ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
Departamento de Engenharia de Computação e Sistemas Digitais

PCS 3111 - LABORATÓRIO DE PROGRAMAÇÃO ORIENTADA A OBJETOS PARA A ENGENHARIA ELÉTRICA

EXERCÍCIO PROGRAMA 1 – 2º SEMESTRE DE 2020

Resumo

Os EPs de PCS3111 têm como objetivo exercitar os conceitos de Orientação a Objetos aprendidos em aula ao implementar um software para simular a troca de mensagens entre computadores usando uma rede similar à Internet (mas bem simplificada).

1 Introdução

A Internet é uma rede que trabalha com *comutação de pacotes* ao invés de *comutação de circuito*. Na *comutação de circuito* é feita uma reserva de um canal de comunicação entre a origem e o destino. Por exemplo, imagine a rede apresentada na Figura 1. Ela é composta por diversos nós (representados pelos círculos cinzas e pelos computadores), os quais possuem diversos canais de comunicação com outros nós (representados pelas linhas). Para que o computador A converse com o computador B é necessário reservar um canal de comunicação passando por diversos nós (o 1, 2 e 4), como é exemplificado na figura pelos canais em vermelho.

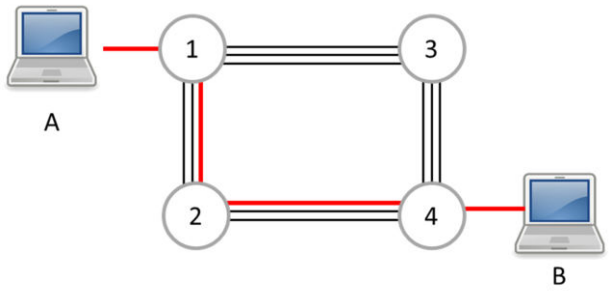


Figura 1: transmissão de mensagens em uma transmissão por comutação de circuitos.

A comutação de circuitos permite uma comunicação com taxa de transferência constante, já que um canal é fisicamente reservado. Mas ela tem algumas desvantagens: ela limita o número de usuários ativos (limitado pelo número de canais de comunicação, no exemplo 3 canais entre cada nó) e desperdiça infraestrutura caso os computadores tenham períodos de silêncio durante a comunicação (por exemplo, caso um computador espere uma ação do usuário para mandar uma nova mensagem).

Na *comutação de pacotes* as mensagens são quebradas em pacotes e cada nó direciona o pacote para um outro nó até que ele chegue ao destino. Ou seja, os pacotes são *repassados* de nó a nó até se chegar ao destino. Com isso, os canais de comunicação não são reservados e podem ser reutilizados por vários computadores. Por exemplo, na Figura 2 se quer transmitir uma mensagem entre os computadores A e B. A mensagem é quebrada em pacotes (representados como retângulos vermelhos), os quais são

transmitidos pelos nós r1, r2 e r4 (representada pela linha tracejada vermelha). Caso o computador X queira transmitir uma mensagem para Y no mesmo instante, os pacotes (representados como retângulos verdes) passariam pelos nós r2, r4 e r3 até chegar ao destino, usando parte dos canais de comunicação usados por A e B.

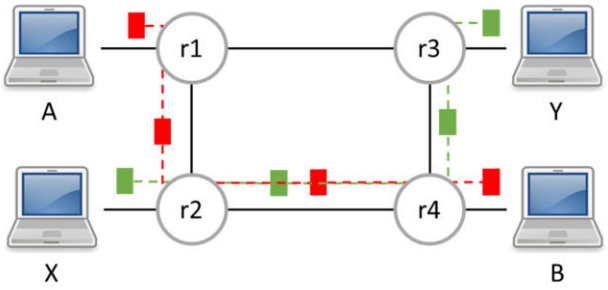


Figura 2: transmissão de pacotes entre computadores em uma transmissão por comutação de pacotes.

1.1 Funcionamento

Neste projeto simularemos a Internet de forma bem simplificada, trabalhando, portanto, com a comutação de pacotes. Quem quiser ver com detalhes como a Internet realmente funciona pode consultar o livro do Kurose e Ross[1]. Note que esse assunto também será tratado por *PTC3360 - Introdução a Redes e Comunicações*, que é uma disciplina do 3º ano de Engenharia Elétrica.

O foco neste primeiro EP é na troca de pacotes entre nós intermediários da rede – os círculos cinza da Figura 2. Na Internet esses nós são chamados de *roteadores*. Esses dispositivos simplesmente encaminham para um outro nó o pacote recebido, até que o pacote chegue ao seu destinatário. Para que eles consigam se conversar, é necessário haver um padrão que define, entre outros detalhes, o formato do pacote e as ações que devem ser tomadas quando se recebe um pacote. Esse padrão é definido através de um *protocolo*. Por exemplo, em um protocolo pode-se definir que o pacote deve conter o endereço da origem, o endereço do destino e um dado. Na Internet o principal protocolo nesta camada[2] é o protocolo IP (*Internet Protocol*) e nele o endereço dos nós é indicado pelo *endereço IP*. Neste simulador, por simplicidade, o endereço será um número inteiro.

