Universidade Federal de Minas Gerais
DCC023: Redes de Computadores
TP3 – Um sistema *peer-to-peer* de armazenamento chave-valor

Última modificação: 28 de Maio de 2019

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 o TCP.

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* TCP.

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 conexões, (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 já foram disparados e estão 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* TCP no porto indicado para receber conexões de outros *servents* e *clients* e proceder ao estabelecimento de conexões com os outros *servents* identificados nos parâmetros recebidosa da linha de comando.

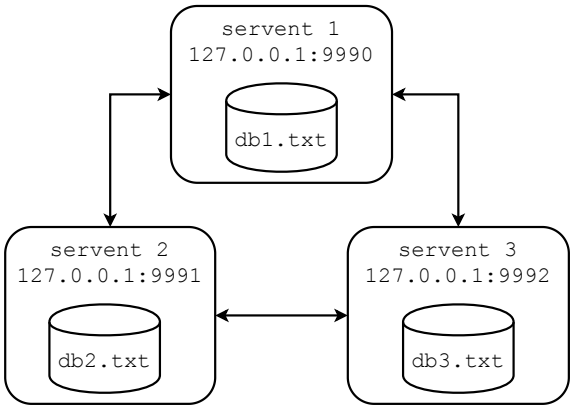
A lista de endereços no formato "`ip:porto`" identifica os *servents* que devem ser "vizinhos" do *servent* a ser disparado. Cada *servent* pode ser considerado como relacionado aos seus vizinhos e a rede formada pelas relações entre eles cria uma rede sobreposta (*overlay*). Note que o protocolo de transporte usado entre os *servents* é o TCP, e logo deverão existir conexões permanentes entre os *servents*. Assim, se um *servent* A é indicado como "vizinho" de um *servent* B, ao disparar A é necessário que B já esteja ativo para que a conexão possa ser estabelecida. Logo após ser disparado, um *servent* deve se conectar a cada um dos *servents* indicados na lista e informá-los que é "vizinho" deles para que eles também conheçam a vizinhança deles. A maneira como isso deve ser feito será descrito mais a frente.

3.1 Exemplo

A sequência de comandos

```
./TP3node 9990 db1.txt &
./TP3node 9991 db2.txt 127.0.0.1:9990 &
./TP3node 9992 db3.txt 127.0.0.1:9990 127.0.0.1:9991 &
```

deve produzir uma rede sobreposta com 3 *servents* (exibida na figura abaixo), onde existem conexões fixas entre cada par de nós.



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

O programa *client* deve ser disparado com o (i) o número de porto onde escutará por mensagens enviadas por *servents* e (ii) endereço de um *servent* da rede sobreposta que será seu ponto de contato com o sistema distribuído. O *client* deve ser disparado da seguinte forma:

```
./TP3client <porto-local> <ip:porto>
```

Ao ser disparado, o *client* deve abrir uma conexão fixa com o *servent* indicado e permanecer em um *loop*, esperando que o usuário digite comandos, montando mensagens de consulta (descritas abaixo) relacionadas àqueles comandos e enviando-as para o *servent* que é seu 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. Por exemplo, o comando "`? rtmp`"requisita uma consulta pela chave `rtmp`.
- 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 "`Comando desconhecido`" deve ser exibida e um novo *prompt* deve ser exibido para o usuário.

5 O protocolo de conexão entre programas

Como neste trabalho usaremos o TCP, as comunicações ocorrerão através de conexões entre pares de nós. Nós *servents* podem receber requisições de conexão de outros nós *servents*, para a formação da rede sobreposta, e de nós *clients*, para a execução de consultas. Para que um *servent* seja capaz de determinar os tipos dos nós conectados a ele, faz-se necessário a introdução de uma forma de identificação de nós.

A identificação adotada funciona da forma descrita a seguir. Quando um programa A se conecta a um *servent* B (através da chamada da função `connect`), ele deve enviar uma mensagem ID para B se identificando como *servent* ou *client*. Isso é feito enviando um valor igual a zero na mensagem se for um *servent*, ou diferente de zero, se for *client*. Nesse último caso, o valor deve ser o número do porto onde o *client* receberá as respostas às consultas que fizer (formato da mensagem será especificado a seguir).

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

Depois de estabelecer a conexão com o cliente e enviar a mensagem ID, o cliente passará a esperar por comandos o teclado, como descrito anteriormente.

Para procurar por uma chave, o *client* então deve enviar uma mensagem KEYREQ contendo o texto da chave em caracteres ASCII. 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 começar com zero e deve ser incrementado sempre que uma mensagem KEYREQ ou TOPOREQ for enviada.

Em ambos os comandos, o *client* deve esperar por respostas da rede P2P. Isso é feito esperando por conexões (utilizando a função `accept`) feitas por nós da rede que tenham respostas para uma dada consulta. O Cliente deve então aceitar a conexão, receber a mensagem de respsotas e exibi-la na sua saída. Essas conexões devem ser curtas, devendo ser encerradas após o recebimento completo de uma única mensagem.

