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 2 – 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 nós usando uma rede similar à Internet (mas bem simplificada). O software desenvolvido no EP1 será melhorado neste EP2 para permitir que o software tenha outras funcionalidades.

1 Introdução

O EP1 tratou da troca de mensagens entre nós intermediários da rede, os *roteadores*. No EP2 simularemos a rede com mais um tipo de nó: os *hospedeiros*. Exemplos de hospedeiros (também chamados de *sistemas finais*) são computadores, celulares, smart TVs e auto falantes inteligentes. Assim como no EP1, quem quiser ver com detalhes como a Internet realmente funciona pode consultar o livro do Kurose e Ross[1]. Esse assunto também será tratado por *PTC3360 - Introdução a Redes e Comunicações*, que é uma disciplina do 3º ano de Engenharia Elétrica.

Da mesma forma que os roteadores, os hospedeiros possuem endereços e recebem pacotes. Mas, além disso, os hospedeiros conseguem executar vários *processos* ao mesmo tempo. Por exemplo, um hospedeiro como um computador pode executar um navegador (cada janela é um processo), o *Code::Blocks*, um leitor de PDF e o *Spotify*. Quando o hospedeiro está em rede (como a Internet), esses processos podem trocar *pacotes* com processos em outros hospedeiros. Por exemplo, um navegador troca pacotes com um servidor de páginas Web para carregar uma página; o programa do *Spotify* troca pacotes com o servidor do *Spotify* para tocar uma música. Com isso, este EP tratará da troca de pacotes entre processos – e não simplesmente a troca de pacotes entre nós[2], que foi o foco do EP1.

A troca de pacotes entre processos tem preocupações diferentes da troca de pacotes entre nós. Por exemplo, pode-se desejar que os pacotes cheguem na ordem que foram enviados e que nenhum pacote seja perdido. Outra preocupação é identificar o processo de destino, ou seja, qual dos vários processos do destinatário que se deseja conversar. A forma de fazer isso é através de um número de *porta*. Cada processo fica esperando os pacotes que são recebidos pelo hospedeiro por uma porta específica. Por exemplo, servidores de páginas Web tipicamente esperam pacotes recebidos pela porta 80 ou 443; o programa do *Spotify* (rodando no dispositivo do usuário) tipicamente espera pacotes da porta 4070.

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

[2] Como comentado no enunciado do EP1, a arquitetura de uma rede é tipicamente organizada em várias camadas. No EP1 simulamos apenas a *camada de rede*, preocupada com a troca de pacotes entre nós. Neste EP2 simularemos também a *camada de transporte*, preocupada com a troca de pacotes entre processos.

Uma vez que a troca de pacotes entre processos possui algumas particularidades, é necessário usar um protocolo específico para fazê-la - na Internet é comum o uso dos protocolos TCP e o UDP. O pacote nessa camada também possui um nome diferente: é chamado de *segmento*.

Para que os segmentos enviados por um processo cheguem ao processo destinatário é necessário que os nós conversem entre si. A forma como uma rede faz isso é similar à forma como enviamos cartas pelos correios. O nome do destinatário e o conteúdo da carta não são relevantes para os correios; eles só precisam olhar o endereço no envelope para fazer com que a carta chegue ao local de destino. Nesse local podem existir vários moradores, então tipicamente alguém do local repassa a carta à pessoa que é o destinatário – e que será quem verá o conteúdo. Em uma rede, as pessoas seriam os *processos* e os armazéns dos correios (onde as cartas ficam armazenadas para serem entregues) seriam os *roteadores*; o *segmento* seria o conteúdo da carta junto com o nome do destinatário, e o *datagrama* seria o envelope sem o nome da pessoa. Portanto, o segmento é o conteúdo do datagrama e os roteadores repassam os pacotes olhando apenas para o datagrama.

1.1 Objetivo

O objetivo deste projeto é fazer um simulador de uma rede simplificada, evoluindo o programa desenvolvido no EP1. Neste EP será possível que a rede possua também hospedeiros, os quais podem executar dois tipos de processos: navegadores e servidores Web. Para tornar o programa mais flexível, também será possível carregar uma rede descrita em um arquivo.

A solução deve empregar adequadamente conceitos de orientação a objetos apresentados na disciplina: classe, objeto, atributo, método, encapsulamento, construtor e destrutor, herança, classe abstrata, membros com escopo de classe, programação defensiva, persistência em arquivo e os containers da STL. A qualidade do código também será avaliada (nome de atributos/métodos, nome das classes, duplicação de código etc.).

Para desenvolver o EP deve-se manter a mesma dupla do EP1. Será possível apenas **desfazer a dupla**, mas não formar uma nova.

2. Projeto

Deve-se implementar em C++ as classes **Datagrama**, **Fila**, **Hospedeiro**, **Navegador**, **No**, **PersistenciaDaRede**, **Processo**, **Rede**, **Roteador**, **Segmento**, **ServidorWeb** e **TabelaDeRepasse**, além de criar um `main` que permita o funcionamento do programa como desejado. 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).