Os pacotes recebidos por um roteador são colocados em uma fila[3] para que eles sejam processados. Isso é necessário pois o roteador pode receber vários pacotes ao mesmo tempo e, além disso, o processamento de um pacote não é imediato. Dessa forma, o roteador tira um pacote da fila, o processa, e o repassa. Esse repasse deve ser feito a um dos seus nós adjacentes, apesar de o roteador também receber pacotes cujos destinatários são nós mais distantes. A indicação de qual dos nós adjacentes o roteador deve repassar um pacote está em uma *tabela de repasse*[4], interna ao roteador. Essa tabela

[1] KUROSE, J. F.; ROSS, K. W. *Computer Networking: A top-down approach*. Pearson, 7.ed., 2017.

[2] A arquitetura de uma rede é tipicamente organizada em várias camadas. Neste primeiro EP simularemos apenas uma dessas camadas, que é chamada de *camada de rede*.

[3] Fila é um conceito visto em *PCS3110 – Algoritmos e Estruturas de Dados para a Engenharia Elétrica*. Ela é um conjunto dinâmico que segue a política de que o primeiro elemento a entrar no conjunto é o primeiro a sair.

[4] Essa tabela é tipicamente uma *tabela hash*, assunto explicado em *PCS3110 – Algoritmos e Estruturas de Dados para a Engenharia Elétrica*.

basicamente mapeia endereços a nós adjacentes, existindo um dos nós adjacentes que é definido como padrão - caso o roteador receba um endereço que ele não sabe para quem repassar. Por exemplo, o roteador r2 da Figura 2 pode ter em sua tabela de repasse que o roteador r1 é o padrão e que pacotes para o computador B devem ser repassadas para o roteador r4.

Por fim, existem diversos problemas que podem acontecer em uma rede de comutação de pacotes. Um problema sério é uma falha na tabela de repasse. Por exemplo, na rede da Figura 2 caso o roteador r4 tenha em sua tabela que pacotes para o computador B devem ser repassados para o roteador r2, o pacote ficaria sendo repassado eternamente – sem nunca chegar ao destino, já que o nó r2 devolveria o pacote ao roteador r4! Para evitar esse tipo de problema, os pacotes nessa camada, que serão chamados de *datagrama* (mais detalhes na Seção 2.1), possuem também uma informação chamada de TTL (*time to live*, ou seja, tempo de vida). O TTL indica por quantos nós *no máximo* um datagrama pode ser transmitido.

1.2 Objetivo

O objetivo deste projeto é fazer um simulador de uma rede simplificada de computadores. Este projeto será desenvolvido incrementalmente e **em dupla** nos dois Exercícios Programas de PCS3111.

Neste primeiro EP trabalharemos apenas com roteadores. Os roteadores possuirão uma fila, para armazenar os datagramas recebidos, e uma tabela de repasse. Todos os roteadores ficarão em uma rede.

A solução deve empregar adequadamente conceitos de Orientação a Objetos apresentados na disciplina: classe, objeto, atributo, método, encapsulamento, construtor e destrutor – o que representa o conteúdo até, *inclusive*, a Aula 5. A qualidade do código também será avaliada (nome de atributos/métodos, nome das classes, duplicação de código etc.).

2. Projeto

Deve-se implementar em C++ as classes **Datagrama**, **Fila**, **Rede**, **Roteador** e **TabelaDeRepasse**, além de criar um `main` que permita o funcionamento do programa como desejado.

Atenção:

- O nome das classes e a assinatura dos métodos **devem seguir exatamente** o especificado neste documento. As classes não devem possuir outros membros (atributos ou métodos) **públicos** além dos especificados. Note que você poderá definir atributos e método privados, caso necessário.
- Não é permitida a criação de outras classes além dessas.
- Não faça outros `#defines` além dos definidos neste documento. Use os valores de `#define` deste documento.

O não atendimento a esses pontos pode resultar em **erro de compilação** na correção automática e, portanto, nota `0` na correção automática.

Cada uma das classes deve ter um arquivo de definição (".h") e um arquivo de implementação (".cpp"). Os arquivos devem ter exatamente o nome da classe. Por exemplo, deve-se ter os arquivos "Datagrama.cpp" e "Datagrama.h". Note que você deve criar os arquivos necessários. Não se esqueça de configurar o Code::Blocks para o uso do C++11 (veja a apresentação da Aula 03 para mais detalhes).

