Redes de Computadores

TP3: Um sistema *peer-to-peer* de armazenamento chave-valor

Trabalho individual ou em duplas.
Data de entrega: verifique no Moodle da turma.
(Não serão aceitos trabalhos fora do prazo.)

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*, P2P). Além disso, oferece também uma oportunidade para implementar um protocolo de aplicação sobre UDP.

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

1 O problema

A ideia aqui é implementar a funcionalidade básica de um sistema de armazenamento chave-valor do tipo *peer-to-peer*, onde os programas que mantêm a informação de chaves e valores podem agir simultaneamente como clientes e servidores. Dada uma chave, o sistema deve ser capaz de localizar quais programas possuem o valor associado àquela chave. Por exemplo, de forma simplificada, na rede BitTorrent, as chaves são metadados (*hashes*) dos arquivos e os valores são o conteúdo dos arquivos compartilhados. No BitTorrent, a busca por uma chave (metadado de um arquivo) descobre quem tem o valor referente à chave (conteúdo do arquivo).

Como nos trabalhos anteriores, pode-se usar as linguagens Python e C/C++, sem módulos especiais. Vocês devem implementar um protocolo em nível de aplicação, utilizando interface de *sockets* UDP.

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 parceiros, e um programa de interface com o usuário denominado *client*, que receberá do usuário as chaves que devem ser consultadas e exibirá os resultados que forem recebidos para as consultas.

2 O arquivo de dados chave-valor

Por simplicidade, vamos definir a base de dados chave-valor como um arquivo de texto simples, onde a primeira palavra em uma linha representa a chave e o restante da linha a partir do primeiro caractere que não seja de espaçamento (*whitespace*) após a chave é o valor associado à chave. Para facilitar o aproveitamento de arquivos já existentes, considere que linhas que começam com `#` devem ser ignoradas e se uma chave aparecer mais de uma vez no arquivo, deve-se guardar apenas o último valor. Ocorrências do caractere `#` fora do início da linha fazem parte do valor. Sendo assim, um pedaço do arquivo `/etc/services` do Linux poderia ser um banco de pares chave-valor válido:

```
# 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`, com valor "`1/ddp  # Routing Table Maintenance Protocol`",
- `tmpmux`, com valor "`1/tcp  # TCP Port Service Multiplexer`",
- `nbp`, com valor "`2/ddp  # Name Binding Protocol`" e
- `compressnet`, com valor "`2/tcp  # Management Utility`".

3 O programa de cada nó da rede P2P (*servent*)

O programa da rede sobreposta (*servent*), ao ser iniciado, deverá receber (1) o número de porto onde escutará por mensagens, (2) o nome de um arquivo contendo um conjunto de chaves associadas com valores como descrito anteriormente e (3) uma lista de até 10 endereços de outros *servents* que estarão executando no sistema no formato `IP:porto`. Por exemplo:

```
./TP3node <porto-local> <banco-chave-valor> [ip1:porto1 [ip2:porto2 ...]]
```

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

A lista de endereços no formato "`ip:porto`" identifica os *servents* que devem ser adicionados como "vizinhos" do *servent* sendo executado. Se um *servent* A é vizinho de um *servent* B, ao disparar A e B o identificador de B deve ser passado como parâmetro para A e vice-versa. Cada *servent* pode ser considerado como relacionado aos seus vizinhos e a rede formada pelas relações entre eles cria uma rede sobreposta (*overlay*). Note que o protocolo de transporte usado entre os *servents* é o UDP, então não há conexões permanentes entre *servents*, apenas uma relação lógica de um *servent* conhecer o endereço do outro.

4 O programa de interface com o usuário (*client*)

O programa *client* deve ser disparado com o endereço de um *servent* da rede sobreposta que será seu ponto de contato com o sistema distribuído. Por exemplo:

```
clientTP3 <ip:porto>
```

O *client* deve então permancer em um *loop*, esperando que o usuário digite comandos, montando mensagens de consulta (descritas abaixo) relacionadas àqueles comandos e enviando-as para o *servent* ponto de contato.

Para padronizar a interface, os seguintes comandos devem ser especificados *exatamente* com a interface descrita a seguir:

- uma consulta por uma chave deve ser feita digitando-se uma linha começando com um ponto de interrogação, seguido por um ou mais espaços em branco ou tabulações, seguido por uma palavra, que identifica a chave a ser procurada;
- uma consulta pela topologia da rede é feita digitando-se uma linha que contenha apenas a letra T no início da linha;
- uma linha com apenas uma letra Q ou um fim de arquivo devem causar o término da execução do *client*.

Qualquer linha de entrada que não siga esses formatos deve ser ignorada; uma mensagem indicando que a linha não foi aceita deve ser exibida e um novo *prompt* deve ser exibido para o usuário.

4.1 O protocolo entre programas *client* e *servent*

A comunicação entre o *client* e seu *servent* ponto de contato na rede P2P também se dá através de mensagens UDP. Segundo os comandos descritos, um usuário pode realizar duas operações: procurar por uma chave ou pedir para o *servent* lhe dar uma descrição da topologia da rede.

Para procurar por uma chave, o *client* então deve enviar uma mensagem KEYREQ contendo o texto da chave em caracteres ASCII (formato da mensagem será especificado abaixo). Para pedir a topologia da rede, o *client* deve enviar uma mensagem TOPOREQ, que não tem texto associado. Mensagens KEYREQ e TOPOREQ possuem um número de sequência, que deve ser incrementado sempre que uma mensagem KEYREQ ou TOPOREQ for enviada.

Em ambos os comandos, o *client* deve ficar esperando por respostas da rede e exibi-las à medida que forem chegando. Essas respostas são sempre do tipo RESP e devem ter número de sequência igual ao enviado na requisição. Para cada resposta recebida, o cliente deve escrever na tela o texto retornado pela mensagem RESP, seguido do endereço do *servent* que enviou a resposta. Por exemplo, se uma resposta for recebida do IP de origem 10.1.2.3, do porto de origem 5151 contendo o texto "RedesRules", a linha escrita deve ser "`RedesRules 10.1.2.3:5151`".

Se o programa de interface esperar por 4 segundos e não receber nenhuma resposta, ele deve retransmitir a consulta uma única vez, com um novo número de sequência, e voltar a esperar. Se depois de mais 4 segundos nenhuma resposta for recebida, ele deve escrever a mensagem: "`Nenhuma resposta recebida`".

Se o programa receber mensagens diferentes das esperadas ou com número de sequência diferente do esperado, ele deve escrever uma mensagem: "`Mensagem incorreta recebida de ip:porto`", onde `ip:porto` deve ser substituído pelo endereço de origem.

Se o *client* receber uma primeira resposta, ele não sabe quantas outras respostas pode receber, pois vários nós podem ter respostas a enviar. Sendo assim, ele deve entrar em um *loop* lendo as mensagens de resposta que porventura receba e exibindo-as na tela. Depois que tiver recebido pelo menos uma mensagem, se o *client* ficar esperando por 4 segundos sem que nenhuma outra mensagem chegue, ele deve dar a consulta por encerrada e voltar ao *loop* para ler outro comando da entrada padrão. Note que as temporizações do *client* podem ser feitas simplesmente associando uma temporização com a função `recvfrom`.

5 O protocolo para propagação de consultas entre *servents*

Um *servent* que receba uma mensagem KEYREQ ou TOPOREQ deve iniciar o protocolo de alagamento para divulgar a consulta pela rede sobreposta P2P como explicado a seguir. Esse protocolo segue um princípio semelhante ao alagamento confiável usado em OSPF. Ao receber uma consulta de um programa *client*, um *servent* gera uma mensagem KEYFLOOD ou TOPOFLOOD, respectivamente, que será disseminada pela rede. Ambas as mensagens carregam o endereço do *client* que iniciou a consulta e o número de sequência usado por ele. As mensagens KEYFLOOD e TOPOFLOOD carregam um campo de tempo de vida (`TTL`), que especifica o quanto elas ainda precisam ser propagadas durante o alagamento. O *servent* que recebe a requisição do *client* e cria a mensagem de alagamento pela primeira vez deve iniciar esse `TTL` com o valor 3.

Todo *servent* que recebe uma requisição por alagamento deve primeiro certificar-se de que a mensagem não foi recebida anteriormente, mantendo um dicionário de mensagens já vistas. Requisições de alagamento podem ser identificadas unicamente usando o endereço do *client* e seu número de sequência. Além disso, o *servent* deve decrementar o valor do `TTL` e, se o valor resultante for maior que zero, ele deve repassar a mensagem para seus vizinhos, exceto aquele de onde a mensagem foi recebida. Se o `TTL` for zero, a mensagem não deve ser retransmitida. Como o valor inicial do `TTL` é 3, se tivermos cinco *servents* conectados em sequência e o primeiro da cadeia receber uma mensagem de um *client*, a consulta gerada atingirá apenas 3 *servents* depois dele. Isto é, o quinto *servent* na cadeia não receberia a consulta. Em programas reais, um `TTL` maior seria recomendável, mas neste trabalho vamos mantê-lo em 3 para simplificar. Não use um `TTL` inicial maior do que 3.

As mensagens KEYFLOOD e TOPOFLOOD também possuem um campo `info` com semântica específica. O campo `info` da mensagem KEYFLOOD contém o string exato da chave fornecida pelo usuário na mensagem KEYREQ. Todo *servent* que recebe uma mensagem KEYFLOOD (e o nó que

cria a primeira mensagem KEYFLOOD) deve consultar seu dicionário de chaves para verificar se ele contém a chave procurada. Em caso positivo, o *servent* deve enviar, diretamente para o *client* que iniciou a consulta, uma mensagem RESP com o número de sequência fornecido na consulta e com o campo `info` preenchido com o valor associado àquela chave no dicionário daquele *servent*.

O campo `info` da mensagem TOPOFLOOD deve ser inicializado pelo *servent* que recebeu requisição do *client* e montou a mensagem TOPOFLOOD com uma string com seu próprio endereço no formato `ip:porto` (isto é, o endereço do *servent*). O campo `info` recebido em uma mensagem TOPOFLOOD deve ser alterado por um `servent` antes de ser enviada a seus vizinhos. Cada `servent` deve acrescentar um espaço e seu próprio endereço no formato `ip:porto` ao final do campo `info`. Depois de fazer isso, o `servent` deve enviar diretamente para o programa de interface que iniciou a consulta, uma mensagem RESP com o número de sequência fornecido na consulta e com o campo `info` preenchido com o valor resultante. Note que dessa forma, cada nó vai enviar de volta para o programa *client* que iniciou a consulta um string com a sequência dos endereços dos `servents` por onde a consulta passou até chegar àquele `servent`.

6 Formato das mensagens

As diversas mensagens são sempre diferenciadas pelo valor do primeiro campo, `tipo`. Os seguintes tipos são definidos:

	KEYREQ	TOPOREQ	KEYFLOOD	TOPOFLOOD	RESP
Valor de `tipo`	5	6	7	8	9

A seguir, os campos `chave` e `info` são strings (vetores de caracteres) de no máximo 400 caracteres. Os strings transmitidos pela rede não devem ser terminados com o caractere nulo `\0` (o final do string é indicado pelo final da mensagem UDP recebida do *socket* — isso é particularmente importante para as implementações em C/C++). `IP_ORIG` é a representação binária do endereço IP do programa de interface (4 bytes). Os demais campos são inteiros e devem ser representados na rede em *network byte order*. O tamanho de cada campo em bytes é indicado nas figuras abaixo. Por exemplo, `tipo` tem dois bytes, enquanto `nseq` tem 4 bytes.

KEYREQ