Em relação às exceções (assunto da Aula 9), todas as especificadas são da biblioteca padrão (não se esqueça de fazer `#include <stdexcept>`). O texto usado como motivo da exceção não é especificado no enunciado e não será avaliado. Jogue as exceções criando um objeto usando new. Por exemplo, para jogar um `invalid_argument` faça algo como:

```
throw new invalid_argument("Mensagem de erro");
```

Caso a exceção seja jogada de outra forma, pode haver erros na correção e, consequentemente, desconto na nota.

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**, a menos dos métodos definidos na superclasse e que precisaram ser redefinidos**. Note que você poderá definir atributos e método privados e protegidos caso necessário.
- Não é permitida a criação de outras classes além dessas.
- Não faça `#define` para constantes. Você pode (e deve) fazer `#define` para permitir a inclusão adequada de arquivos.

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

2.1 Classe Segmento

Um **Segmento** é o pacote transmitido entre processos. Ele possui a porta de origem, a porta de destino e o dado que será transmitido. Com isso a classe **Segmento** deve possuir apenas os seguintes métodos **públicos**:

```
Segmento(int portaDeOrigem, int portaDeDestino, string dado);
virtual ~Segmento();

virtual int getPortaDeOrigem();
virtual int getPortaDeDestino();
virtual string getDado();

virtual void imprimir();
```

Os métodos `getPortaDeOrigem`, `getPortaDeDestino` e `getDado` devem retornar, respectivamente, os valores da porta de origem, da porta de destino e do dado informados no construtor.

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

2.2 Classe Datagrama

Um **Datagrama** é o pacote que é transmitido entre roteadores. A diferença do **Datagrama** no EP2 é que o conteúdo não é uma **string**: ele é um **Segmento**. Um impacto disso é no destrutor. Com isso a classe **Datagrama** deve possuir apenas os seguintes métodos **públicos**:

```
Datagrama(int origem, int destino, int ttl, Segmento* dado);
virtual ~Datagrama();

virtual int getOrigem();
virtual int getDestino();
virtual int getTtl();
virtual Segmento* getDado();

virtual void processar();
virtual bool ativo();

virtual 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. No destrutor do **Datagrama** destrua o **Segmento** recebido.

Da mesma forma que no EP1, 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.

2.3 Classe Fila

A **Fila** implementa uma fila de **Datagramas**, a qual será usada por um **No**. Essa classe teve poucas alterações no EP2, apenas para permitir o uso de exceções (note que a assinatura de enqueue mudou por causa disso). Com isso, a classe **Fila** deve possuir apenas os seguintes métodos **públicos**:

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

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

virtual void imprimir();
```

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 (ou outra estrutura) 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 jogar uma exceção do tipo `overflow_error`. 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, jogue uma exceção do tipo `underflow_error`. Tanto `overflow_error` quanto `underflow_error` são da biblioteca padrão.

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

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

2.4 Classe TabelaDeRepasse

Uma **TabelaDeRepasse** mapeia endereços a **Nos**, gerenciando para qual **No** devem ser repassados os **Datagramas** que possuem um determinado endereço de destino. O funcionamento dessa classe é praticamente o mesmo especificado no EP1. A principal diferença é que os métodos trabalham com **Nos** ao invés de **Roteadores**, uma vez que a rede é composta por **Roteadores** e **Hospedeiros**. Além disso, deve-se jogar uma exceção em caso de *overflow* e definiu-se uma constante (`static const`) para o

tamanho máximo da tabela, ao invés de usar um `#define`. Com isso, a classe **TabelaDeRepasse** deve possuir apenas os seguintes métodos **públicos** e a constante:

```
TabelaDeRepasse();
virtual ~TabelaDeRepasse();

virtual void mapear(int endereco, No* adjacente);
virtual No ** getAdjacentes();
virtual int getQuantidadeDeAdjacentes();

virtual void setPadrao(No* padrao);

virtual No* getDestino(int endereco);

virtual void imprimir();
static const int MAXIMO_TABELA = 5;
```

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

O método `mapear` deve associar o endereço passado como parâmetro ao **No** adjacente também informado no método. Caso o endereço já esteja na tabela, esse método deve *substituir* o valor do **No** adjacente. Esse método deve jogar uma exceção do tipo `overflow_error` caso não seja possível fazer o mapeamento pois a tabela já contém `MAXIMO_TABELA` elementos.

Os **Nos** mapeados à tabela devem ser obtidos pelo método `getAdjacentes`, que retorna um vetor de **Nos**. A quantidade de elementos nesse vetor deve ser obtida pelo método `getQuantidadeDeAdjacentes`.

O método `setPadrao` deve definir o **No** padrão para essa tabela.

O método `getDestino` é o método que retorna para qual **No** deve ser repassado o **Datagrama** cujo destino foi passado como parâmetro. Para isso ele deve considerar os endereços mapeados pelo método `mapear` e o **No** padrão (definido por `setPadrao`). Se o endereço estiver mapeado, o método deve retornar o **No** mapeado ao endereço. Caso o endereço não esteja mapeado, o método deve retornar o **No** padrão.

Veja o enunciado do EP1 para uma explicação mais detalhada do funcionamento dos métodos `getAdjacentes`, `setPadrao` e `getDestino` (lembrando-se que agora eles trabalham com **Nos**).