2.1 Classe Datagrama

Um **Datagrama** é o pacote que é transmitido entre roteadores. Além do dado a ser transmitido, que neste EP será apenas uma string, o **Datagrama** também possui o endereço de origem (qual nó o enviou) e o de destino (qual nó deve recebe-lo). Uma outra informação contida no **Datagrama** é o TTL (*time to live*), representando por quantos nós no máximo o **Datagrama** deve passar antes de ser destruído - evitando que ele fique eternamente na rede.

A classe **Datagrama** deve possuir apenas os seguintes métodos **públicos**:

```
Datagrama(int origem, int destino, int ttl, string dado);
~Datagrama();
int getOrigem();
int getDestino();
int getTtl();
string getDado();

void processar();
bool ativo();

void imprimir();
```

Os métodos `getOrigem`, `getDestino` e `getDado` devem retornar, respectivamente, os valores do endereço de origem, do endereço de destino e do dado informados no construtor.

O método `getTtl` deve retornar o valor atual do TTL (o valor inicial é informado no construtor). O método `processar` deve decrementar o valor do TTL em uma unidade. Enquanto o TTL for maior que zero, o método `ativo` deve retornar true. O Datagrama ficará inativo (ou seja, o método `ativo` deve retornar false) quando o TTL for menor ou igual a zero.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado.

2.2 Classe Fila

Uma **Fila**[5] é um conjunto dinâmico que segue a política de que o primeiro elemento a entrar no conjunto é o primeiro a sair, assim como as filas que temos no mundo real (filas de cinema, do bandejão, etc.). Essa classe deve implementar uma fila de **Datagramas**, a qual será usada por um **Roteador**.

A classe **Fila** deve possuir apenas os seguintes métodos **públicos**:

```
Fila(int tamanho);
~Fila();

bool enqueue(Datagrama* d);
Datagrama* dequeue();
bool isEmpty();

void imprimir();
```

[5] Seguiremos a mesma nomenclatura usada por *PCS3110 – Algoritmos e Estruturas de Dados para a Engenharia Elétrica*. Mais detalhes do funcionamento e implementação de uma fila podem ser vistos nas videoaulas disponíveis em http://eaulas.usp.br/portal/video.action?idItem=17756, http://eaulas.usp.br/portal/video.action?idItem=17758 e http://eaulas.usp.br/portal/video.action?idItem=17760.

A **Fila** ser implementada como uma *fila circular*, ou seja, o fim da **Fila** pode ser uma posição anterior ao início de forma a evitar desperdício de espaços. O construtor deve receber o tamanho máximo da **Fila**, o qual deve representar o número máximo de elementos que a **Fila** deve efetivamente possuir. Ou seja, se o tamanho for 4, no máximo 4 **Datagramas** poderão ser colocados na fila. Ao tentar fazer o enqueue do 5º **Datagrama** deve ocorrer um *overflow*. No destrutor apenas destrua o vetor alocado dinamicamente, mas não destrua os elementos da **Fila**.

O método enqueue deve inserir o **Datagrama** passado como parâmetro na última posição da **Fila**. Caso a **Fila** não tenha espaço disponível (*overflow*), esse método não deve inserir o **Datagrama** e deve retornar `false`. Caso o **Datagrama** seja colocado na **Fila**, este método deve retornar `true`. O método dequeue deve remover o primeiro **Datagrama** da **Fila** e o retornar. Em caso de *underflow*, ou seja, a tentativa de remover um elemento em uma **Fila** vazia, retorne `NULL`.

O método `isEmpty`[6] informa se a **Fila** está vazia (retornando `true`) ou não (retornando `false`).

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado.

2.3 Classe TabelaDeRepasse

Uma **TabelaDeRepasse** mapeia endereços a **Roteadores**, gerenciando para qual **Roteador** devem ser repassados os **Datagramas** que possuem um determinado endereço de destino. Além disso, ela deve possuir um **Roteador** padrão, para o qual serão repassados os **Datagramas** cujos endereços não estão na **TabelaDeRepasse**. Por simplicidade, recomenda-se que essa classe seja implementada usando dois vetores: um contendo os endereços de destino e o outro contendo os **Roteadores** adjacentes. Quando se fizer um mapeamento de um endereço a um **Roteador** deve-se colocar na próxima posição disponível do vetor de endereços o endereço informado e na mesma posição do vetor de roteadores o **Roteador** associado. No início ambos os vetores estarão vazios. Por exemplo, ao mapear o endereço 4 ao **Roteador** r1, na posição 0 do vetor de endereço ficará o valor 4 e na posição 0 do vetor de roteadores ficará a referência a r1. Se em seguida for mapeado o endereço 7 ao **Roteador** r4, na posição 1 do vetor de endereço ficará o valor 7 e na posição 1 do vetor de roteadores ficará a referência à r4.

