UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

ACH2147 – Desenvolvimento de Sistemas de Informação Distribuídos
1^o semestre de 2024

Prof. Dr. Renan Cerqueira Afonso Alves

Exercício Programa

1 Introdução

O objetivo deste exercício programa é implementar um sistema de busca peer-to-peer não estruturado. Cada nó participante da rede armazena um conjunto (possivelmente vazio) de pares chave-valor. Qualquer nó da rede pode iniciar uma busca de um determinado valor de chave. Uma vez que a chave buscada for encontrada, o valor associado é enviado para o requisitante inicial. Deverão ser implementados três métodos de busca: inundação (flooding), caminhada aleatória (random walk) e busca em profundidade. Este tipo de sistema pode servir de base para diversos tipos de aplicações, por exemplo, compartilhamento de arquivos distribuído.

Dicas

1. Implemente o exercícios aos poucos, não deixe para fazer tudo de uma vez ao final do prazo
2. Teste o funcionamento primeiro em topologias simples, mas não deixe de testar topologias mais complexas.
3. Testar o programa através do menu pode ser trabalhoso pois requer a digitação repetida de valores. Portanto, recomenda-se criar testes com valores fixos para agilizar o processo. Não esqueça de remover estes testes antes de entregar a versão final, deixando apenas o menu textual.
4. Pode ser interessante, num primeiro momento, testar os procedimentos que envolvem valores aleatórios com valores determinísticos
5. Dúvidas sobre o enunciado podem ser colocadas no fórum do eDisciplinas específico do EP ou discutidas ao final das aulas. Lembre-se de **não colocar** trechos de seu código no fórum!

2 Especificação

Cada nó pertencente ao sistema deverá ser identificado através de um endereço e de uma porta, funcionando como um servidor TCP. O programa deve receber como parâmetro a sua identificação na forma `<endereço>:<porta>`. Como todos os integrantes do sistema peer-to-peer executam o mesmo código, este parâmetro permite executar diversos nós na mesma máquina para fins de teste, apenas trocando a porta de cada um. O endereço deve ser o endereço IP (vamos considerar apenas IPv4) atribuído a alguma interface de rede disponível, podendo até mesmo ser o endereço de loopback (127.0.0.1), que está sempre disponível.

Além disso, há dois outros parâmetros opcionais: um arquivo de texto contento uma lista de vizinhos conhecidos e um arquivo contento uma lista de pares chave-valor.

```
$ ./ep_distsys <endereco>:<porta> [vizinhos.txt [lista_chave_valor.txt]]
```

A lista de vizinhos deve apenas conter identificadores de outros nós do sistema na forma `<endereço>:<porta>`, um por linha. Estes serão os nós que podem ser contactados diretamente. Um exemplo é mostrado na figura abaixo:

```
192.168.100.50:5000
192.168.100.50:5001
192.168.100.60:5000
192.168.100.70:5001
```

A lista de pares chave-valor deve conter um par chave-valor por linha, na qual a chave é separada do valor por um espaço. Vamos assumir, por simplicidade, que nem a chave nem o valor possuem espaços de nenhum tipo. Um exemplo é mostrado na figura abaixo:

```
Katharine_Hepburn 4
Frances_McDormand 3
Emma_Stone 2
Ingrid_Bergman 2
Jane_Fonda 2
```

2.1 Inicialização

Ao inicializar, o programa deve processar os parâmetros passados:

- Deve ser criado um socket TCP com o endereço e a porta indicados.
- Para cada vizinho conhecido, se houver algum, a seguinte mensagem deve ser exibida na saída padrão: `Tentando adicionar vizinho <endereço>:<porta>`. Em seguida, o nó deve tentar enviar uma mensagem de HELLO para ele (mais detalhes na Seção 2.2.2). Se a mensagem for entregue com sucesso, adicionar o vizinho numa tabela interna.
- Adicionar cada par chave-valor em uma outra tabela interna.

