Redes de Computadores

Exercício Prático – Programação com Sockets

Prof. Vitor Barbosa Souza
vitor.souza@ufv.br
DPI – Departamento de Informática
UFV – Universidade Federal de Viçosa

Alguns softwares que permitem chamadas de áudio/vídeo, como o Whatsapp e Zoom, utilizam uma arquitetura cliente servidor, de modo que todos os dados passam por um servidor central responsável por fazer o encaminhamento para o destinatário. Uma outra possibilidade, é se utilizar uma arquitetura P2P para que pacotes de dados sejam trocados diretamente entre dois usuários. Essa abordagem é utilizada pelo Skype para chamadas de áudio e vídeo. Nesse caso, um servidor central pode ser utilizado apenas para se conseguir informações sobre o estado dos usuários da rede, por exemplo, saber quem está ativo e como estabelecer uma conexão.

Neste exercício você deverá escrever um programa em Python, C++ ou Java baseado na arquitetura **P2P** descrita acima para a comunicação entre pares. Por simplicidade, toda a comunicação trocada entre pares será no formato texto, como em um chat onde os dois lados devem estar online ao mesmo tempo. Para se obter as informações necessárias para estabelecimento da comunicação, um segundo programa será criado para funcionar como um servidor centralizado.

Para que as partes possam se comunicar corretamente, um protocolo de aplicação é proposto abaixo, ou seja, **não** serão utilizados protocolos de aplicação existentes como HTTP ou outros. Você deverá implementar o protocolo de aplicação descrito a seguir utilizando *sockets*.

Especificação do protocolo de aplicação

O servidor central estará constantemente escutando a porta 10000 para que cada cliente possa estabelecer uma conexão TCP para enviar/receber informações para/do o servidor. Ao mesmo tempo, cada cliente deverá escutar alguma outra porta (não especificada pelo protocolo) para que outro cliente possa estabelecer uma comunicação P2P diretamente com ele. Para que um cliente (nesse caso funcionando um par, ou *peer*, de uma comunicação P2P) possa receber conexões de outro, ele deverá utilizar uma mensagem própria para informar ao servidor central qual porta estará aberta para recebimento de conexões. Assim, um par que deseja se conectar a outro par, deve, primeiro, perguntar ao servidor qual a porta a ser utilizada (além do endereço IP). A comunicação entre pares também deve ser realizada utilizando conexões TCP, assim, o protocolo de aplicação é simplificado e a aplicação não precisa verificar se cada mensagem é entregue.

O protocolo da aplicação deverá suportar as seguintes instruções:

Instrução/Formato	Descrição
Mensagens enviadas dos clientes para o servidor central	
`USER <nome>:<porta>`	Primeira mensagem enviada pela aplicação cliente ao servidor central para informar o <nome> de usuário e a <porta> a qual ele estará escutando. O servidor deverá manter a relação de usuários ativos no momento. O nome não deve conter o caractere ":" e a porta deve ser um número inteiro. Ex: `USER Fulano:20000`
`LIST`	Enviada para ao servidor quando o usuário solicita a lista dos demais clientes ativos.
`KEEP`	Mensagem *keepalive*. Essa mensagem é enviada a cada 5 segundos ao servidor por cada cliente conectado. O objetivo é informar que o cliente ainda está ativo. Se essa mensagem deixar de ser enviada por qualquer motivo (como fechamento inesperado do cliente), o servidor deverá esperar 3 vezes o tempo do keepalive (15 segundos) antes de forçar o fim da conexão e excluir o cliente da relação de clientes ativos.
`ADDR <nome>`	Requisita ao servidor o endereço IP e porta a ser usada para se conectar ao usuário <nome> Ex: `ADDR Fulano`
Respostas enviadas do servidor central às solicitações dos clientes	
`LIST <nome1>:<nome2>`	Reposta ao comando LIST. Deve listar o nome de usuários ativos, separados por ":"
`ADDR <ip>:<porta>`	Resposta ao comando ADDR, informando o endereço <ip> do usuário consultado e a <porta> em que o mesmo espera receber um pedido de conexão. Essa resposta só é enviada se o usuário procurado se encontra na lista de usuários ativos do servidor.
Mensagens trocadas entre os dois peers em uma conexão ativa. Nesse caso, o peer que inicia a conexão (comando connect) faz o papel de cliente, e o peer que aguardava (comando accept) essa conexão, faz o papel de servidor.	
`USER <nome>`	Primeira mensagem enviada pelo cliente (que inicia conexão TCP) para se identificar ao servidor (que estava aguardando).
`DISC`	Tendo uma conexão estabelecida, qualquer um dos pares pode decidir encerrar a mesma. No entanto, antes de fechar a conexão (comando *close()*), o nó envia a mensagem DISC. Assim, o outro lado pode encerrar a conexão de forma simétrica ao receber o DISC.