Com isso, a classe **TabelaDeRepasse** deve possuir apenas os seguintes métodos **públicos** e este **define**:

```
#define MAXIMO_TABELA 5

TabelaDeRepasse();
~TabelaDeRepasse();

bool mapear(int endereco, Roteador* adjacente);
Roteador** getAdjacentes();
int getQuantidadeDeAdjacentes();

void setPadrao(Roteador* padrao);

Roteador* getDestino(int endereco);

void imprimir();
```

[6] Não usaremos o nome `Queue-Empty` de PCS3110 pois ele é redundante em uma solução Orientada a Objetos – o método é da classe **Fila** (*queue* em inglês) e o nome não precisa repetir essa informação. Além disso, o '-' não é um caractere válido para nomes em C++.

O construtor deve criar uma tabela em que cabem no máximo `MAXIMO_TABELA` endereços de destinos e roteadores adjacentes. No construtor defina o roteador padrão como `NULL`. O destrutor deve destruir os vetores alocados, mas não deve destruir os **Roteadores** adicionados ao vetor.

O método `mapear` deve associar o endereço passado como parâmetro ao **Roteador** adjacente também informado no método. Caso o endereço já esteja na tabela, esse método deve *substituir* o valor do **Roteador** adjacente. Esse método deve retornar `true` se foi possível fazer o mapeamento ou se o valor foi substituído; e deve retornar `false` caso não seja possível fazer o mapeamento pois a tabela já contém `MAXIMO_TABELA` elementos.

Os **Roteadores** mapeados à tabela devem ser obtidos pelo método `getAdjacentes`, que retorna um vetor de **Roteadores** (note que é possível que um **Roteador** apareça várias vezes nesse vetor caso ele seja mapeado a vários endereços). A quantidade de elementos nesse vetor deve ser obtida pelo método `getQuantidadeDeAdjacentes`. Por exemplo, se o vetor tiver os Roteadores {r1, r2, r1, r3, r2} o método `getQuantidadeDeAdjacentes` deve retornar 5.

O método `setPadrao` deve definir o **Roteador** padrão para essa tabela. O **Roteador** padrão deve ser retornado como destino para endereços que não estejam mapeados. Note que o **Roteador** padrão não deve ser retornado pelo método `getAdjacentes`, a menos que ele tenha sido mapeado a um endereço.

O método `getDestino` é o método que retorna para qual **Roteador** deve ser repassado o **Datagrama** cujo destino foi passado como parâmetro. Para isso ele deve considerar os endereços mapeados pelo método `mapear` e o **Roteador** padrão (definido por `setPadrao`). Se o endereço estiver mapeado, o método deve retornar o **Roteador** mapeado ao endereço. Caso o endereço não esteja mapeado, o método deve retornar o **Roteador** padrão. Por exemplo, considere que um **Roteador** r1 teve endereço 4 mapeado ao **Roteador** r4 e tem o **Roteador** r2 como padrão. A chamada `getDestino(4)` deve retornar o **Roteador** r4. Para qualquer outra chamada de `getDestino` deve-se retornar o **Roteador** r2 - por exemplo, se for feito `getDestino(2)` ou `getDestino(8)`.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado. E veja na Seção 4 como lidar com o problema de dependência circular entre essa classe e a classe **Roteador**.

2.4 Classe Roteador

O **Roteador** é o elemento central deste EP e que fará o repasse de **Datagramas**. Como esta rede não possui outros nós além de **Roteadores**, os **Datagramas** terão como origem e destino os **Roteadores**.

A classe **Roteador** deve possuir apenas os seguintes métodos **públicos** e este **define**:

```
#define TAMANHO_FILA 3

Roteador(int endereco);
~Roteador();

TabelaDeRepasse* getTabela();
Fila* getFila();
int getEndereco();
void receber(Datagrama* d);
void processar();
string getUltimoDadoRecebido();
void imprimir();
```

O construtor deve receber o endereço do **Roteador**. Na criação de um **Roteador** você deve criar a **TabelaDeRepasse** e a **Fila** com `TAMANHO_FILA` de tamanho. No destrutor deve-se destruir a **TabelaDeRepasse** e a **Fila** que foram criadas.

O método `getTabela` deve retornar a **TabelaDeRepasse** desse **Roteador**, o método `getFila` a **Fila** e o método `getEndereco` deve retornar o endereço informado no construtor.

O método `receber` deve adicionar o **Datagrama** recebido como parâmetro na **Fila** do **Roteador**. Caso a fila estoure, não adicione o Datagrama e imprima a mensagem:

```
\tFila em <endereço> estourou
```

Onde `<endereço>` é o endereço do **Roteador**. Note o '\t' (tab) para indentar o texto.