Após realizar estes passos, o nó deve escutar por conexões no seu endereço e porta designados e exibir um menu de opções ao usuário.

2.2 Menu, comandos e mensagens

Após a inicialização, o seguinte menu deverá ser exibido ao usuário:

```
Escolha o comando
    [0] Listar vizinhos
    [1] HELLO
    [2] SEARCH (flooding)
    [3] SEARCH (random walk)
    [4] SEARCH (busca em profundidade)
    [5] Estatisticas
    [6] Alterar valor padrao de TTL
    [9] Sair
```

A maioria dos comando requer enviar alguma mensagem para algum vizinho. Por simplicidade, vamos utilizar um formato de mensagem codificado em texto puro (apesar de não ser muito eficiente), conforme mostrado na Figura 1. Toda mensagem será composta de um cabeçalho com quatro itens, separados por espaço: a identificação do servidor de origem da mensagem (no formato `<endereço>:<porta>`), um número de sequência, um tempo de vida (TTL) e uma operação a ser executada. Dependendo da operação, poderá seguir uma lista de argumentos, também separados por espaço. Em todo caso, a mensagem deve ser terminada por um caractere de nova linha (`\n`). Note que estamos especificando mensagens na camada de aplicação da pilha de protocolos.

```
<ORIGIN> <SEQNO> <TTL> <OPERACAO>[ ARGUMENTOS]\n
```

Figura 1: Formato geral de mensagem

O número de sequência tem a função de diferenciar mensagens de uma dada origem. Sugere-se que a primeira mensagem enviada tenha número de sequência 1, com incremento monotônico a cada nova mensagem enviada.

O tempo de vida deve ter valor padrão inicial 100 para as mensagens de busca. Ao receber uma mensagem, o nó deve primeiro verificar se é o destinatário. Caso a mensagem deva ser reencaminhada, o TTL deve ser decrementado e o reencaminhamento acontece apenas se o TTL for diferente de zero.

Toda tentativa de envio de mensagem deve ser exibida na saída padrão no formato `Encaminhando mensagem <mensagem> para <endereço:porta destino>`. Mensagens enviadas com sucesso devem ser indicadas na saída padrão no formato `Envio feito com sucesso: <mensagem>`. Veja o exemplo:

```
Encaminhando mensagem "127.0.0.1:5002 4 1 HELLO" para 127.0.0.1:5001
       Envio feito com sucesso: "127.0.0.1:5002 4 1 HELLO"
```

Uma mensagem será dada como enviada com sucesso se o transmissor receber como resposta **na mesma conexão** a mensagem `<OPERACAO>_OK`[1]. Caso haja algum erro de conexão ou se não for recebido `<OPERACAO>_OK` como resposta, uma mensagem de erro deve ser exibida na saída padrão.

[1]Note que esta confirmação é em grande parte desnecessária, pois uma camada de transporte confiável está sendo utilizada

2.2.1 Comando `Listar vizinhos`

Este comando é executado localmente, ou seja, não envolve enviar mensagens. Ao executar este comando, o conteúdo da tabela de vizinhos deve ser exibido, um vizinho por linha. Veja o exemplo abaixo:

```
Escolha o comando
       [0] Listar vizinhos
       [1] HELLO
       [2] SEARCH (flooding)
       [3] SEARCH (random walk)
       [4] SEARCH (busca em profundidade)
       [5] Estatisticas
       [6] Alterar valor padrao de TTL
       [9] Sair
0

Ha 3 vizinhos na tabela:
       [0] 127.0.0.1 5001
       [1] 127.0.0.1 5004
       [2] 127.0.0.1 5005
```