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado. Em relação à constante `MAXIMO_TABELA`, mantenha-a com o valor 5. Por fim, note que agora não há mais a dependência circular, já que essa classe não depende mais de **Roteador**.

2.5 Classe No

O **No** representa elementos da rede que possuem um endereço e recebem **Datagramas**. Um **No** pode ser um **Roteador** ou um **Hospedeiro**. Um **Roteador** é um **No** intermediário da rede que repassa os **Datagramas**, enquanto que um **Hospedeiro** é um **No** que executa processos que se comunicam pela rede.

A classe **No** deve ser uma classe **abstrata**. Escolha o(s) método(s) mais adequado(s) para serem **abstrato(s)**. Em relação ao EP1, esta classe absorveu alguns dos métodos do **Roteador**. A seguir são apresentados os métodos **públicos** dessa classe e a constante:

```
No(int endereco);
virtual ~No();

virtual Fila* getFila();
virtual int getEndereco();

virtual void processar();
virtual void receber(Datagrama* d);

virtual void imprimir();
static const int TAMANHO_FILA = 5;
```

O construtor deve receber o endereço do **No**. Na criação de um **No** você deve criar a **Fila** com `TAMANHO_FILA` de tamanho (mantenha essa constante com valor 5). No destrutor deve-se destruir a **Fila** que foi criada (mas não o seu conteúdo).

O método `getFila` deve retornar a **Fila** criada no construtor e o método `getEndereco` deve retornar o endereço informado no construtor.

O funcionamento do método `processar` depende do tipo de **No** e, por isso, ele será explicado nas classes **Roteador** e **Hospedeiro**.

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 **No**. Note o '\t' (tab) para indentar o texto.

Não é especificado o funcionamento do método `imprimir`. Implemente-o como desejado. Note que para acompanhar o que está acontecendo no **No** e em seus subtipos devem ser feitas algumas impressões em tela (usando o `cout`) conforme descrito na Seção 4.1. Não faça outras impressões, pois isso pode afetar a correção.

2.6 Classe Roteador

O **Roteador** é um subtipo de **No** que faz o repasse de **Datagramas**. Diferentemente do EP1, o **Roteador** não guarda mais a última mensagem recebida (já que isso será uma funcionalidade de um **Processo** do **Hospedeiro**). Com isso ela deve ter os seguintes métodos públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
Roteador(int endereco);
virtual ~Roteador();

virtual TabelaDeRepasse* getTabela();
```

O construtor deve receber o endereço do **Roteador**. Na criação de um **Roteador** você deve criar a **TabelaDeRepasse**; no destrutor deve-se destruí-la. O método `getTabela` deve retornar a **TabelaDeRepasse** desse **Roteador**.

O processamento do **Datagrama** no **Roteador** 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 destruir o **Datagrama**.
4. Caso o destino não seja o endereço deste **Roteador**, deve-se consultar a **TabelaDeRepasse** para descobrir para qual **No** o **Datagrama** deve ser repassado. O **Datagrama** deve ser então repassado para o **No** 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. Um exemplo do processamento do **Roteador** está no enunciado do EP1. Só tome cuidado com a questão dos **Datagramas** endereçados ao **Roteador**.

2.7 Classe Processo

O **Processo** é um software em execução no **Hospedeiro**. No EP os **Processos** podem ser de dois tipos: **Navegadores** e **ServidoresWeb**. A classe **Processo** deve ser uma classe **abstrata**. Escolha o(s) método(s) mais adequado(s) para serem **abstrato(s)**. A seguir são apresentados os métodos **públicos** dessa classe:

```
Processo(int endereco, int porta, Roteador* gateway);
virtual ~Processo();

virtual int getEndereco();
virtual int getPorta();

virtual void receber(int origem, Segmento* mensagem);

virtual void imprimir();

static void setTtlPadrao(int padrao);
static int getTtlPadrao();
```

O construtor deve receber o endereço do **Hospedeiro**, a porta do **Processo** e o *gateway* padrão usado pelo **Hospedeiro**[3] (na classe **Hospedeiro** é explicado o que é o *gateway*). O endereço e o *gateway* são importantes caso o **Processo** precise enviar **Datagramas**. No destrutor *não* destrua o *gateway*.

Os métodos `getEndereco` e `getPorta` apenas retornam os valores do endereço e a porta informados no construtor.

O método `receber` depende do tipo do **Processo** e, por isso, ele será explicado nas classes **Navegador** e **ServidorWeb**. Para acompanhar o recebimento de **Segmentos** no **No** e em seus subtipos devem ser feitas algumas impressões em tela conforme descrito na Seção 4.1.

Quando um **Processo** precisa se comunicar com um **Processo** em outro **Hospedeiro**, ele precisa criar um **Datagrama**. Entre as informações necessárias para o **Datagrama**, o **Processo** precisa informar o TTL[4]

[3] Isso foi feito para simplificar a relação entre o **Processo** e o **Hospedeiro**.

[4] Veja no enunciado do EP1 a explicação do que é o TTL.

(*time to live*). Uma vez que este EP é um simulador, será possível configurar qual é o valor de TTL usado através do método estático `setTtlPadrao`. Inicie o TTL padrão com o valor 5. O valor atual do TTL deve ser informado pelo método `getTtlPadrao`. Com isso, caso se altere o valor do TTL padrão, todos os **Processos** devem usar esse valor na criação de novos **Datagramas**. Não altere o valor dos TTLs dos **Datagramas** já criados.

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

2.8 Classe Navegador

O **Navegador** é um subtipo de **Processo** que representa uma janela de um navegador como o Firefox, Chrome ou Edge. Ele deve ter os seguintes métodos públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
Navegador(int endereco, int porta, Roteador* gateway);
virtual ~Navegador();

virtual void abrir(int endereco, int porta);
virtual void abrir(int endereco);

virtual string getConteudo();
```

