Um Sistema *Peer-to-peer* de Armazenamento Chave-valor

Execução: Em grupo de até três alunos
Data de entrega: 10 de julho

Introdução

Este trabalho tem por objetivo fazer com que os alunos experimentem na prática com as decisões de projeto necessárias para a implementação de um sistema de armazenamento chave-valor (*key-value store*) entre pares, sem servidor, em arquitetura frequentemente denominada *peer-to-peer*.

Dessa forma, o trabalho deve expor os alunos a pelo menos dois aspectos de projeto em redes de computadores: os aspectos de implementação de um algoritmo distribuído sobre um ambiente composto por um número desconhecido de participantes, e uma solução para roteamento em uma topologia não conhecida *a-priori*.

As seções a seguir descrevem o projeto em linhas gerais. Alguns detalhes são definidos, mas diversas decisões de funcionalidade, projeto e implementação estão a cargo dos alunos.

O Problema

A idéia deste trabalho é implementar a funcionalidade básica de um sistema de armazenamento chave-valor do tipo *peer-to-peer*, onde os programas de todos os usuários da rede podem agir simultaneamente como cliente e servidor. Como no trabalhos anterior, pode-se utilizar as linguagens C/C++, Java e Python, sem módulos especiais (siga as recomendações na seção de restrições e dicas).

Vocês devem implementar um protocolo em nível de aplicação, utilizando interface de *sockets* UDP. Dois programas devem ser desenvolvidos:

1. Um programa do sistema *peer-to-peer*, às vezes denominado *servent* (de *server/client*), que será responsável pelo armazenamento da base de dados chave-valor e pelo controle da troca de mensagens com seus pares.
2. Um programa cliente, que receberá do usuário chaves que devem ser consultadas e exibirá os resultados recebidos.

Um programa do sistema, ao ser iniciado, deverá receber um número de porto onde escutará por mensagens (`localport`), o nome de um arquivo contendo um conjunto de chaves associadas com valores (`key-values`, mais detalhes à frente) e uma lista de endereços de outras instâncias desse mesmo programa que estarão executando no sistema (em format `IP:porto`. A linha abaixo mostra um exemplo de invocação ao programa:

```
serventTP3 <localport> <key-values> <ip1:port1> ... <ipN:portN>
```

O programa *servent* deve então ler o arquivo key-values e criar um dicionário onde os pares chave-valor serão armazenados e abrir um socket UDP no porto local indicado e ficar esperando por mensagens.

A lista de pares `IP:porto` recebida na linha de comando identifica os pares que serão *vizinhos* daquele nó. Cada nó pode trocar mensagem com seus vizinhos, e a rede *peer-to-peer* é formada pelas vizinhanças formadas entre os nós da rede, criando uma rede sobreposta (*overlay*).

O programa cliente deve ser disparado com o endereço e porto de um *servent* da rede sobreposta que será seu ponto de contato com o sistema distribuído:

```
clientTP3 <IP:port>
```

O client deve então esperar que o usuário digite uma chave, montar uma mensagem de consulta e enviá-la para o ponto de contato.

O protocolo de comunicação entre os pares já está pré-definido e será um protocolo de alagamento confiável, como o utilizado pelo OSPF, que é a opção mais simples para esse tipo de problema.

O Arquivo de Dados Chave-valor

Por simplicidade, vamos definir a base de dados chave-valor como um arquivo texto simples, onde a primeira palavra em uma linha representa a chave e o restante da linha é o valor associado à chave. Caracteres de espaçamento (*whitespace*) ao início e ao final da chave devem ser removidos (por exemplo, usando `str.strip` em Python). Para facilitar o aproveitamento de arquivos já existentes, considere que linhas começando com um caractere de tralha (#) são comentários e podem ser ignoradas. Ocorrências de tralha no meio de uma linha devem ser considerados como parte do valor. Se uma chave aparecer mais de uma vez em um arquivo, deve-se guardar o último valor. Sendo assim, um pedaço do arquivo `/etc/services` de distribuições padrões do Linux poderia ser uma entrada válida:

```
# WELL KNOWN PORT NUMBERS
#
rtmp              1/ddp    #Routing Table Maintenance Protocol
tcpmux            1/udp    # TCP Port Service Multiplexer
tcpmux            1/tcp    # TCP Port Service Multiplexer
#                          Mark Lottor <MKL@nisc.sri.com>
nbp               2/ddp    #Name Binding Protocol
compressnet       2/udp     # Management Utility
compressnet       2/tcp     # Management Utility
```