2.2.2 Comando `HELLO`

O comando `HELLO` deve exibir a lista de vizinhos, permitindo que o usuário escolha um vizinho para enviar uma mensagem com a operação `HELLO`. Mensagens com a operação `HELLO` não possuem argumentos e devem possuir TTL igual a 1. No lado do transmissor, nenhuma ação precisa ser tomada independentemente do sucesso da operação (exceto mostrar o resultado na saída padrão conforme indicado na Seção 2.2). Veja o exemplo abaixo:

```
Escolha o comando
        [0] Listar vizinhos
        [1] HELLO
        [2] SEARCH (flooding)
        [3] SEARCH (random walk)
        [4] SEARCH (busca em profundidade)
        [5] Estatisticas
        [6] Alterar valor padrao de TTL
        [9] Sair
1

Escolha o vizinho:
Ha 3 vizinhos na tabela:
        [0] 127.0.0.1 5001
        [1] 127.0.0.1 5004
        [2] 127.0.0.1 5005
0
Encaminhando mensagem "127.0.0.1:5002 4 1 HELLO" para 127.0.0.1:5001
        Envio feito com sucesso: "127.0.0.1:5002 4 1 HELLO"
```

Ao receber uma mensagem `HELLO`, o receptor deve adicionar o transmissor na tabela de vizinhos, de acordo com o campo `ORIGIN` da mensagem recebida:

```
Mensagem recebida: "127.0.0.1:5002 1 1 HELLO"
        Adicionando vizinho na tabela: 127.0.0.1:5002
```

Se o vizinho já estiver na tabela de vizinhos, uma aviso deve ser exibido:

```
Mensagem recebida: "127.0.0.1:5002 4 1 HELLO"
        Vizinho ja esta na tabela: 127.0.0.1:5002
```

2.2.3 Comando `SEARCH (flooding)`

Ao acionar o comando `SEARCH (flooding)`, o programa deve pedir que o usuário digite uma chave a ser buscada na rede através do método de inundação (flooding).

Caso a chave esteja na tabela local, o programa deve informar ao usuário e não enviar nenhuma mensagem:

```
Digite a chave a ser buscada
ach2147
Valor na tabela local!
       chave: ach2147 valor: SistemasDistribuidos
```

Caso a chave não esteja na tabela local, uma mensagem de busca deve ser enviada para cada um dos vizinhos conhecidos do nó. As mensagens de busca devem ter o seguinte formato:

```
<ORIGIN> <SEQNO> <TTL> SEARCH <MODE> <LAST_HOP_PORT> <KEY> <HOP_COUNT>
```

Na qual: `MODE` deve ser igual a `"FL"` para busca através de flooding, `LAST_HOP_PORT` indica a porta do servidor que enviou a mensagem[2], `KEY` é a chave sendo buscada e `HOP_COUNT` simboliza a quantidade de saltos que foram realizados até que a mensagem atingisse o nó receptor (igual a 1 no primeiro envio).

Exemplo: `"127.0.0.1:5002 1 100 SEARCH FL 5002 ach2147 1"`.

O pseudocódigo abaixo contém o procedimento para iniciar uma busca por flooding:

```
1  iniciar flooding(CHAVE, SEQNO):
2      MSG = "<ORIGIN> SEQNO <TTL PADRAO> SEARCH FL <SERVER PORT> CHAVE 1"
3      marcar MSG como mensagem ja vista
4      para cada vizinho na tabela de vizinhos:
5          enviar mensagem MSG para vizinho
```

Ao receber uma mensagem de busca por flooding, deve ser verificado nesta ordem: 1) se a mensagem já foi vista antes (em caso positivo, descartar a mensagem; em caso negativo, marcar a mensagem como já vista)[3], 2) se a chave buscada encontra-se na tabela local (em caso positivo, enviar o par chave-valor para a origem da requisição), e 3) decrementar o TTL da mensagem e descartá-la caso o novo valor seja zero.

Por fim, caso nenhuma destas condições for verdadeira, a mensagem deve ter seu contador saltos incrementado e ser reencaminhada para cada um dos vizinhos, exceto o vizinho que enviou a mensagem.

