CAMADA DE APLICAÇÃO

1 Introdução

NESTA PRÁTICA, vamos implementar um protocolo simples de camada de aplicação. A aplicação é uma sala de bate-bapo (*chat*) com uma arquitetura tradicional cliente-servidor. Os clientes poderão enviar mensagens de uma linha. O servidor replicará as mensagens de forma que possam ser lidas por todos os clientes conectados.

Após dar uma visão geral do protocolo proposto, este roteiro te guiará em passos para que você construa aos poucos uma implementação completa.

2 Protocolo

2.1 Estado do cliente

Um cliente conectado ao servidor pode estar em algum dos seguintes estados:

1. *Sem apelido definido*: o cliente acabou de conectar.
2. *Com apelido*: o cliente escolheu um apelido desde que conectou.

2.2 Mensagens do cliente ao servidor

Os mensagens enviadas do cliente para o servidor são sempre de uma única linha terminada por `'\n'`. A interpretação da mensagem deve ser feita da seguinte forma:

1. Caso a mensagem comece com `"/nick "`, ela é tratada como um comando de trocar apelido. Ao receber uma mensagem `"/nick apelido"`, o servidor valida o apelido: ele não pode conter os caracteres `':'` nem `' '`, nem estar em uso por outro cliente. Se o apelido for inválido, o servidor volta a mensagem `"/error"`. Se for válido, o servidor troca o apelido do cliente para o solicitado e muda o cliente para o estado *Com apelido*.

   Sempre que algum cliente adquire com sucesso um apelido, o servidor informa esse fato a todos os clientes conectados (inclusive ao que solicitou o apelido). Caso o estado anterior do cliente seja *Sem apelido definido*, o servidor envia a mensagem `"/joined apelido"`. Caso seja *Com apelido*, o servidor envia `"/renamed apelido_antigo apelido_novo"`.

2. Caso contrário, a mensagem é tratada como texto a ser enviado. Se o cliente estiver no estado *Sem apelido definido*, o servidor volta a mensagem `"/error"`. Se estiver no estado *Com apelido*, o servidor replica a mensagem no formato `"apelido: mensagem"` para todos os clientes conectados (inclusive aquele que enviou a mensagem).

Se um cliente fechar a conexão com o servidor e estiver no estado *Com apelido*, o servidor deve enviar a mensagem `"/quit apelido"` para todos os outros clientes.

2.3 Mensagens do servidor ao cliente

As mensagens do servidor ao cliente podem ser dos cinco tipos a seguir, e são enviadas nas situações descritas na subseção anterior. Elas também são sempre formadas por uma única linha terminada por `'\n'`.

1. `"/error"`: significa que a operação anterior solicitada pelo cliente causou um erro.
2. `"/joined apelido"`: significa que uma nova pessoa entrou na conversa.
3. `"/renamed apelido_antigo apelido_novo"`: significa que alguém que já estava na conversa mudou de apelido.
4. `"apelido: mensagem"`: significa que alguém enviou uma mensagem no bate-papo.
5. `"/quit apelido"`: significa que alguém saiu do bate-papo.

3 Cliente

Para testar o servidor, você pode usar `telnet` ou `nc`. Não há a necessidade de implementar um cliente específico para ele. Para conectar ao seu servidor, você poderá chamar, por exemplo:

```
nc localhost 7000
```

4 Servidor

Implemente o seu servidor no arquivo `servidor`. Os exemplos que daremos são em Python, mas você pode usar a linguagem que quiser. Se você usar uma linguagem compilada, como C ou Rust, crie um shell script chamado `compilar` contendo os comandos necessários para compilar seu código. Se precisar de algum compilador que não estiver instalado no servidor, entre em contato com o professor para verificar a possibilidade de instalá-lo.

Para testar seu código, execute `./autograde.py`.

Passo 1

Para utilizar os serviços de rede do sistema operacional, o primeiro passo é criar um *socket*.

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

A opção `AF_INET` indica que estamos usando IPv4. A opção `SOCK_STREAM` indica que queremos operar em cima de um protocolo que se comporte como um fluxo contínuo de dados, que no caso da suíte de protocolos da internet é o TCP.

Em seguida, vamos habilitar a opção `SO_REUSEADDR`, que indica que queremos que nosso programa tome controle da porta mesmo que ela esteja em uso por conexões pendentes de serem finalizadas. Essa opção geralmente não é usada em produção, mas no nosso caso será importante para permitir que executemos um teste logo em seguida do outro.

```
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Depois, precisamos atrelar o *socket* a uma porta e mandá-lo escutar.

```
s.bind(('', 7000))
s.listen(5)
```

Agora finalmente conseguimos mandar o *socket* aceitar um pedido de conexão.