O construtor deve receber o endereço do **Hospedeiro**, a porta do **Processo** e o gateway padrão usado pelo **Hospedeiro**. No destrutor não destrua o *gateway*.

O método `abrir` deve enviar um **Datagrama** ao *gateway* solicitando o conteúdo de um **ServidorWeb** cujo endereço e porta são informados. Com isso, o **Datagrama** e o **Segmento** criados devem ter como origem o endereço do **Hospedeiro** e a porta do **Processo**, ambos definidos no construtor. O destino deve ser o endereço e a porta informados como parâmetros do método – no método que não recebe a porta, use como destino a porta 80. No TTL, use o que está definido no método `getTtlPadrao`. Em relação ao conteúdo, use sempre o texto "GET". Uma vez que se chamar o método `abrir`, o **Navegador** deve ficar esperando pela resposta – nesse caso, o conteúdo deve ficar com a string vazia ("").

O método `getConteudo` deve retornar o conteúdo da última mensagem recebida de um **ServidorWeb**. Caso o **Navegador** esteja esperando pela resposta (ou seja, o método `abrir` foi chamado, mas a resposta ainda não foi recebida), esse método deve retornar a string vazia (""). O conteúdo ao criar o **Navegador** também deve ser a string vazia ("").

Quando o **Navegador** recebe um **Segmento**, ele deve verificar se estava esperando por uma resposta. Caso esteja esperando, o conteúdo do **Segmento** deve ser armazenado de modo que o método `getConteudo` retorne agora o seu valor. Note que ao receber um **Segmento** esperado o **Navegador** não deve mais ficar esperando por uma resposta. Caso o **Navegador** não esteja esperando por uma resposta, o pacote recebido deve ser ignorado. (Note que devem ser feitas algumas impressões em tela conforme descrito na Seção 4.1.)

2.9 Classe ServidorWeb

O **ServidorWeb** é um subtipo de **Processo** que representa um servidor de páginas Web. Ele possui um conteúdo fixo, o qual é retornado para qualquer **Processo** que envie um **Segmento** para ele. Dessa forma, esta classe deve ter os seguintes métodos públicos específicos (ou seja, redefina métodos da superclasse caso necessário):

```
ServidorWeb(int endereco, int porta, Roteador* gateway, string conteudo);
virtual ~ServidorWeb();
```

Assim como no **Navegador**, o construtor deve receber o endereço do **Hospedeiro**, a porta do **Processo** e o *gateway padrão* usado pelo **Hospedeiro**. Além disso, ele deve receber uma string com o conteúdo. No destrutor não destrua o *gateway*.

Quando um **ServidorWeb** recebe um **Segmento** ele deve simplesmente retornar à origem do **Segmento** (mesmo endereço e porta) um novo **Datagrama** cujo dado do **Segmento** é o conteúdo informado no construtor. Naturalmente o **Datagrama** e o **Segmento** devem possuir como origem o endereço e porta do **ServidorWeb**. Para retornar o pacote à origem, o **ServidorWeb** deve usar o **Roteador** *gateway*. Note que devem ser feitas algumas impressões em tela conforme descrito na Seção 4.1.

2.10 Classe Hospedeiro

O **Hospedeiro** representa um dispositivo computacional, como um computador, um celular, uma Smart TV, etc. Essa classe é subtipo de **No** que possui **Processos** em execução que conversam com outros **Processos** através da rede. Ele deve ter os seguintes métodos públicos específicos a essa classe (ou seja, redefina métodos da superclasse caso necessário):

```
Hospedeiro(int endereco, Roteador* gateway);
virtual ~Hospedeiro ();

virtual void adicionarServidorWeb(int porta, string conteudo);
virtual void adicionarNavegador(int porta);

virtual Processo* getProcesso(int porta);
virtual vector<Processo*>* getProcessos();
```

O construtor deve receber o endereço do **Hospedeiro** e o seu *gateway padrão* (também chamado de *roteador de borda padrão*, ou simplesmente de *gateway*). O *gateway* é um **Roteador** que permite que as mensagens geradas pelos **Processos** do **Hospedeiro** sejam repasadas na rede.