O pseudocódigo da Figura 2 contém o procedimento para processar uma mensagem de busca por flooding. Atente-se às mensagens que devem ser exibidas na saída padrão.

Para enviar o par chave-valor, deve ser utilizada uma mensagem com a operação `VAL`, que possui como argumentos o modo de busca utilizado, a chave buscada, o valor referente à chave e a quantidade de saltos que foi necessário para que a a mensagem de busca chegasse ao nó que contém a chave (ou seja, é o valor de `HOP_COUNT` da mensagem `SEARCH`). Note que os campos do cabeçalho desta mensagem de resposta (`ORIGIN`, `SEQNO` e `TTL`) são referentes ao nó que encontrou a chave, não ao nó que iniciou a busca.

[2] note que o endereço do transmissor pode ser obtivo através das informações da conexão, porém apenas a porta do transmissor como cliente é disponibilizada desta forma.

[3] utilize uma combinação do endereço de origem e número de sequência para identificar a mensagem

```
processar mensagem de flooding(MSG):
    se MSG ja foi vista antes:
        print "Flooding: Mensagem repetida!"
        return

    marcar MSG como mensagem ja vista

    se MSG.KEY esta na tabela de chave-valor:
        print "Chave encontrada!"
        enviar para chave-valor para MSG.ORIGIN
        return

    decrementar TTL de MSG
    se TTL igual a 0:
        print "TTL igual a zero, descartando mensagem"
        return

    incrementar HOP_COUNT de MSG
    para cada vizinho na tabela de vizinhos, exceto transmissor da mensagem:
        enviar mensagem MSG para vizinho
```

Figura 2: Flooding

O formato genérico da mensagem é:

```
<ORIGIN> <SEQNO> <TTL> VAL <MODE> <KEY> <VALUE> <HOP_COUNT>
```

Exemplo: `"127.0.0.1:5001 1 100 VAL FL ach2147 SistemasDistribuidos 3"`

Ao receber uma mensagem com operação `VAL`, o nó receptor deve exibir uma mensagem na saída padrão com o seguinte formato: `"Valor encontrado! Chave: <KEY> valor: <VALUE>"`.

2.2.4 Comando `SEARCH (random walk)`

Ao acionar o comando `SEARCH (random walk)`, o programa deve pedir que o usuário digite uma chave a ser buscada na rede através do método de caminhada aleatória (random walk).

Caso a chave esteja na tabela local, o programa deve informar ao usuário e não enviar mensagem alguma. como no caso da busca por flooding.

Caso a chave não esteja na tabela local, uma mensagem de busca deve ser enviada para um dos vizinhos conhecidos do nó, escolhido aleatoriamente. As mensagens de busca devem ter o seguinte formato:

```
<ORIGIN> <SEQNO> <TTL> SEARCH <MODE> <LAST_HOP_PORT> <KEY> <HOP_COUNT>
```

Na qual: `MODE` deve ser igual a "RW"para buscas através de random walk, `LAST_HOP_PORT` indica a porta do servidor que enviou a mensagem, `KEY` é a chave sendo buscada e `HOP_COUNT` simboliza a quantidade de saltos que foram realizados até que a mensagem atingisse o nó receptor (igual a 1 no primeiro envio).

O pseudocódigo abaixo contém o procedimento para iniciar uma busca por random walk:

```
1  iniciar random walk(CHAVE, SEQNO):
2      MSG = "<ORIGIN> SEQNO <TTL PADRAO> SEARCH RW <SERVER PORT> CHAVE 1"
3      vizinho = um vizinho sorteado aleatoriamente da tabela de vizinhos
4      enviar mensagem MSG para vizinho
```

Ao receber uma mensagem de busca por random walk, deve ser verificado nesta ordem: 1) se a chave buscada encontra-se na tabela local (em caso positivo, enviar o par chave-valor para a origem da requisição), e 2) decrementar o TTL da mensagem e descartá-la caso o novo valor seja zero.