```
conexao, endereco = s.accept()
```

Enquanto não aparecer nenhum cliente pedindo para conectar, o programa fica parado na linha acima. Assim que aparecer alguém, ele aceita a conexão. A variável `conexao` receberá um novo *socket*, que permite conversar com aquela conexão específica. A variável `endereco` receberá uma tupla que permite identificar, se necessário, o endereço IP e a porta da outra ponta da conexão (o cliente).

Uma vez estabelecida a conexão, é possível receber e enviar dados. Para receber dados, utilize o método `recv`:

```
conexao.recv(2048)
```

O argumento é o número **máximo** de bytes que você quer receber de uma vez. Pode ser que venha menos bytes que o que você pediu, mas nunca virão bytes a mais.

Para enviar dados, utilize o método `send`:

```
conexao.send(b"/error\n")
```

Utilize as operações acima para implementar um programa que aceita a conexão de um cliente na porta 7000 e responde toda linha que o cliente enviar com a mensagem `"/error"`. Essa implementação mínima deve ser suficiente para passar no primeiro teste.

Passo 2

Um erro muito comum de pessoas que estão começando a trabalhar com *sockets* é acreditar que uma mensagem sempre vai ser transportada de uma só vez de uma ponta até a outra. Essa situação ideal pode até persistir enquanto você estiver testando seu programa localmente, mas quando ele estiver funcionando em uma rede real, duas situações eventualmente acontecerão:

1. Uma mensagem do tipo `"linha\n"` pode ser quebrada em várias partes. Por exemplo, podemos receber primeiro `"lin"`, depois `"h"` e depois `"a\n"`.
2. Duas ou mais mensagens podem ser recebidas de uma só vez. Por exemplo, podemos receber `"linha 1\nlinha 2\nlinha 3\n"`.

As duas coisas também podem acontecer ao mesmo tempo. Podemos receber, por exemplo, algo do tipo `"a 1\nlinha 2\nli"`.

Adapte seu servidor para tratar situações similares às descritas acima. É interessante escrever uma função auxiliar (chamada, por exemplo, `recvline`) para receber uma linha completa. A forma mais fácil de implementar é receber um caractere por vez (`conexao.recv(1)`) e ir acumulando num *buffer*, mas você pode tentar fazer um gerenciamento mais sofisticado se quiser. Feito isso, você deve conseguir passar no segundo teste.

Passo 3

Como saber que uma pessoa fechou a conexão? Nesse caso o método `recv` retorna uma *string* de bytes vazia. O ideal, nesse caso, é que sua ponta também encerre a conexão, chamando `conexao.close()`.

Implemente agora todo o protocolo, ou seja, trate e gere corretamente todos os tipos de mensagens descritos na Seção 2. Por enquanto, você não precisa lidar com mais de um cliente — assuma que sempre haverá uma única pessoa conectada ao servidor.

Passo 4

Agora temos um servidor praticamente completo, mas ele só aceita a conexão de um cliente por vez. Como fazer para conseguir trabalhar com múltiplas conexões ao mesmo tempo? Existem algumas formas diferentes, mas uma das mais clássicas é usar a chamada de sistema `select`.

```
import select
leitura_pendente, _, _ = select.select(lista, [], [])
```

O exemplo acima utiliza apenas os recursos essenciais do `select`. Passamos como argumento uma `lista` de *sockets*, que deve incluir o *socket* principal do servidor (aquele que chamamos de `s` no exemplo anterior), além de todos os *sockets* que correspondam a conexões abertas no momento. Todos os *sockets* que possuírem dados disponíveis para leitura serão retornados em `leitura_pendente`.

Sempre que o *socket* principal for incluído em `leitura_pendente`, significa que há alguma nova conexão pendente, que pode ser aceita com o método `accept`. O novo *socket* retornado pelo `accept` deve, então, ser adicionado à `lista`.

Sempre que algum *socket* de conexão for incluído em `leitura_pendente`, significa que existem dados pendentes para leitura, ou seja, significa que é possível chamar o método `recv` sem pausar (bloquear) o processo.

Este é apenas um exemplo bastante simplificado. Para exemplos mais completos, recomendamos a leitura de https://pymotw.com/3/select/.

✎ Adapte o seu servidor para funcionar com diversas conexões simultâneas.

Passo 5

A função `recvline` do Passo 2 é um excelente primeiro passo para conseguir trabalhar com mensagens que chegam quebradas. O problema é se um cliente começar a enviar uma linha e nunca terminar de enviar. Isso pode acontecer acidentalmente (a conexão da outra ponta pode ter caído repentinamente) ou de forma proposital (um adversário tentando causar uma negação de serviço). Nesse caso, o seu servidor vai ficar esperando o `'\n'` para sempre, e ele nunca vai chegar!

Uma das formas de abordar esse problema é, em vez de manter um *buffer* numa variável local e fazer um *loop*, ler no máximo máximo um caractere por vez depois do *socket* ser retornado pelo `select` e armazenar esse caractere no *buffer* correspondente àquele cliente. Para manter um *buffer* por cliente, é possível usar um dicionário indexado pelo *socket* do cliente ou outra estrutura de dados similar. Apenas ao receber um `'\n'` o servidor pode começar a tratar a mensagem.

✎ Assegure-se que o seu servidor **não** trave ao receber uma mensagem incompleta (sem a terminação `'\n'`).

Reflexão

O objetivo desta prática foi construir um protótipo bastante preliminar de um protocolo de camada de aplicação. Para um software em produção, haveria uma série de problemas de ordem prática e questões de segurança que não consideramos. Reflita a respeito dessas questões e inclua junto com seus códigos um pequeno texto descrevendo algumas situações que você imagina que possam ser problemáticas com o seu código.