Esse trecho definiria quatro chaves: `rtmp`, `tcpmux`, `nbp` e `compressnet` com os respectivos valores:

- `1/ddp    #Routing Table Maintenance Protocol`
- `1/tcp    # TCP Port Service Multiplexer`
- `2/ddp    #Name Binding Protocol`
- `2/tcp     # Management Utility`

O Protocolo

A comunicação entre os programas cliente e o seu ponto de contato e entre os pares se dá através de mensagens UDP. O cliente envia uma mensagem UDP para o seu ponto de contato contendo apenas um campo de tipo de mensagem (`uint16_t`) com valor 1 (CLIREQ) e o texto da chave. Se ele esperar por 4 segundos e não receber nenhuma resposta, ele deve retransmitir a consulta *uma vez apenas* e voltar a esperar. Se ele receber uma resposta, ele não sabe quantas outras respostas ele pode receber, pois vários nós podem conter dados para uma mesma chave. Sendo assim, seu cliente deve entrar em um *loop* lendo as mensagens de resposta que porventura receba, até que ele fique esperando por 4 segundos sem receber novas respostas. À medida que as respostas cheguem ele pode exibi-las para o usuário, indicando que par respondeu (definido pelo par `IP:porto`) e ao final do tempo de espera indicar que não há mais respostas.

Note que para realizar a temporização será necessário utilizar um temporizador para fazer todo `recvfrom` retornar em no máximo 4 segundos. Para isso, você pode utilizar `socket.settimeout` em Python ou `setsockopt` em C. Alternativamente, você pode utilizar o temporizador embutido na função select. Caso nenhum pacote seja recebido dentro de 4 segundos, a função `recvfrom` irá retornar e você deverá tratar o erro.

Os *servents* trocam apenas um tipo de mensagem entre si, que tem os seguintes campos:

1. Um campo de tipo de mensagem (`uint16_t`) com valor 2 (QUERY),
2. Um campo de TTL (`uint16_t`),
3. O endereço IP (`struct in_addr`) e o número do porto (`uint16_t`) do programa cliente que fez a consulta,
4. Um campo de número de sequência (`uint32_t`) e
5. O texto da chave pela qual o cliente está buscando.

Para permitir interoperabilidade, todos os campos inteiros da estrutura acima devem estar em *network byte order*. O *servent* que recebe a consulta do cliente (CLIREQ) gera a primeira mensagem QUERY associada, preenchendo o IP e o número de porto do cliente (que podem ser obtidos do `recvfrom`), inicializa o campo TTL com o valor 3 e preenche o número de sequência com o valor de um contador que é incrementado a cada mensagem CLIREQ recebida. Essa mensagem é então repassada a todos os seus vizinhos. Em sequência, o *servent* deve procurar a chave no dicionário local e responder ao cliente se a chave for encontrada. Note que se o

cliente retransmitir uma consulta, do ponto de vista do servidor, a segunda mensagem receberá um novo número de sequência.

Cada *servent* deve implementar um protocolo de alagamento confiável (*reliable flooding*), semelhante ao usado pelo OSPF para disseminar as mensagens de QUERY. Para isso, todo nó que recebe uma QUERY deve primeiro certificar-se de que a mensagem não foi recebida anteriormente, mantendo um dicionário de mensagens já vistas (as mensagens devem ser identificadas pelos campos de IP e porto do cliente, número de sequência, e chave). Se a mensagem não foi vista anteriormente, o *servent* deve procurar pela chave no seu dicionário local e enviar uma resposta para o cliente se encontrá-la. Além disso, o *servent* deve decrementar o valor do TTL e, se o valor resultante for maior que zero, ele deve retransmitir a mensagem para seus vizinhos, menos aquele do qual recebeu a mensagem.