Se nenhuma destas condições for verdadeira, a mensagem deve ter seu contador de saltos incrementado e ser reencaminhada para algum de seus vizinhos, sorteado aleatoriamente. O salto anterior deve ser escolhido com baixa prioridade, ou seja, a mensagem volta pelo caminho que veio apenas se for o único caminho disponível.

O pseudocódigo abaixo contém o procedimento para processar uma mensagem de busca por random walk:

```
1  processar mensagem de random walk(MSG):
2      se MSG.KEY esta na tabela de chave-valor:
3          print "Chave encontrada!"
4          enviar para chave-valor para MSG.ORIGIN
5          return
6
7      decrementar TTL de MSG
8      se TTL igual a 0:
9          print "TTL igual a zero, descartando mensagem"
10         return
11
12     incrementar HOP_COUNT de MSG
13     vizinho = sorteio da tabela de vizinhos
14     enviar mensagem MSG para vizinho
```

O par chave-valor encontrado deve ser enviado para o nó solicitante da mesma forma explicada na Seção 2.2.3, porém com `RW` no campo `MODE`.

2.2.5 Comando `SEARCH (busca em profundidade)`

O último método de busca ser implementado é o comando `SEARCH (busca em profundidade)`. Da mesma forma como nos outros métodos, o programa deve pedir que o usuário digite uma chave a ser buscada na rede através do método de busca em profundidade.

Para executar este tipo de busca, cada nó deverá manter três valores para cada mensagem de busca que passar por ele: o primeiro nó de quem recebeu o pedido de busca (nó mãe), o último nó para o qual encaminhou o pedido de busca (filho ativo), e a lista de vizinhos candidatos que ainda podem receber a mensagem. Lembre-se que uma mensagem pode ser identificada pelo endereço de origem e número de sequência.

Para o nó que inicia a busca, o nó mãe deve ser o próprio endereço. A lista de vizinhos candidatos é inicializada com todos os vizinhos conhecidos. Um dos vizinhos conhecidos do nó deve ser escolhido aleatoriamente para ser receber a primeira mensagem. Este vizinho, que chamaremos de vizinho ativo, deve ser salvo para conferência futura. Além disso, ele deve ser removido da lista de vizinhos candidatos. Por fim, a mensagem é enviada para este vizinho.

As mensagens de busca devem ter o seguinte formato:

```
<ORIGIN> <SEQNO> <TTL> SEARCH <MODE> <LAST_HOP_PORT> <KEY> <HOP_COUNT>
```

Na qual: `MODE` deve ser igual a "BP"para buscas em profundidade, `LAST_HOP_PORT` indica a porta do servidor que enviou a mensagem, `KEY` é a chave sendo buscada e `HOP_COUNT` simboliza a quantidade de saltos que foram realizados até que a mensagem atingisse o nó receptor (igual a 1 no primeiro envio).

O pseudocódigo abaixo contém o procedimento para iniciar uma busca em profundidade:

```
1  iniciar random walk(CHAVE, SEQNO):
2      noh_mae = proprio endereco
3      MSG = "<ORIGIN> SEQNO <TTL PADRAO> SEARCH BP <SERVER PORT> CHAVE 1"
4      vizinhos candidatos = todos os vizinhos da tabela de vizinhos
5      proximo = um vizinho sorteado aleatoriamente dentre os vizinhos candidatos
6      vizinho ativo = proximo
7      remove proximo de vizinhos candidatos
8      enviar mensagem MSG para proximo
```

Ao receber uma mensagem de busca em profundidade, primeiramente devem ser feitas as verificações gerais, nesta ordem: 1) se a chave buscada encontra-se na tabela local (em caso positivo, enviar o par chave-valor para a origem da requisição), e 2) decrementar o TTL da mensagem e descartá-la caso o novo valor seja zero.