O processamento do **Datagrama** só será feito na chamada do método `processar`. Esse método deve retirar 1 (e apenas 1) **Datagrama** da **Fila** e fazer o seguinte:

1. Chamar o método `processar` do **Datagrama**.
2. Caso o **Datagrama** esteja inativo (TTL <= 0), destruir o **Datagrama**.
3. Caso o destino do **Datagrama** seja o endereço deste **Roteador**, deve-se guardar o dado recebido (só é necessário guardar o último dado) e então destruir o **Datagrama**. O último dado recebido será retornado pelo método `getUltimoDadoRecebido`.
4. Caso o destino não seja o endereço deste **Roteador**, deve-se consultar a **TabelaDeRepasse** para descobrir para qual **Roteador** o **Datagrama** deve ser repassado. O **Datagrama** deve ser então repassado para o **Roteador** de destino ao chamar o método `receber` dele. Caso a **TabelaDeRepasse** retorne `NULL`, o **Datagrama** deve ser destruído.

Caso a **Fila** esteja vazia, o método `processar` não deve fazer nada.

O método `getUltimoDadoRecebido` deve retornar o último dado *endereçado* ao **Roteador** que foi recebido. Caso o **Roteador** ainda não tenha recebido um **Datagrama** cujo destino seja ele, este método deve retornar uma `string` vazia ("").

Por exemplo, suponha que o **Roteador** com endereço 2 recebeu (pelo método `receber`) um **Datagrama** `{origem=1, destino=5, ttl=1, dado="Oi"}` e depois um outro **Datagrama** `{origem=4, destino=2, ttl=2, dado="Alo"}`. A primeira chamada do método `processar` deve retirar o **Datagrama** `{origem=1, destino=2, ttl=1, dado="Oi"}` da **Fila** e chamar o método `processar` do **Datagrama**. Como o TTL ficou em 0, esse **Datagrama** ficou inativo, devendo ser destruído, e terminando esta execução do método `processar`. Na próxima chamada do método `processar` será retirado o **Datagrama** `{origem=4, destino=2, ttl=2, dado="Alo"}` da **Fila**. O método `processar` do **Datagrama** deve ser chamado e, como o **Datagrama** ainda ficou ativo, deve-se olhar o endereço de destino dele. Como o destino é o próprio **Roteador**, o dado do **Datagrama** ("Alo") deve ser armazenado como último dado recebido, o **Datagrama** destruído e terminar esta execução do método `processar`.

Para acompanhar o que está acontecendo no **Roteador** devem ser feitas algumas impressões em tela (usando o `cout`) conforme descrito na Seção 3.1. Não faça outras impressões, pois isso pode afetar a correção. Assim como as outras classes, nesta classe é também definido um método `imprimir`, o qual não tem seu funcionamento especificado. Implemente-o como desejado. E veja na Seção 4 como lidar com o problema de dependência circular entre essa classe e a classe **TabelaDeRepasse**.

2.5 Classe Rede

A **Rede** é a classe responsável por ter a lista de **Roteadores**. Ela também cuidará da "passagem do tempo" no nosso simulador: a cada instante de tempo todos os **Roteadores** da **Rede** devem ter seu método `processar` chamado *apenas uma vez*, simulando que cada **Roteador** consegue processar um **Datagrama** por instante de tempo. Como ainda não temos computadores, a **Rede** também é a responsável por enviar um **Datagrama**.

Essa classe deve possuir apenas os seguintes métodos **públicos**:

```
Rede(Roteador** roteadores, int quantidadeDeRoteadores);
~Rede();

Roteador* getRoteador(int endereco);
void enviar(string texto, Roteador* origem, int destino, int ttl);
void passarTempo();

void imprimir();
```

O construtor da **Rede** deve receber um vetor de roteadores e o tamanho desse vetor, passado pelo parâmetro `quantidadeDeRoteadores`. No destrutor não destrua os **Roteadores**.

O método `getRoteador` deve retornar o **Roteador** (dentre os passados no vetor do construtor) que possui o endereço informado. Caso não haja um **Roteador** com esse endereço, este método deve retornar `NULL`.

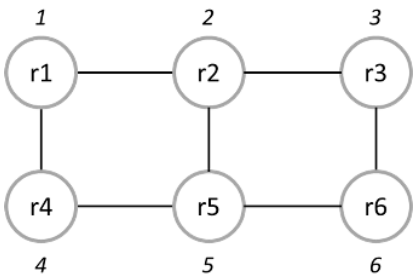
O método `enviar` deve criar um **Datagrama** e enviá-lo ao **Roteador** `origem` (chamando o método `receber` dele). O **Datagrama** deve ter como dado o parâmetro `texto`, como endereço de origem o endereço do **Roteador** `origem` e o `destino` e o `ttl` informados.

O método `passarTempo` deve chamar o método `processar` para todos os **Roteadores** que foram passados no construtor, começando pelo **Roteador** na posição 0 e terminando com o **Roteador** na posição `quantidadeDeRoteadores - 1`.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado.