As respostas enviadas para os comandos KEYREQ e TOPOREQ 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 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. Ele deve continuar esperando outras respostas normalmente.

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* recebendo as conexões que surjam, lendo as mensagens de resposta enviadas 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 conexão 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 ao *socket* que espera por pedidos de conexão.

7 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, o número do porto informado pelo *client* na sua mensagem ID e o número de sequência usado por ele na requisição. 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* (IP:porto) e seu número de sequência. Se uma mensagem recebida já foi vista antes, ela deve ser simplesmente descartada. Caso contrário, 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 abrir uma conexão diretamente para o *client* que iniciou a consulta, no endereço indicado nas mensagens de alagamento, enviar 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*, e fechar a conexão.

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 abrir uma conexão diretamente para o programa de interface que iniciou a consulta, enviar uma mensagem RESP com o número de sequência fornecido na consulta e com o campo `valor` preenchido com o mesmo valor final no campo `info` (da mensagem TOPOFLOOD) que será enviada para os vizinhos e fechar a conexão. 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`.

8 Formato das mensagens

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

	ID	KEYREQ	TOPOREQ	KEYFLOOD	TOPOFLOOD	RESP
Valor de `tipo`	4	5	6	7	8	9

A seguir, os campos `chave`, `info` e `valor` são strings (vetores de caracteres) de no máximo 400 caracteres. Elas devem ser sempre representadas em ASCII (em Python, use `encode/decode`). Como mensagens enviadas por TCP não têm tamanho bem definido (ao contrário das mensagens enviadas por UDP), o campo `tamanho` é utilizado para enviar o tamanho das strings nas mensagens que têm esse tipo de campo. Dessa forma, os strings transmitidos pela rede não devem ser terminados com o caractere nulo `\0` — 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.

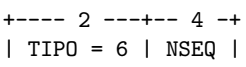
ID

```
+---- 2 ---+--- 2 --------------------------+
| TIPO = 4 |  PORTO (ou zero se for servent) |
+----------+---------------------------------+
```

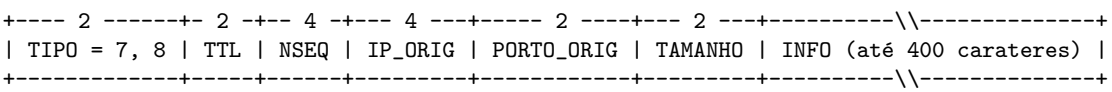
KEYREQ

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

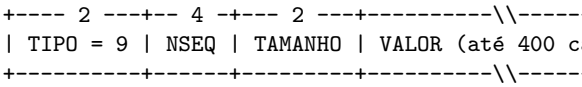
TOPOREQ



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



RESP



9 Relatório e programas

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 (*client*), discussão. O relatório deve ser entregue em formato PDF.

Caso desenvolva seu trabalho utilizando a linguagem Python, deixe claro no relatório ou no código qual a versão utilizada — recomendamos fortemente que adote Python 3. Caso utilize C/C++, submeta também um Makefile para a compilação do código e não inclua executáveis.

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. Submissões onde os programas não seguem as especificações de parâmetros e nomes, Makefiles não funcionando ou com arquivos essenciais faltando não serão corrigidos.

Dicas e cuidados a serem observados

- *Servents* deverão lidar com múltiplas conexões. Uma forma simples de fazer isso é utilizando a função `select`. Essa função permite o monitoramento de múltiplos descritores de arquivos, esperando até que um ou mais deles estejam preparado para fazer operações de I/O sem bloquear. Logo, é possível utilizar a função `select` para monitorar múltiplos *sockets* e detectar, por exemplo, quando um ou mais deles receberam dados.

  O programa deve montar um conjunto de descritores indicando todos os *sockets* dos quais espera uma mensagem (inclusive o socket usado para fazer o `accept`, que deve ser um só). A função permite indicar *sockets* também onde se deseja escrever ou onde se procura por alguma exceção, mas esses dois conjuntos não interessam neste trabalho. Também não é necessário usar o temporizador que pode ser associado ao `select` (o único temporizador será no `accept` do cliente). Uma vez chamada, a função `select` bloqueia até que algo ocorra nos `sockets` que foram indicados na chamada. Ao retornar, o conjunto indica quais `sockets` têm operações pendentes.

- O guia de programação em rede do Beej (`http://beej.us/guide/bgnet/`) e o Python Module of the Week (`https://pymotw.com/2/select/`) têm bons exemplos de como organizar um servidor com `select`. Mais detalhes sobre essa função podem ser encontrados nos links `http://man7.org/linux/man-pages/man2/select.2.html` (C/C++) e `https://docs.python.org/3/library/select.html#select.select` (Python 3).

- Sobre o uso de timouts associados à função `accept`: `https://stackoverflow.com/questions/7354476/python-socket-object-accept-time-out`.
- Lembre-se que a transmissão de dados sobre o TCP acontece através de um fluxo de bytes sem delimitação, e não por datagramas (mensagens) como é no UDP.
- 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.