Caso a chave não esteja na tabela local, em seguida, devem ser realizados os procedimentos específicos deste tipo de busca.

Se o nó mãe e a lista de vizinhos candidatos ainda não tiverem sido inicializadas, estes devem ser inicializados, respectivamente, com a lista de todos os vizinhos conhecidos, e com o nó que enviou a mensagem (lembre-se que a porta do servidor do salto anterior pode ser encontrada no campo `LAST_HOP_PORT` da mensagem recebida).

Em seguida, o nó que enviou a mensagem deve ser removido da lista de vizinhos candidatos (se estiver nela).

Três condições especiais devem ser verificadas: 1) se a execução do algoritmo chegou ao fim; 2) se um ciclo foi detectado; e 3) se todos os vizinhos já foram consultados.

A condição 1 pode ser detectada somente no nó que iniciou a busca, ou seja, no nó cujo nó mãe é o próprio nó. Além disso, o conjunto de vizinho candidatos deve estar vazio e a mensagem deve ter sido recebida do vizinho ativo. Se todas estas condições forem atingidas, nenhuma mensagem deve ser enviada, e o programa deve exibir na saída padrão que não foi possível localizar a chave.

Um ciclo é detectado (condição 2) caso o nó tenha o seu vizinho ativo definido, porém recebeu a mensagem através de outro vizinho. Neste caso, mensagem deve ser devolvida para o transmissor.

A condição 3 ocorre caso a condição 2 seja falsa e o conjunto de vizinhos candidatos está vazio. Neste caso, a mensagem deve ser encaminhada para o nó mãe.

Se nenhuma destas 3 condições for verdadeira, a mensagem deve ser encaminhada para algum dos vizinhos candidatos, sorteado aleatoriamente. O vizinho ativo torna-se este nó escolhido.

De qualquer forma, a mensagem deve ter seu contador saltos incrementado e ser reencaminhada para o vizinho escolhido.

O pseudocódigo abaixo contém o procedimento para processar uma mensagem de busca em profundidade (atente-se às mensagens impressas na saída padrão):

```
1  processar mensagem de busca em profundidade(MSG):
2      se MSG.KEY esta na tabela de chave-valor:
3          print "Chave encontrada!"
4          enviar para chave-valor para MSG.ORIGIN
5          return
6
7      decrementar TTL de MSG
8      se TTL igual a 0:
9          print "TTL igual a zero, descartando mensagem"
10         return
11
12     se for a primeira vez que MSG eh vista:
13         noh_mae = noh_anterior
14         vizinhos candidatos = todos vizinhos
15
16     vizinhos candidatos = vizinhos candidatos exceto noh_anterior
17
18     # condicao de parada
19     se noh_mae == proprio endereco E
20             vizinho ativo == noh_anterior E
21             vizinhos candidatos esta vazio:
22         print "BP: Nao foi possivel localizar a chave <MSG.KEY>"
23         return
24
25     se vizinho ativo esta definido E vizinho ativo != noh_anterior:
26         print "BP: ciclo detectado, devolvendo a mensagem..."
27         proximo = noh_anterior
28     caso contrario se vizinhos candidatos esta vazio:
29         print "BP: nenhum vizinho encontrou a chave, retrocedendo..."
30         proximo = noh_mae
```

```
31      caso contrario:
32          proximo = um vizinho sorteado dentre os vizinhos candidatos
33          vizinho ativo = proximo
34          remove proximo de vizinhos candidatos
35
36      incrementar HOP_COUNT de MSG
37      enviar mensagem MSG para proximo
```

O par chave-valor encontrado deve ser enviado para o nó solicitante da mesma forma explicada na Seção 2.2.3, porém com `BP` no campo `MODE`.

