Redes de Computadores

# TP3: um sistema *peer-to-peer* de armazenamento chave-valor

Trabalho individual ou em duplas

Data de entrega: verifique o moodle da turma
**(Não serão aceitos trabalhos fora do prazo)**
Enunciado do trabalho em PDF

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 (frequentemente denominado *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 aqui 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 nos trabalhos anteriores, deve-se usar a linguagem Python, sem módulos especiais (siga as recomendações em "Módulos recomendados"). Vocês devem implementar um protocolo em nível de aplicação, utilizando interface de *sockets* UDP que funcione tanto com topologias emuladas pelo mininet quanto diretamente na rede local.

Dois programas devem ser desenvolvidos: 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, e um programa cliente, que receberá do usuário chaves que devem ser consultadas e exibirá os resultados das consultas que forem recebidos.

Um programa do sistema, ao ser iniciado, deverá receber um número de porto onde escutará por mensagens, o nome de um arquivo contendo um conjunto de chaves associadas com valores (mais detalhes à frente) e uma lista de (até 10) endereços de outras instâncias desse mesmo programa que estarão executando no sistema:

```
serventTP3 meu-porto nome-do-arquivo.txt ip1:porto1 ip2:porto2 ip3:porto3 ...
```

Aquele *servent* deve então ler o arquivo e criar um dicionário onde os pares chave-valor serão armazenados e abrir um socket UDP no porto indicado e ficar esperando por mensagens naquele socket. A lista de pares IP:porto identifica os pares que serão "vizinhos" daquele nó. Assim, cada par pode ser considerado como ligado aos seus vizinhos e a rede formada pelas ligações entre eles cria uma rede sobreposta (*overlay*).

O programa cliente deve ser disparado com o endereço de um dos nós da rede sobreposta que será seu ponto de contato com o sistema distribuído:

```
clientTP3 ip-servent:porto-servent
```

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

Ao contrário do que foi mencionado em sala, o protocolo de comunicação entre os pares já está pré-definido e será um protocolo de alagamenteo confiável, 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, que deve ser seguido por um

ou mais caractere(s) de tabulação (tab), é o valor associado à chave. Para facilitar o aproveitamento de arquivos já existentes, considere que linhas que começam com `#` deve ser ignoradas e se uma chave aparecer mais de uma vez no arquivo, deve-se guardar o último valor. Ocorrências daquele caractere no resto da linha apenas fazem parte do valor. Sendo assim, um pedaço do arquivo `/etc/services` 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`, `tmpmux`, `nbp` e `compressnet`.

## 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 (`unsigned short`, 2 bytes) 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, ele deve entrar em um loop lendo as mensagens de resposta que porventura receba, até que ele fique esperando por 4 segundos sem que mais nenhuma resposta. À medida que as respostas cheguem ele pode exibi-las para o usuário, indicando que par responde (seu par IP:porto) e ao final do tempo de indicar que não há mais respostas. Note que essa temporização é simples em Python, bastando usar o comando `socket.settimeout(4)` para indicar que todo `recvfrom` deve retornar com 4 segundos (nesse caso). Basta então tratar o erro.

Os servents trocam apenas uma mensagem entre si, que tem o seguinte formato: um campo de tipo de mensagem (`unsigned short`, 2 bytes) com valor 2 (QUERY), um campo de TTL (`unsigned short`, 2 bytes), o endereço IP (4 bytes) e o número do porto (`unsigned short`, 2 bytes) do programa cliente que fez a consulta, um campo de número de sequência(`unsigned int`, 4 bytes) e o texto da chave que o cliente forneceu.

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. Essa mensagem é então repassada a todos os seus vizinhos. Em paralelo, o servent deve procurar a chave localmente e responder ao cliente, se ela 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 (baseado nos campos de IP e porto de origem, 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, ele 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 de onde a mensagem veio.

Note que cada servent altera apenas o TTL da mensagem de QUERY. Todos os demais campos permanecem com os valores preenchidos por quem 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 mensage RESPONSE diretamente para o cliente que fez a consulta (que pode ser identificado pelos campos de endereço/porto na mensagem QUERY. A mensagem de volta também é simples, contendo apenas um campo de tipo de mensagem (`unsigned short`, 2 bytes) com valor 3 (RESPONSE) e um string contendo a chave, um caractere de tabulação e o texto associado.

## Relatório e scripts

Cada aluno/dupla deve entregar junto com o código um relatório que deve conter uma descrição da arquitetura adotada para o servidor, os refinamentos das ações identificadas no mesmo, as estruturas de dados utilizadas e decisões de implementação não documentadas nesta especificação. Como sugestão, **considere incluir as seguintes seções no relatório: introdução, arquitetura, servent, cliente, discussão.** O relatório deve ser entregue em formato PDF.

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).

## Módulos recomendados

Alguns dos comentários dos trabalhos anteriores se aplicam aqui e seguem reproduzidos para lembrá-los.

A linguagem Python possui muitos módulos destinados ao desenvolvimento de aplicações em rede, tão alto nível quanto se queira. Não queremos isso. Faz parte do trabalho de vocês aprender a lidar com os principais desafios em redes de computadores. Para os exemplos a seguir, leia a variável `s` como o socket UDP criado para a comunicação. Considere os seguintes pontos na hora de escrever o código:

- Envio e recebimento de mensagens UDP. O módulo vocês já conhecem: `sockets`. Envie mensagens com `s.sendto(msg_bytes,addr)` e receba mensagens com `(msg_bytes,addr) = s.recvfrom(MAXMSGLEN)`.
- De alguma forma você precisa traduzir suas mensagens para um array de bytes, tanto no envio quanto no recebimento. As mensagens deverão ser traduzidas para um formato binário e então transformadas em um array de bytes. O módulo utilizado para fazer isso é o `struct`, que faz uma correspondência direta com structs C. Fica assim,

  `msg_bytes = struct.pack(fmt_str, *fields_list)` (ao enviar)

  `fields_list = struct.unpack(fmt_str, msg_bytes)` (ao receber)

  Veja detalhes na especificação do módulo. Mas a ideia é que `fmt_str` seja uma string que defina os tipos dos campos da mensagem (cabeçalho + corpo). Por exemplo, "!Hi" define uma representação binária para rede ('!') constituída de um `unsigned short` ('H') seguido de um `int` ('i').
- Como estamos usando UDP, o serviço é de datagramas. Quando você faz um `send`, todos os bytes chegam necessariamente ao mesmo tempo e eles não podem se juntar a uma outra mensagem, então não é preciso ter os mesmos cuidados que com TCP.

## Dicas e cuidados a serem observados

- Poste suas dúvidas no fórum específico para este TP na disciplina, para que todos vejam.
- Procure escrever seu código de maneira clara, com comentários pontuais e indentado.
- Consulte-nos antes de usar qualquer módulo ou estrutura diferente dos indicados aqui.
- 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).

Última alteração: 30 de junho de 2016