Os métodos `adicionarServidorWeb` e `adicionarNavegador` devem criar os **Processos** do tipo **ServidorWeb** e **Navegador**, respectivamente, adicionando-o à lista de **Processos** do **Hospedeiro**. Em `adicionarServidorWeb` deve-se receber a porta e o conteúdo do **ServidorWeb**; em `adicionarNavegador` deve-se apenas receber a porta. Em qualquer um desses métodos caso já exista um **Processo** na porta informada, o método deve jogar um `logic_error` (da biblioteca padrão). Ao criar o **ServidorWeb** e o **Navegador** deve-se usar o endereço e o *gateway* do **Hospedeiro**.

Os **Processos** adicionados ao **Hospedeiro** devem ser retornados pelo método `getProcessos`, o qual retorna um `vector` (assunto da Aula 11). O método `getProcesso` deve retornar o **Processo** que está na porta informada ou NULL caso não haja um **Processo** na porta.

No destrutor destrua todos os **Processos** adcionados ao **Hospedeiro**.

Em relação ao processamento do **Datagrama** no **Hospedeiro**, ele deve retirar 1 (e apenas 1) **Datagrama** da **Fila** e fazer o seguinte:

1. Repassar o **Segmento** ao **Processo** que está na porta de destino informada pelo **Segmento**. Isso deve ser feito chamando o método `receber` do **Processo**. Depois disso, o **Datagrama** deve ser destruído (note que isso também destruirá o **Segmento**).
2. Caso não haja um **Processo** na porta informada pelo **Segmento**, destruir o **Datagrama**.

Não se preocupe com os **Datagramas** enviados incorretamente ao **Hospedeiro**. Note que o **Hospedeiro** *não chama* o método `processar` do **Datagrama** e nem verifica o seu TTL.

Faça as impressões em tela conforme descrito na Seção 4.1.

2.11 Classe Rede

A **Rede** é a classe responsável por ter a lista de **Nos**. Ela também cuidará da "passagem do tempo" no nosso simulador: a cada instante de tempo todos os **Nos** da **Rede** devem ter seu método `processar` chamado *apenas uma vez*, assim como no EP1.

Essa classe sofreu algumas alterações no EP2. A principal delas é que a **Rede** agora é composta por **Nos** – e não simplesmente por **Roteadores**. Por isso, a **Rede** não envia mais **Datagramas** – o envio deve ser feito pelos **Processos** dos **Hospedeiros** da **Rede**. Além disso, os **Nos** podem ser adicionados dinamicamente a **Rede**, ao invés de serem definidos no construtor. Também foram definidos dois novos métodos: um para obter um **No** e outro para obter os **Hospedeiros** da **Rede**. Dessa forma, esta classe deve possuir apenas os seguintes métodos **públicos**:

```
Rede();
virtual ~Rede();
virtual void adicionar(No* no);

virtual No* getNo(int endereco);
virtual list<Hospedeiro*>* getHospedeiros();

virtual void passarTempo();
virtual void imprimir();
```

O construtor da **Rede** não recebe parâmetros. No destrutor *destrua* todos os nós adicionados à **Rede**, assim como a estrutura auxiliar criada para armazená-los. Em relação a essa estrutura, use a que você desejar (para simplificar, você pode assumir que não serão criadas **Redes** com mais de 20 **Nós**).

O método `adicionar` deve adicionar o **No** à **Rede**. Caso já exista um **No** na **Rede** com o mesmo endereço, jogue um `logic_error` (da biblioteca padrão) – e não adicione o **No**.

O método `getNo` deve retornar o **No** que possui o endereço informado ou `NULL` caso não haja **No** na **Rede** com esse endereço. O método `getHospedeiros` deve retornar um `list` (assunto da Aula 11) que contém apenas os **Hospedeiros** existentes na **Rede**. Caso a **Rede** não possua **Hospedeiros**, retorne um `list` vazio.

Assim como no EP1, o método `passarTempo` deve chamar o método `processar` para todos os **Nos** adicionados na **Rede**, seguindo a ordem de adição. Por exemplo, se forem adicionados primeiro um **Hospedeiro** com endereço 123, depois um **Roteador** com endereço 120 e finalmente um **Hospedeiro** com endereço 125, primeiro deve-se chamar o método `processar` do **Hospedeiro** com endereço 123, depois do **Roteador** com endereço 120 e então o **Hospedeiro** com endereço 125.

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

3 Persistência

O software permitirá carregar **Redes** salvas em arquivos. Para isso deve ser implementada a classe **PersistenciaDeRede**. A seguir é apresentado o formato do arquivo, um exemplo de arquivo e a especificação da classe.

3.1 Formato do arquivo

A persistência da **Rede** deve seguir o formato de arquivo apresentado a seguir. Entre "<" e ">" são especificados os valores esperados. Por simplicidade, será utilizado um caractere de nova linha ('\n') ou espaço (' ') como delimitador (pode ser qualquer um deles) e *assuma* que não existem espaços no texto do conteúdo do **ServidorWeb**

```
<quantidade de roteadores>
<endereço do roteador 1>
<endereço do roteador 2>
...
<quantidade de hospedeiros>
<hospedeiro 1>
<processo 1 do hospedeiro 1>
<processo 2 do hospedeiro 1>
<hospedeiro 2>
<processo 1 do hospedeiro 2>
<processo 2 do hospedeiro 2>
...
<tabela de repasse do roteador 1>
<tabela de repasse do roteador 2>
...
```