Perceba que o protocolo não especifica instruções para transferência de mensagens relacionadas à conversa dos usuários finais. Dessa forma, qualquer mensagem que não atenda aos padrões acima, são entendidas como parte da conversa entre os *peers*. Caso não exista uma conexão estabelecida com um *peer*, esse erro deve ser informado na interface do programa. Lembre-se que os clientes sempre terão uma conexão ativa com o servidor central, mas não faz sentido enviar essas mensagens não especificadas pelo protocolo ao servidor.

Perceba também que o protocolo especifica apenas as mensagens trocadas entre os nós através da rede. Os comandos digitados pelo usuário no terminal ficam a cargo das preferências do programador da aplicação. Por exemplo, o usuário não precisa saber da existência dos comandos ADDR, USER, DISC, etc. Ao invés disso, o programa poderia mostrar comandos úteis ao usuário em uma mensagem de boas-vindas. Por exemplo:

```
Para iniciar uma conversa digite: /chat <nome_de_usuário_ativo>
```

Assim, internamente o comando `/chat` seria traduzido em uma consulta usando ADDR, seguida do estabelecimento de uma conexão TCP e envio da identificação do usuário, com o comando USER. Da mesma forma, a resposta do comando ADDR é utilizada apenas internamente para estabelecimento da conexão, não sendo relevante para o usuário da aplicação dados como endereço IP e porta.

Veja no quadro mostrado na próxima página um exemplo de algumas ações sendo executadas em sequência. Para facilitar a visualização e compreensão, diferentes cores são utilizadas:

- os textos em **verde** descrevem uma ação sendo realizada
- os textos na cor **preta** são exemplos de mensagens impressas na tela (essas mensagens são independentes do protocolo, ou seja, a aplicação está livre para implementar essas mensagens como quiser)
- os textos em **vermelho** representam as mensagens do **protocolo de aplicação** descrito acima, ou seja, são as mensagens trocadas entre entidades diferentes e devem ser exatamente como especificadas
- os textos em **azul** representam um texto digitado pelo usuário da aplicação

Outras considerações e dicas

- Utilizando a tabela exemplo da próxima página, perceba que cada *peer* escuta uma porta distinta. Isso é desejável já que é possível que mais de um cliente execute em uma mesma máquina (principalmente para teste). Assim, ao invés de fixar a porta, utilize o comando *bind* do *socket* para verificar se uma porta está disponível para ser utilizada. Se não estiver, o comando *bind* lança a exceção do tipo *OSError*.

- O encerramento da conexão usando a função *close* do socket exige que os dois lados encerrem a conexão (veja novamente a descrição da instrução DISC do nosso protocolo). No entanto, em algumas situações pode ser necessário forçar o encerramento da conexão sem esperar que o outro lado faça um *close*, por exemplo quando um servidor desconecta o cliente que não envia o KEEP. Nesse caso, pode ser usado o `shutdown(socket.SHUT_RDWR)` na conexão que se desejar derrubar.