2.2.6 Comando `Estatisticas`

O comando `Estatisticas` deve imprimir alguns dados a respeito dos métodos de busca utilizados. Cada nó deve contar quantas mensagens de cada tipo de busca foram vistas, ou seja, ao receber uma mensagem de busca, um contador referente ao tipo de busca deve ser incrementado.

Além disso, sempre que que um nó receber uma mensagem com a operação `VAL`, a quantidade de saltos contida na mensagem deve ser armazenada, com a finalidade de calcular a média e desvio padrão do número de saltos que um determinado tipo de busca precisa para encontrar as chaves buscadas.

Lembre-se do formato das mensagens com operação `VAL`:

```
<ORIGIN> <SEQNO> <TTL> VAL <MODE> <KEY> <VALUE> <HOP_COUNT>
```

O campo `MODE` permite identificar o método, enquanto que o campo `HOP_COUNT` indica a quantidade de saltos percorrida até a chave ser encontrada.

De posse destas informações, o programa deve exibir as estatísticas na saída padrão de acordo com o seguinte modelo:

```
Estatisticas
       Total de mensagens de flooding vistas: <valor>
       Total de mensagens de random walk vistas: <valor>
       Total de mensagens de busca em profundidade vistas: <valor>
       Media de saltos ate encontrar destino por flooding: <valores>
       Media de saltos ate encontrar destino por random walk: <valores>
       Media de saltos ate encontrar destino por busca em profundidade: <valores>
```

2.2.7 Comando `Alterar valor padrao de TTL`

Este comando pede para o usuário digitar um novo valor para o TTL inicial com o qual as mensagens de busca são enviadas. Exemplo:

```
Escolha o comando
       [0] Listar vizinhos
       [1] HELLO
       [2] SEARCH (flooding)
       [3] SEARCH (random walk)
       [4] SEARCH (busca em profundidade)
       [5] Estatisticas
       [6] Alterar valor padrao de TTL
       [9] Sair
6

Digite novo valor de TTL
5

Escolha o comando
       [0] Listar vizinhos
       [1] HELLO
       [2] SEARCH (flooding)
       [3] SEARCH (random walk)
       [4] SEARCH (busca em profundidade)
       [5] Estatisticas
       [6] Alterar valor padrao de TTL
       [9] Sair
2
Digite a chave a ser buscada
teste
Encaminhando mensagem "127.0.0.1:5009 1 5 SEARCH FL 5009 teste 1" para 127.0.0.1:5008
```

2.2.8 Comando `Sair`

Por fim, o comando `Sair` deve ser utilizado para encerrar todas as conexões ativas e sair do programa. Antes de encerrar as conexões, o nó deve enviar uma mensagem com a operação `BYE` para cada um dos seus vizinhos conhecidos (esta operação não possui argumentos).

Ao receber uma mensagem com a operação `BYE`, o nó deve remover o nó transmissor de sua tabela de vizinhos, exibindo a informação na saída padrão. Por simplicidade, vamos assumir que todos os nós avisam ao sair rede. Portanto, receber uma mensagem com a operação `BYE` é a única forma de remover um vizinho da tabela.

Exemplo de nó saindo da rede:

```
Escolha o comando
       [0] Listar vizinhos
       [1] HELLO
       [2] SEARCH (flooding)
       [3] SEARCH (random walk)
       [4] SEARCH (busca em profundidade)
       [5] Estatisticas
       [6] Alterar valor padrao de TTL
       [9] Sair
9
Voce escolheu 9
Saindo...
Encaminhando mensagem "127.0.0.1:5009 1 1 BYE" para 127.0.0.1:5010
       Envio feito com sucesso: "127.0.0.1:5009 1 1 BYE"
Encaminhando mensagem "127.0.0.1:5009 2 1 BYE" para 127.0.0.1:5008
       Envio feito com sucesso: "127.0.0.1:5009 2 1 BYE"
```

Exemplo de nó recebendo uma mensagem com a operação `BYE`:

```
Mensagem recebida: "127.0.0.1:5009 2 1 BYE"
       Removendo vizinho da tabela 127.0.0.1:5009
```

3 Implementação

O exercício deverá ser realizado em duplas, a ser informada no eDisciplinas na atividade Escolha de dupla EP. Os integrantes do grupo podem escolher a linguagem de programação que desejarem para implementar o exercício programa, desde que a especificação fornecida seja seguida. Como consequência, será necessário incluir instruções detalhadas de como compilar e executar o código fonte entregue.

