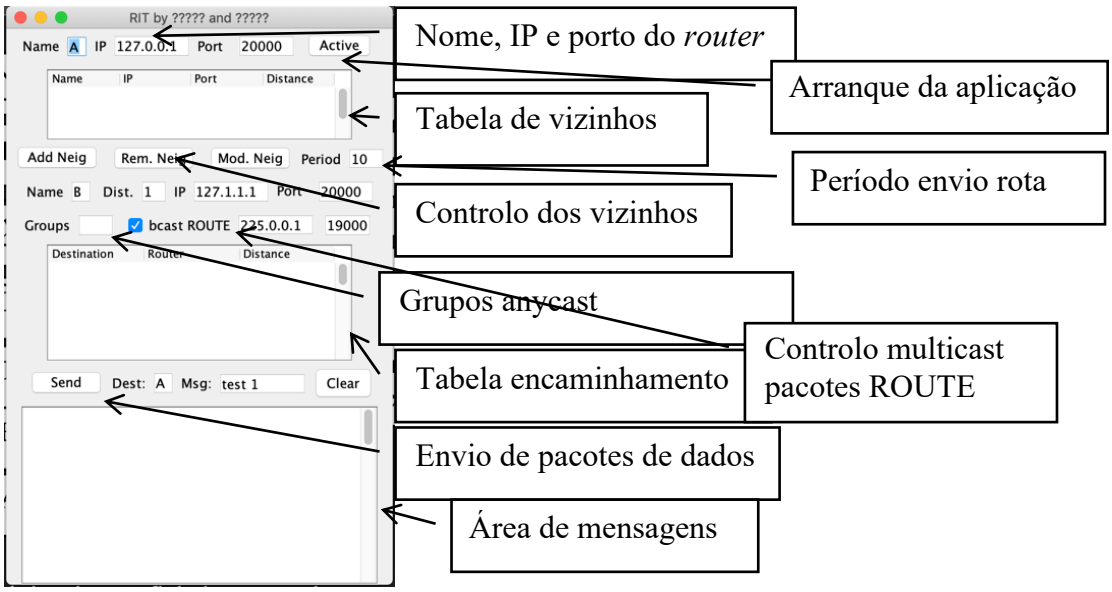
3. Desenvolvimento do programa

3.1. Código fornecido

Para facilitar o desenvolvimento do programa e tornar possível o desenvolvimento do programa durante as oito 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 correta realização do algoritmo de encaminhamento e no envio dos dados.

O programa fornecido é composto por dez classes:

- *Entry.java* (completa) – Descritor de elemento no vetor de um pacote ROUTE;
- *Neighbour.java* (completa) – Descritor de vizinho, usado no controlo de vizinhança e no envio e receção de pacotes HELLO/BYE;
- *NeighbourList.java* (completa) – Descritor de lista de vizinhos, usado no controlo de vizinhança e no envio de pacotes ROUTE/DATA para nós vizinhos;
- *RouteEntry.java* (completa) – Descritor de elementos da tabela de encaminhamento;
- *RoutingTable.java* (completa) – Guarda uma tabela de encaminhamento. Oferece funções para manipular o conteúdo da tabela;
- *Router_info.java* (completar) – Classe responsável pelo armazenamento e gestão da informação recebida num pacote ROUTE;
- *UnicastDaemon.java* (completa) – Classe que suporta comunicação *unicast* (envio/receção de pacotes);
- *MulticastDaemon.java* (completa) – Classe que suporta comunicação multicast (envio/receção de pacotes ROUTE em difusão);
- *Router.java* (completa) – Classe principal com interface gráfica (herda de JFrame), que faz a gestão de sincronismo dos vários objetos usados;
- ***Routing.java* (a completar) – Classe responsável pelo envio e processamento de pacotes ROUTE, pelo cálculo da tabela de encaminhamento e pelo tratamento dos pacotes DATA e ROUTE**.



O programa fornecido inclui todos os ficheiros completos excepto o ficheiro *Routing.java*, que deve ser completado pelos alunos. A ativação dos sockets e das *thread* de receção de dados e a gestão de vizinhança já é feita no programa fornecido. Falta apenas realizar os aspetos relacionados com o algoritmo de encaminhamento (controlo do envio e processamento de pacotes ROUTE, cálculo da tabela de encaminhamento). 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 (s)
private int min_interval;  // Minimum interval between sends (ms)
private Router win;        // Main window
private DatagramSocket ds; // Unicast datagram socket
private JTable tabela;     // GUI 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 BcastROUTE_selected();     // Multicast ROUTE set
public String local_groups();   // String with anycast addresses
public boolean is_local_group(char nm);
                                  // true if nm is local anycast
```

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

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

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

```
/** Construtor da classe routing – iniciar variáveis adicionadas */
public routing(char local_name, NeighbourList neig, int period,
            int min_interval, String multi_addr, int multi_port,
            Router win,  DatagramSocket ds, JTable tableObj) {…}

/** Cria o objeto relógio e arranca-o */
private void run_announce_timer(int initial_delay) {…}

/** 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, boolean mcast) {…}

/** Implement the Dijkstra algorithm */
public RoutingTable run_dijkstra(char origin) {…}

/** Inicia um pacote ROUTE local – só suporta multicast */
public boolean send_local_ROUTE(boolean use_multicast) {…}

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

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* das classes *Neighbour* ou método *send_packet* da classe *NeighbourList*), obter um vetor 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 o método *Routing.run_announce_timer*, para criar e arrancar o temporizador. O método deve chamar a função *send_local_ROUTE*, para enviar o pacote ROUTE (a versão distribuída apenas suporta multicast);
2. Crie e inicialize a variável *map*, para guardar os dados dos pacotes ROUTE relativos a cada *router*, em objetos da classe *RouterInfo*. Defina a variável como um *HashMap* de *RouterInfo* indexado pelo nome do router;
3. Programe a função *process_ROUTE* de maneira a guardar em *map* o conteúdo do pacote ROUTE recebido;
4. Programar a geração da tabela de encaminhamento na função *run_dijkstra*(). Esta é a parte mais IMPORTANTE do trabalho. Deve programar o algoritmo de Dijkstra, devolvendo a tabela de encaminhamento num objeto do tipo *RoutingTable*;
5. Programar o algoritmo de inundação de pacotes ROUTE, utilizando agora o socket unicast. Não esquecer que é necessário decrementar o valor de TTL e testar se é maior que 0;

6. Implementar o encaminhamento *anycast* para grupos. Implementar a receção de pacote anycast nos nós usando a lista de grupos obtida em *win.local_groups()* e o método de teste *win.is_local_group(char)*. Pensem como pode ser feito e implementem!

TODOS os alunos devem tentar concluir **pelo menos a fase 4**. Na primeira semana do trabalho é feita uma introdução geral do trabalho, devendo-se estar na fase 3 do trabalho. No fim da segunda semana deve estar no passo 4. 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.