3 Main

Coloque o `main` em um arquivo em separado, chamado `main.cpp`. Nele você deverá criar a rede apresentada a seguir, composta por 6 roteadores (r1 a r6) e cujos endereços vão de 1 a 6.

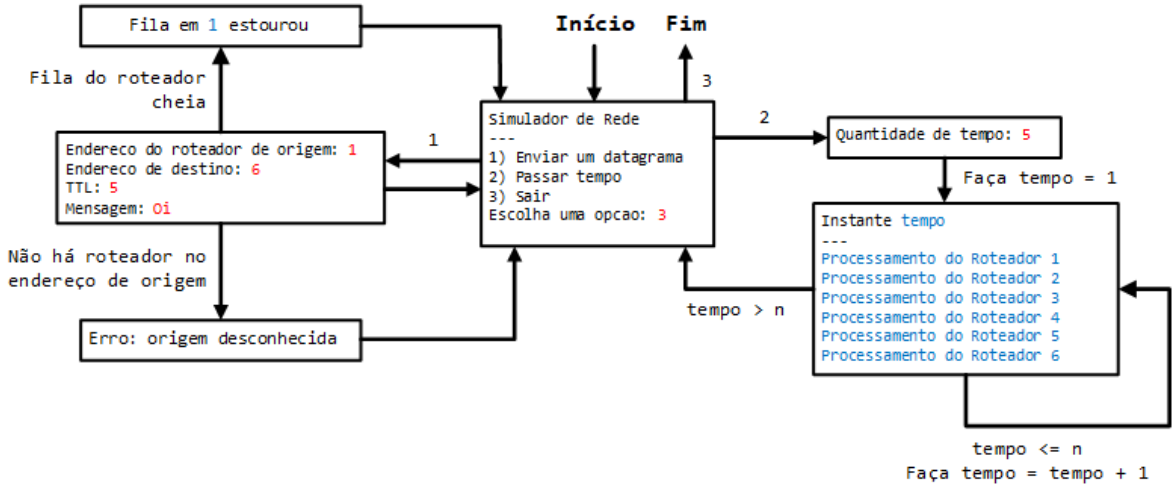


A tabela de repasse dos roteadores ser a apresentada a seguir. Ou seja, o roteador r1 tem o roteador r2 como roteador padrão e repassa ao roteador r4 o endereço 4. Da mesma forma, o roteador r2 tem o roteador r5 como padrão e repassa ao roteador 1 o endereço 1 e ao roteador 3 o endereço 3.

	Endereço						
Roteador	**Padrão**	**1**	**2**	**3**	**4**	**5**	**6**
r1	r2				r4		
r2	r5	r1		r3			
r3	r2						r6
r4	r5	r1					
r5	r2				r4		r6
r6	r5			r3			

3.1 Interface

Além de criar a rede, o `main` deve possuir uma interface em console que permite enviar um **Datagrama** e simular a passagem de tempo. Essa interface é apresentada esquematicamente no diagrama a seguir. Cada retângulo representa uma "tela", ou seja, um conjunto de texto impresso e solicitado ao usuário. As setas representam as transições de uma tela para outra – os textos na seta representam o valor que deve ser digitado para ir para a tela destino ou a condição necessária (quando não há um texto é porque a transição acontece incondicionalmente). Quando a transição apresenta "Faça", considere que é um comando a ser executado. Em **vermelho** são apresentados exemplos de dados inseridos pelo usuário; em **Azul** são as informações que dependem do contexto.



- A opção 1 ("Enviar um datagrama") deve enviar um **Datagrama** usando o método `enviar` da **Rede**.
- A opção 2 ("Passar tempo") deve chamar o método `passarTempo` da **Rede** na quantidade de vezes que for informada como tempo.
- Não deve ser apresentada a informação do processamento do roteador caso o Roteador não tenha **Datagramas** em sua **Fila**. Caso ele possua **Datagramas**, deve ser impresso:

  `Roteador <e>`

  Onde `<e>` é o endereço do **Roteador** como, por exemplo:

```
Roteador 1
```

Além disso, deve ser impresso o resultado do processamento do **Datagrama** pelo **Roteador** da seguinte forma:

- Caso o **Datagrama** retirado da **Fila** seja destruído por TTL <= 0:

  ```
  \tDestruido por TTL: <datagrama>
  ```

  Note que \t é um tab. Por exemplo:

  ```
  \tDestruido por TTL: Origem: 2, Destino: 3, TTL: 0, Exemplo
  ```

- Caso o **Datagrama** retirado da **Fila** seja repassado:

  ```
  \tEnviado para <r>: <datagrama>
  ```

  Onde `<r>` é o **Roteador** para o qual o **Datagrama** foi repassado. Por exemplo:

  ```
  \tEnviado para 2: Origem: 1, Destino: 5, TTL: 3, Algo
  ```