O formato do `hospedeiro` deve seguir o seguinte padrão:

```
<endereço> <gateway> <quantidade de processos>
```

O formato do processo deve seguir o seguinte padrão:

- Se o processo for um **ServidorWeb**:

  ```
  w <porta> <conteúdo sem espaços>
  ```

- Se o processo for um **Navegador**:

  ```
  n <porta>
  ```

Por fim, a tabela de repasse deve ter o seguinte formato:

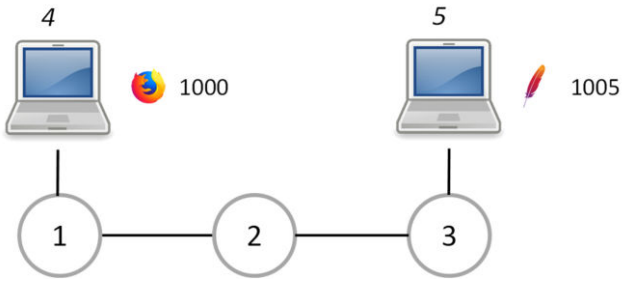
```
<roteador> <roteador padrão> <quantidade de mapeamentos>
<mapeamento 1>
<mapeamento 2>
...
```

O mapeamento deve ser simplesmente:

```
<endereço> <nó destino>
```

Note que há uma linha vazia no final do arquivo.

Por exemplo, considere a **Rede** a seguir, composta por 3 **Roteadores** (os círculos em cinza; os endereços são os números dentro do círculo) e 2 **Hospedeiros** (os notebooks da figura; os endereços são os números acima de cada notebook). O **Hospedeiro** 4 possui um **Navegador** na porta 1000 e o **Hospedeiro** 5 um **ServidorWeb** na porta 1005, com o conteúdo site_do_PCS



Considere que os **Roteadores** 1, 2 e 3 possuem a seguinte **TabelaDeRepasse**:

	Endereço					
Roteador	**Padrão**	**1**	**2**	**3**	**4**	**5**
1	2				4	
2	1			3		3
3	2					5

O arquivo seria:

```
3
1
2
3
2
4 1 1
n 1000
5 3 1
w 1005 site_do_PCS
1 2 1
4 4
2 1 2
3 3
5 3
3 2 1
5 5
```

3.2 Classe PersistenciaDeRede

A classe **PersistenciaDeRede** é a classe responsável por carregar a **Rede** de um arquivo texto. Os únicos métodos públicos que a classe deve possuir são:

```
PersistenciaDeRede();
virtual ~PersistenciaDeRede();

virtual Rede* carregar(string arquivo);
```

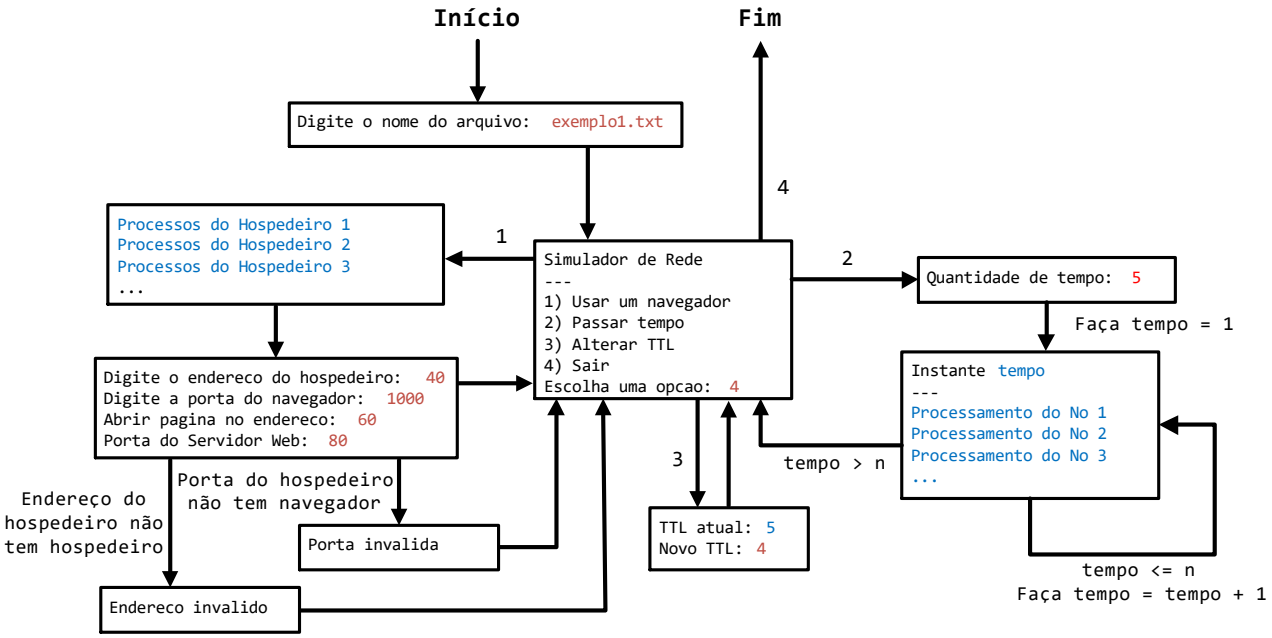
O método `carregar` deve criar uma nova **Rede** a partir dos dados do arquivo informado como parâmetro e retorná-lo. Caso o arquivo não exista ou caso haja algum problema de leitura (erro de formato ou outro problema), jogue uma exceção do tipo `invalid_argument`.