- O exemplo da tabela a seguir mostra o caso de uma conexão entre Bruno e Camila. É importante notar que ambos continuam aguardando novas conexões (da mesma forma que o servidor central pode receber várias conexões em paralelo) existindo a possibilidade de Ana iniciar uma conversa com um deles, por exemplo com Camila. Nesse caso, use a criatividade para tratar esses casos. Como Camila irá informar se a mensagem digitada em seu terminal deverá ser enviada para Ana ou para Bruno? Você pode simplificar e considerar, por exemplo, que a comunicação é feita prioritariamente com o *peer* com conexão mais antiga (ou mais nova). Ou então inserir alguma forma de especificar isso na linha de comando. O importante é que isso é uma decisão que afeta apenas a interface com o usuário e nenhuma das opções irá alterar o protocolo em si.

- Ao terminar uma conexão, se o outro lado está bloqueado na função *recv*, a função desbloqueia, retornando texto vazio. Por outro lado, o *accept* (para aguardar uma conexão) tem comportamento diferente dependendo da versão da biblioteca. No exemplo mostrado, ao digitar `/exit`, o socket responsável por receber conexões dos *peers* pode ser fechado mas a thread bloqueada no *accept* poderá não finalizar. Ao invés de utilizar uma thread para leitura do teclado e outra para tratar recepção das conexões, considere o uso do comando select do python para resolver esse problema.

- O TCP é um protocolo de fluxo de bytes. Isso significa que no envio de mensagens consecutivas em uma mesma conexão, duas ou mais mensagens podem ser vistas pelo receptor sem uma clara separação entre as mesmas. Por exemplo, se a thread que envia o KEEP de forma recorrente faz o envio ao mesmo tempo que é enviada a instrução LIST, a mensagem recebida pelo destinatário poderá ser KEEPLIST ou LISTKEEP. Para controlar esse tipo de problema, o nosso protocolo deverá acrescentar os caracteres especiais `\r\n` ao final de cada instrução enviada. Assim, o receptor poderá processar as instruções de forma individual e sem ambiguidades.

- Se o seu código implementar a troca de mensagens do protocolo de forma correta, será possível fazer a comunicação entre clientes diferentes (como os desenvolvidos por seus colegas) sem erros!

O trabalho pode ser feito individualmente ou em dupla. A entrega deve ser realizada por apenas um dos membros da dupla. Colocar o nome dos dois no cabeçalho do código.

Servidor central	Peer	Peer	Peer
<aguardando conexão>	–	–	–
<conexão TCP realizada> *<cadastro usuário ativo>*	Nome de usuário: Ana *<inicia conexão TCP serv.>* USER Ana:20000 *<aguardando conexão>*	–	–
<conexão TCP realizada> *<cadastro usuário ativo>*		Nome de usuário: Bruno *<inicia conexão TCP serv.>* USER Bruno:21500 *<aguardando conexão>*	–
LIST Ana		/list LIST Ativos (além de você): Ana	–
<conexão TCP realizada> *<cadastro usuário ativo>*			Nome de usuário: Camila *<inicia conexão TCP serv.>* USER Camila:45000 *<aguardando conexão>*
LIST Ana:Bruno			/list LIST Ativos (além de você): Ana, Bruno
ADDR 200.235.90.80:21500		*<conexão TCP realizada>* Conectado a Camila	/chat Bruno ADDR Bruno *<inicia conexão TCP peer>* USER Camila
		Oi Bruno! Tudo bem, Camila?	Oi Bruno! Tudo bem, Camila?
		Tchau /bye DISC *<encerrar conexão peer>*	Tchau *<encerrar conexão peer>*
<encerrar conexão>	Alguém aí? Nenhuma conexão P2P ativa /exit *<encerrar conexão serv.>*		
<encerrar conexão>		/exit *<encerrar conexão serv.>*	
<encerrar conexão>			/exit *<encerrar conexão serv.>*