- Caso o **Datagrama** retirado da **Fila** tenha o Roteador como destinatário:

  ```
  \tRecebido: <d>
  ```

  Onde `<d>` é o dado do **Datagrama** como, por exemplo:

  ```
  \tRecebido: Exemplo
  ```

- Em relação a `<datagrama>`, ele deve possuir o seguinte formato:

  ```
  Origem: <o>, Destino: <d>, TTL: <ttl>, <a>
  ```

  Onde:

  - `<o>`: é o endereço de origem no **Datagrama**;
  - `<d>`: é o endereço de destino no **Datagrama**;
  - `<ttl>`: é o valor do TTL (note que o processamento do **Datagrama** é feito logo que o **Datagrama** é retirado da Fila, portanto esse valor já é o processado pelo **Roteador**);
  - `<a>`: é o dado.

  Por exemplo, a impressão do **Datagrama** `{origem=1, destino=4, ttl=2, dado="Exemplo"}` seria:

  ```
  Origem: 1, Destino: 4, TTL: 2, Exemplo
  ```

- No diagrama não são apresentados casos de erro (por exemplo, a digitação de uma opção de menu inválida ou de um texto em um valor que deveria ser um número). Não é necessário fazer tratamento disso. Assuma que o usuário *sempre* digitará um valor correto - a, menos, claro do endereço do **Roteador** de origem e do valor do endereço de destino que pode ser um endereço de um nó que não está na **Rede**.
- Por simplicidade considere que o dado do **Datagrama** não possui espaços.

Atenção: A interface com o usuário deve seguir exatamente a ordem definida (e exemplificada). Se a ordem não for seguida, haverá desconto de nota. Não adicione outros textos além dos apresentados no diagrama e especificados.

3.2 Exemplo

Segue um exemplo de funcionamento do programa com a saída esperada e ressaltando em **vermelho** os dados digitados pelo usuário.

```
Simulador de Rede
---
1) Enviar um datagrama
2) Passar tempo
3) Sair
Escolha uma opcao: 1

Endereco do roteador de origem: 1
Endereco de destino: 6
TTL: 5
Mensagem: Oi

Simulador de Rede
---
1) Enviar um datagrama
2) Passar tempo
3) Sair
Escolha uma opcao: 1

Endereco do roteador de origem: 2
Endereco de destino: 8
TTL: 4
Mensagem: Erro

Simulador de Rede
---
1) Enviar um datagrama
2) Passar tempo
3) Sair
Escolha uma opcao: 2

Quantidade de tempo: 2

Instante 1
---
Roteador 1
        Enviado para 2 Origem: 1, Destino: 6, TTL: 4, Oi
Roteador 2
        Enviado para 5 Origem: 2, Destino: 8, TTL: 3, Erro
Roteador 5
        Enviado para 2 Origem: 2, Destino: 8, TTL: 2, Erro

Instante 2
---
Roteador 2
        Enviado para 5 Origem: 1, Destino: 6, TTL: 3, Oi
Roteador 5
        Enviado para 6 Origem: 1, Destino: 6, TTL: 2, Oi
Roteador 6
        Recebido: Oi

Simulador de Rede
---
1) Enviar um datagrama
2) Passar tempo
3) Sair
Escolha uma opcao: 2

Quantidade de tempo: 3
```

```
Instante 1
---
Roteador 2
        Enviado para 5 Origem: 2, Destino: 8, TTL: 1, Erro
Roteador 5
        Destruido por TTL: Origem: 2, Destino: 8, TTL: 0, Erro

Instante 2
---

Instante 3
---

Simulador de Rede
---
1) Enviar um datagrama
2) Passar tempo
3) Sair
Escolha uma opcao: 3
```

4 Dependência circular

Um problema de compilação é a existência de *dependências circulares*. Caso uma classe **A** use a classe **B** e a classe **B** use a classe **A**, ocorre uma dependência circular. O problema disso é que para compilar a classe **A** é preciso *antes* compilar a classe **B**, mas para compilar a classe **B** é preciso *antes* compilar a classe **A**! Esse problema acontece no EP com as classes **Roteador** e **TabelaDeRepasse**.

Para resolver esse problema em C++ é preciso definir um protótipo da classe. A seguir é apresentado o exemplo com as classes **A** e **B**:

```cpp
#ifndef A_H
#define A_H

#include "B.h"

class B; // Protótipo

class A {
private:
  B* b; // Exemplo de uso de B em A
  ...
};

#endif // A_H
```

```cpp
#ifndef B_H
#define B_H

#include "A.h"

class A; // Protótipo

class B {
private:
  A* a; // Exemplo de uso de A em B
  ...
};

#endif // B_H
```

Note que é feito um `#include` da classe usada, como usual. Porém, após o `#include` é definido um protótipo da outra classe, assim como protótipos de função (note que ambas as classes `A` e `B` precisam ter protótipos). **Mas tome cuidado**: só coloque o protótipo de uma classe se for realmente necessário. Colocar um protótipo de uma classe quando não é necessário pode gerar erros de compilação difíceis de identificar! Neste EP a única situação que há esse problema é na relação das classes **Roteador** e **TabelaDeRepasse.**

Compiladores de outras linguagens resolvem esse problema de outras formas!