4 Main

Coloque o `main` em um arquivo em separado, chamado `main.cpp`.

4.1 Interface

O `main` deve possuir uma interface em console que permite usar um navegador, simular a passagem de tempo e alterar o TTL padrão. 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.



No início do programa o usuário deve informar o arquivo com o conteúdo da **Rede**, a qual será carregada.

A opção 1 ("Usar um navegador") deve apresentar os processos dos **Hospedeiros** no seguinte formato:

```
Hospedeiro <e>
```

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

```
Hospedeiro 40
```

Além disso, deve ser impresso os nomes e portas de cada um dos **Processos** do **Hospedeiro**, com uma tabulação antes. A ordem dos **Processos** deve ser a ordem de adição deles (a do arquivo texto). A impressão deve ser:

```
\t<Tipo> <p>
```

Onde <Tipo> é ServidorWeb ou Navegador, <p> é a porta do **Processo** e \t é um tab. A ordem de apresentação dos hospedeiros deve seguir a ordem de adição deles na **Rede** (ou seja, a ordem do arquivo texto). Por exemplo, para um **Hospedeiro** no endereço 60 e com três processos, dois **Navegadores** (portas 10 e 20) e um **ServidorWeb** (porta 443) e um outro **Hospedeiro** no endereço 20 com apenas um **Navegador** (porta 50) deve ser apresentado:

```
Hospedeiro 60
	Navegador 10
	Navegador 20
	ServidorWeb 443
Hospedeiro 20
	Navegador 50
```

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 **No** caso ele não tenha **Datagramas** em sua **Fila**. Caso ele possua **Datagramas**, deve ser impresso:

```
<Tipo> <e>
```

Onde <Tipo> é Hospedeiro ou Roteador e <e> é o endereço do **No** como, por exemplo:

```
Roteador 1
```

Além disso, deve ser impresso o resultado do processamento do **Datagrama** pelo **No**.

Caso o **No** seja um **Roteador**, o resultado do processamento deve ser:

- 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: 2:10, Destino: 3:5, TTL: 0, Exemplo
  ```

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

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

  Onde <n> é o endereço do **No** para o qual o **Datagrama** foi repassado. Por exemplo:

  ```
  \tEnviado para 2: Origem: 1:1000, Destino: 5:80, TTL: 3, Algo
  ```

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

  ```
  \tRecebido: <datagrama>
  ```

  Por exemplo:

  ```
  \tRecebido: Origem: 1:1000, Destino: 5:80, TTL: 4, Algo
  ```

- Caso a **TabelaDeRepasse** do **Roteador** não tenha um próximo:

  ```
  \tSemProximo: <datagrama>
  ```

  Por exemplo:

  ```
  \tSemProximo: Origem: 1:1000, Destino: 20:10, TTL: 4, Algo
  ```

Caso o **No** seja um **Hospedeiro**, o resultado do processamento deve ser:

- Caso o **Segmento** seja repassado para um **ServidorWeb**, o **ServidorWeb** deve imprimir o seguinte texto ao receber:

  ```
  ServidorWeb <porta do servidor>
  \tDevolvendo mensagem para: <endereço solicitante>:<porta solicitante>
  ```

  Por exemplo:

  ```
  ServidorWeb 80
  \tDevolvendo mensagem para: 5:1000
  ```

- Caso o **Segmento** seja repassado para um **Navegador**, o **Navegador** deve:
  - Caso esteja esperando uma mensagem, imprimir o seguinte texto ao receber:

    ```
    Navegador <porta>
    \tRecebido de <origem>:<porta da origem>: <conteúdo>
    ```

    Por exemplo:

    ```
    Navegador 1005
    \tRecebido de 40:80: site_da_USP
    ```

  - Caso *não* esteja esperando uma mensagem, imprimir o seguinte texto ao receber:

    ```
    Navegador <porta>
    \tMensagem ignorada <origem>:<porta da origem>: <conteúdo>
    ```

    Por exemplo:

    ```
    Navegador 1005
    \tMensagem ignorada 40:1000: GET
    ```

- Caso não haja um **Processo** na porta informada pelo **Segmento**, o **Hospedeiro** deve imprimir:

  ```
  Sem destino: <datagrama>
  ```

  Por exemplo:

  ```
  Sem destino: Origem: 1:1000, Destino: 20:10, TTL: 4, Algo
  ```

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

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

Onde:

- `<o>`: é o endereço de origem no **Datagrama**;
- `<po>`: é a porta de origem do **Segmento**;
- `<d>`: é o endereço de destino no **Datagrama**;
- `<pd>`: é a porta de destino do **Segmento**;
- `<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 do **Segmento**.

Por exemplo, a impressão do **Datagrama** `{origem=1, destino=4, ttl=2, dado={portaDeOrigem=10, portaDeDestino=80, dado=GET}}` seria:

```
Origem: 1:10, Destino: 4:80, TTL: 2, GET
```

A opção 3 ("Alterar TTL") deve permitir a alteração do TTL padrão do **Processo**.

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 o tratamento disso**. Assuma que o usuário *sempre* digitará um valor correto. Em caso de exceções, capture-as e apresente a mensagem de erro e então encerre o programa.

> **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.

4.2 Exemplo

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

```
Digite o nome do arquivo: ex1.txt

Simulador de Rede
---
1) Usar um navegador
2) Passar tempo
3) Alterar TTL
4) Sair
Escolha uma opcao: 1

Hospedeiro 4
        Navegador 1000
Hospedeiro 5
        ServidorWeb 1005

Digite o endereco do hospedeiro: 4
Digite a porta do navegador: 1000
Abrir pagina no endereco: 5
Porta do Servidor Web: 1005

Simulador de Rede
---
1) Usar um navegador
2) Passar tempo
3) Alterar TTL
4) Sair
Escolha uma opcao: 1

Hospedeiro 4
        Navegador 1000
Hospedeiro 5
        ServidorWeb 1005

Digite o endereco do hospedeiro: 4
Digite a porta do navegador: 1000
Abrir pagina no endereco: 2
Porta do Servidor Web: 80
```

```
Simulador de Rede
---
1) Usar um navegador
2) Passar tempo
3) Alterar TTL
4) Sair
Escolha uma opcao: 3

TTL atual: 5
Novo TTL: 2

Simulador de Rede
---
1) Usar um navegador
2) Passar tempo
3) Alterar TTL
4) Sair
Escolha uma opcao: 1

Hospedeiro 4
        Navegador 1000
Hospedeiro 5
        ServidorWeb 1005

Digite o endereco do hospedeiro: 4
Digite a porta do navegador: 1000
Abrir pagina no endereco: 5
Porta do Servidor Web: 1005

Simulador de Rede
---
1) Usar um navegador
2) Passar tempo
3) Alterar TTL
4) Sair
Escolha uma opcao: 3

TTL atual: 2
Novo TTL: 5

Simulador de Rede
---
1) Usar um navegador
2) Passar tempo
3) Alterar TTL
4) Sair
Escolha uma opcao: 2

Quantidade de tempo: 5

Instante 1
---
Roteador 1
        Enviado para 2: Origem: 4:1000, Destino: 5:1005, TTL: 4, GET
Roteador 2
        Enviado para 3: Origem: 4:1000, Destino: 5:1005, TTL: 3, GET
Roteador 3
        Enviado para 5: Origem: 4:1000, Destino: 5:1005, TTL: 2, GET
Hospedeiro 5
ServidorWeb 1005
        Devolvendo mensagem para: 4:1000

Instante 2
---
Roteador 1
```

```
        Enviado para 2: Origem: 4:1000, Destino: 2:80, TTL: 4, GET
Roteador 2
        Recebido: Origem: 4:1000, Destino: 2:80, TTL: 3, GET
Roteador 3
        Enviado para 2: Origem: 5:1005, Destino: 4:1000, TTL: 4, site_do_PCS

Instante 3
---
Roteador 1
        Enviado para 2: Origem: 4:1000, Destino: 5:1005, TTL: 1, GET
Roteador 2
        Enviado para 1: Origem: 5:1005, Destino: 4:1000, TTL: 3, site_do_PCS

Instante 4
---
Roteador 1
        Enviado para 4: Origem: 5:1005, Destino: 4:1000, TTL: 2, site_do_PCS
Roteador 2
        Destruido por TTL: Origem: 4:1000, Destino: 5:1005, TTL: 0, GET
Hospedeiro 4
Navegador 1000
        Recebido de 5:1005: site_do_PCS

Instante 5
---

Simulador de Rede
---
1) Usar um navegador
2) Passar tempo
3) Alterar TTL
4) Sair
Escolha uma opcao: 4
```

5 Entrega

O projeto deverá ser entregue até dia **04/12** no Judge em <https://laboo.pcs.usp.br/ep/>.

Atenção

- Deve ser mantida a mesma dupla do EP1. É possível apenas *desfazer* a dupla. Com isso, cada aluno deve fazer uma entrega diferente (e em separado). Caso você deseje fazer isso, envie um e-mail para levy.siqueira@usp.br até dia **27/11**.
- Não copie código de um outro grupo. Qualquer tipo de cópia será considerada plágio e os 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 **3 problemas** (Parte 1, Parte 2 e Parte 3). 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

- Implemente a solução aos poucos – não deixe para implementar tudo no final. As classes **Hospedeiro** e **Rede** usam conceitos da Aula 11 – você pode deixá-las para o final ou mesmo deixar os métodos que usam `list` e `vector` para depois.
- Separe o `main` em várias funções, para melhorar a organização e reaproveitar código.
- 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).
- Pode ser trabalhoso testar o programa ao executar o `main` *com menus*. 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 o 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() {
}
```

- Submeta no Judge o código com antecedência para descobrir problemas na sua implementação. É normal acontecerem *Runtime Errors* 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 *Runtime Errors* 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 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.**