A comunicação entre os nós deve ser feita através de sockets TCP. Decisões a respeito de paradigma de programação, organização de código e decisões de projeto em geral ficam a cargo dos integrantes do grupo, e devem ser justificadas no relatório.

Considere que os programas serão testados em um ambiente Linux (Ubuntu 22.04). Parte da correção depende da execução da aplicação. Portanto, a nota final do exercício será afetada se as instruções não puderem ser seguidas.

4 Recursos

Serão disponibilizados no eDisciplinas:

- Fórum de dúvidas dedicado ao exercício programa. Atentem-se para não postar código fonte.
- Arquivos de lista vizinhos a ser passada para o programa para formar algumas topologias
- Exemplos da saída do gabarito

5 Entrega e critérios de avaliação

A entrega deve ser feita no eDisciplinas até o dia 01/06/2024. A entrega deve conter, dentro de um arquivo zip, um relatório em formato PDF, o código fonte e explicações detalhadas de como compilar e executar o código.

O relatório deve ser dividido em três partes: detalhes de implementação, testes de funcionamento e análise de dados.

A primeira parte deve conter as decisões de projeto feitas pelos integrantes do grupo. Exemplos de perguntas interessantes a serem respondidas no relatório:

- Qual o paradigma de programação escolhido?
- Como foi feita a divisão do programa em threads?
- Foi escolhido utilizar operações bloqueantes ou não bloqueantes?

A segunda parte deve conter o resultado de testes em topologias simples, mostrando que as operações estão funcionando corretamente. Sugere-se alguns testes:

- Demonstrar o funcionamento das operações HELLO e BYE
- Demonstrar que a lógica do TTL está sendo aplicada
- Demonstrar que a busca por flooding é capaz de encontrar uma chave que existe na rede

- Demonstrar que as mensagens de busca por flooding não são re-enviadas para o nó remetente
- Demonstrar que as mensagens de busca por flooding repetidas são descartadas
- Demonstrar que a busca por random walk é capaz de encontrar uma chave existente tanto numa rede sem ciclos quanto numa rede com ciclos
- Demonstrar que a busca em profundidade é capaz de encontrar uma chave existente tanto numa rede sem ciclos quanto numa rede com ciclos
- Demonstrar a coleta correta de estatísticas

A terceira parte do relatório envolve executar o programa em pelo menos dois computadores e em um topologia um pouco mais complexa (à escolha do grupo, com pelo menos 10 nós). Em seguida, escolher dois nós da rede para fazer as operações de busca algumas vezes (cada nó deve buscar sempre a mesma chave). Analise as estatísticas obtidas. Elas estão de acordo com o esperado?

A nota será atribuída de acordo com o seguinte critério:

Execução do código	3 pontos
Relatório Parte 1	2 pontos
Relatório Parte 2	2 pontos
Relatório Parte 3	3 pontos
Bônus	0.5 pontos

A pontuação bônus será concedida se for incluída uma quarta parte no relatório com uma tentativa de interoperar a operação do programa com a implementação de uma outra dupla. Um teste com poucos nós em uma topologia simples é o suficiente. Documente o que funcionou e o que não funcionou.

Será requerido o preenchimento de questionário a respeito do EP: `https://forms.gle/Bufgcra1YA4qYHpq8`. Somente será atribuída nota ao EP se o questionário for respondido.

Se for detectado plágio, todas as duplas envolvidas terão a nota do EP zerada.

Bom exercício!