5 Entrega

O projeto deverá ser entregue até dia **16/10** em um Judge específico, disponível em <http://judge.pcs.usp.br/pcs3111/ep/> (nos próximos dias vocês receberão um login e uma senha).

As duplas podem ser formadas por alunos de qualquer turma e elas devem ser informadas no e-Disciplinas até dia 02/10. Caso não seja informada a dupla, será considerado que o aluno está fazendo o EP sozinho. Note que no segundo exercício programa deve-se manter a mesma dupla do EP1 (será apenas possível *desfazer* a dupla, mas não formar uma nova).

> **Atenção:** não copie código de um outro grupo. Qualquer tipo de cópia será considerada plágio e **todos** os alunos dos grupos envolvidos terão **nota 0 no EP**. Portanto, **não envie** o seu código para um colega de outro grupo!

Entregue todos os arquivos, inclusive o `main` (que deve **obrigatoriamente** ficar em um arquivo "main.cpp"), em um arquivo comprimido no formato ZIP (outros formatos, como RAR e 7Z, *podem* não ser reconhecidos e acarretar **nota 0**). Os códigos fonte não devem ser colocados em pastas. A submissão pode ser feita por qualquer um dos membros da dupla – recomenda-se que os dois submetam.

> **Atenção:** faça a submissão do mesmo arquivo nos 2 problemas (Parte 1 e Parte 2). Isso é necessário por uma limitação do Judge. Caso isso não seja feito, parte do seu EP não será corrigido – impactando a nota.

Siga a convenção de nomes para os arquivos ".h" e ".cpp". O não atendimento disso pode levar a erros de compilação (e, consequentemente, **nota zero**).

Ao submeter os arquivos no Judge será feita **apenas** uma verificação básica de modo a evitar erros de compilação devidos à erros de digitação no nome das classes e dos métodos públicos. **Note que a nota dada não é a nota final**: nesse momento não são executados testes – o Judge apenas tenta chamar todos os métodos definidos neste documento para todas as classes.

Você pode submeter quantas vezes quiser, sem desconto na nota.

6 Dicas

- Caso o programa esteja travando, execute o programa no modo de depuração do Code::Blocks. O Code::Blocks mostrará a pilha de execução do programa no momento do travamento, o que é bastante útil para descobrir onde o erro acontece!
- Faça `#include` apenas das classes que são usadas naquele arquivo. Por exemplo, se o arquivo .h não usa a classe **X**, mas o .cpp usa essa classe, faça o `include` da classe **X** *apenas* no .cpp. Incluir classes desnecessariamente pode gerar erros de compilação (por causa de referências circulares).
- É muito trabalhoso testar o programa ao executar o `main` *com menus*, já que é necessário informar vários dados para inicializar a rede. Para testar o programa faça o `main` chamar uma função de teste que cria objetos com valores interessantes para testar, sem pedir entrada para o usuário. Não se esqueça de remover a função de teste ao entregar a versão final do EP.

- O método `imprimir` é útil para testes, mas não é obrigatório implementar um comportamento para ele. Por exemplo, se você não quiser implementar esse método para a classe **Rede** você pode fazer no .cpp simplesmente:

```
void Rede::imprimir() {
}
```

- Implemente a solução aos poucos – não deixe para implementar tudo no final.
- Separe o `main` em várias funções para reaproveitar código. Planeje isso!
- Submeta no Judge o código com antecedência para descobrir problemas na sua implementação. É normal acontecerem *RuntimeErrors* e outros tipos de erros no Judge que não aparecem no Code::Blocks (especialmente nas versões antigas do Code::Blocks que usam um outro compilador). Veja a mensagem de erro do Judge para descobrir o problema. Caso você queira testar o projeto em um compilador similar ao do Code::Blocks, use o site https://repl.it/ (note que ele não tem depurador *ainda*).
    - Em geral *RuntimeErrors* acontecem porque você não inicializou um atributo que é usado. Por exemplo, caso você não crie um vetor ou não inicialize o atributo quantidade, para controlar o tamanho do vetor, ocorrerá um *RuntimeError*.
- Use o "Fórum de dúvidas do EP" para esclarecer dúvidas no enunciado ou problemas de submissão no Judge.
- **Evite submeter nos últimos minutos do prazo de entrega. É normal o Judge ficar sobrecarregado com várias submissões e demorar para compilar.**