Note que cada *servent* altera apenas o TTL da mensagem de QUERY. Todos os demais campos permanecem com os valores preenchidos pelo *servent* que criou a mensagem. Como o valor inicial do TTL é 3, se houverem cinco *servents* conectados em sequência e o primeiro da cadeia receber uma mensagem de um cliente, a QUERY gerada por ele atingirá apenas 3 *servents* depois dele. Isto é, o quinto *servent* na cadeia não veria a consulta. Em um cenário prático um TTL maior seria recomendável, mas neste trabalho vamos mantê-lo em 3 para simplificar. Não use um TTL inicial maior.

Qualquer *servent* que encontre a chave indicada em uma mensagem de QUERY no seu dicionário local deve enviar uma mensagem RESPONSE diretamente para o cliente que fez a consulta, que pode ser identificado pelos campos de endereço e porto na mensagem QUERY. A mensagem RESPONSE também é simples, contendo apenas um campo de tipo de mensagem (`uint16_t`) com valor 3 (RESPONSE) e um string contendo a chave, um caractere de tabulação e o valor associado. O string deve ser terminado com um byte nulo `'\0'`.

Para fins de dimensionamento, considere que uma chave tem no máximo 40 caracteres e o valor associado tem no máximo 160 caracteres.O caractere nulo sempre tem que ser adicionado e ele deve ser considerado além desses limites (os *buffers* de recepção precisam ser dimensionados para receber um string de pelo menos 201 bytes (40 + 160 + nulo)). Nota: Os *buffers* de recepção são sempre dimensionados pelo tamanho máximo da mensagem (RESPONSE) que pode ser recebida.

Relatório e Scripts

Cada grupo deve entregar junto com o código um relatório em formato PDF que deve conter uma descrição da arquitetura adotada para o servidor, os refinamentos das operações identificadas e implementadas pelo código, as estruturas de dados utilizadas e decisões de projeto de protocolo não documentadas nesta especificação. Como sugestão, considere incluir as seguintes seções no relatório:

1. Introdução

2. Arquitetura
3. Servent
4. Cliente
5. Discussão

Não se esqueça de descrever como o seu programa foi testado e de incluir os arquivos usados como entrada para gerar a base de registros chave-valor. Note que a forma de execução dos programas cliente e servidor já foi definida (o que cada um deve receber como parâmetros de linha de comando e como devem se comportar).

Restrições e Dicas

Podem ser utilizadas apenas as bibliotecas padrões de cada linguagem de programação. Em particular, a comunicação em rede deve ser realizada com funções compatíveis com o padrão POSIX.

Lembramos que soquetes UDP não estabelecem conexão (não é necessário chamar a função `connect`). Recomendamos o uso das funções `sendto` e `recvfrom` para facilitar captura das informações do endereço IP e porto do *client* e *servent* que enviaram a mensagem. Além disso, como o UDP é orientado a pacotes (datagramas), todas as mensagens chegam necessariamente da forma como foram enviadas e não podem se juntar a outras mensagens, então não é preciso ter os mesmos cuidados que com o TCP.

Outras dicas:

- Poste suas dúvidas como comentários neste documento para que fiquem publicamente acessíveis e para que possamos melhorar a especificação.
- Procure escrever seu código de maneira clara, com comentários pontuais e bem indentado.
- Consulte os professores antes de usar qualquer módulo ou estrutura diferente dos indicados.
- Não se esqueça de enviar o código junto com a documentação.
- Implemente o trabalho por partes. Por exemplo, crie o formato da mensagem e tenha certeza que o envio e recebimento da mesma está correto antes de se envolver com a lógica da entrega confiável e sistema de mensagens.