```
+---- 2 ---+--- 4 --+----------\\-------------------+
| TIPO = 5 |  NSEQ  | CHAVE (até 400 characteres)   |
+----------+--------+----------\\-------------------+
```

TOPOREQ

```
+---- 2 ---+--- 4 --+
| TIPO = 6 |  NSEQ  |
+----------+--------+
```

KEYFLOOD, TOPOFLOOD (valor do tipo indica o tipo de mensagem)

```
+---- 2 ------+-- 2 --+--- 4 --+--- 4 ---+----- 2 ----+----------\\-------------------+
| TIPO = 7, 8 |  TTL   |  NSEQ  | IP_ORIG | PORTO_ORIG | INFO  (até 400 characteres)   |
+-------------+--------+--------+---------+------------+----------\\-------------------+
```

RESP

```
+---- 2 ---+--- 4 --+----------\\-------------------+
| TIPO = 9 |  NSEQ  | CHAVE (até 400 characteres)   |
+----------+--------+----------\\-------------------+
```

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, o detalhamento das ações realizadas pelo 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*, programa de interface, 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 de interface e servidor já foi definida: o que cada um deve receber como parâmetros de linha de comando, o formato das linhas de comando do usuário para o programa de interface e que mensagens devem ser obrigatoriamente exibidas.

Uso de UDP

Como discutido em sala, neste trabalho usaremos UDP, o que muda o comportamento de envio e recebimento de mensagens. Com UDP, vocês precisam obrigatoriamente montar toda a mensagem em um bloco contíguo de memória antes de chamar a função de envio (por exemplo, `sendto`). Não é possível montar o cabeçalho binário, fazer um `send` dele e depois fazer um `send` do string, pois cada `send` gera um datagrama separado e o protocolo assume que uma mensagem chega em um único datagrama UDP. Toda a mensagem deve estar montada como um único vetor de bytes na memória e a chamada tem que indicar exatamente o número de bytes que se deseja enviar. Note que a quantidade de bytes enviados pelas mensagens com os campos `chave` e `info` é *variável*: você deve considerar o tamanho do string. (Não envie um vetor de 400 bytes em todas as mensagens.)

Como a mensagem UDP tem tamanho bem definido (ao contrário das mensagens sobre TCP) não é necessário enviar o tamanho das strings nas mensagens: ele é dado pelo tamanho da mensagem UDP, descontados os outros campos. Entretanto, é importante, ao fazer a chamada da função `sendto` indicar o número de bytes correto a enviar.

Ao receber, da mesma forma, vocês devem receber um vetor de bytes com a mensagem completa. Não é possível (nem necessário) fazer um primeiro `recv` para ler um campo de tamanho, como era feito com TCP, para depois ler o resto. Seu `recvfrom` deve prever um tamanho suficiente para guardar a maior mensagem possível na comunicação. Para esse fim, vamos considerar que o string de valor associado a qualquer chave poderá ter no máximo 400 caracteres, assim como o string que vai ser passado de um par para outro no caso de consultas pela topologia. Não faremos nenhum teste que extrapole esse valor.

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 considerados parte da biblioteca padrão da linguagem.
- 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 por alagamento, por exemplo.

Última alteração: 13 de Novembro de 2017