Especificação do projeto 1 **Chat**
Redes de Computadores/Sistemas Distribuídos 2017.1
Prof. Filipe

Chat

Neste projeto, você construirá uma aplicação simples que se conecta pela rede: um servidor chat. Similar ao IRC e Slack, seu servidor finalizado permitirá que usuários conversem em canais diferentes. Usuários podem criar ou se juntar a canais; desde que um usuário está em um determinado canal, todas as mensagens que ele enviar serão entregues a todos outros usuários do canal.

Este projeto servirá para introduzi-lo em programação socket. Este projeto é o mesmo de [1], portanto pode utilizar os recursos disponibilizados no site, basta ver referência.

Logística

- A data de entrega do projeto é a definida no SIGAA.
- O projeto é individual.
- Há um esqueleto do projeto em python disponível em `https://github.com/NetSys/cs168_student/blob/master/projects/proj1_chat`. Caso programe em uma linguagem de programação diferente de python, você pode adaptar os arquivos de teste para a sua linguagem:

  client_split_messages.py assegura que seu servidor está bufferizando corretamente a mensagem e é descrito em mais detalhes abaixo.

  simples_test.py testa o cenário básico onde dois clientes se comunicam utilizando um canal de comunicação.
- Uma demonstração do projeto pode ser vista em `https://www.youtube.com/watch?v=4btZs--wlpI`.

O que são sockets

Socket é um ponto final de comunicação entre dois programas executando através de uma rede de computadores. Cada socket é associado e um número particular de porta. Sockets são uma abstração provida pelos sistemas operacionais: programas criam sockets, lêem dados daquele socket e escrevem dados naquele socket. Quando um programa escreve em um socket, o sistema operacional envia os dados para uma porta particular associada ao programa; similarmente, quando o sistema operacional recebe dados em uma porta, os dados ficam disponibilizados para leitura por um socket correspondente aquela porta.

Todas linguagens de programação de alto nível disponibilizam uma biblioteca que lida com sockets. Em python, você pode utilizar a bibliotexa socket.

```
import socket
# Construtor do socket aceita poucos argumentos; O padrão é suficiente para este exemplo.
client_socket = socket.socket()
client_socket.connect(("1.2.3.4", 5678))
client_socket.sendall("Hello World")
```

O exemplo acima criou um socket e conectou-o a porta 5678 no IP 1.2.3.4 . Então, enviou a mensagem "Hello World" para o servidor em 1.2.3.4:5678.

O exemplo acima criou um cliente que foi conectado a exatamente um ponto remoto. Quando você cria um servidor, você vai querer que diversos clientes possam conectar, e você geralmente não sabe os endereços desses clientes que conectarão. Assim, o socket servidor funciona de maneira diferente.

```
server_socket = socket.socket()
server_socket.bind(("1.2.3.4", 5678))
server_socket.listen(5)
```

Depois de criar um socket, ao invés de conectar a um destino remoto particular, o código acima associa o socket a um ip e porta, o sistema operacional faz isso. Finalmente, listen faz com que se escute conexões direcionadas ao socket. Quando um cliente se conecta ao socket, a biblioteca socket criará um novo socket para se comunicar com o cliente, portanto o socket servidor continua a ser utilizado para esperar conexões de outros clientes:

```
(new_socket, address) = server_socket.accept()
```

Essa chamada bloqueia o programa até um cliente conectar (chamando connect()) e retorna um novo socket, new_socket, que pode ser utlizado para enviar e receber dados de um cliente. Por exemplo, a chamada

```
message = new_socket.recv(1024)
```

Bloqueará o programa até que receba dados do cliente, e retornará até 1024 bytes de dados.

Projeto

Você pode quebrar o projeto em duas partes, para facilitar a verificação da evolução da programação.

0.1 Parte 0

A primeira parte ajudará a você iniciar com a programação de sockets, introduzindo a interação básica entre cliente e servidor. O cliente enviará uma mensagem simples, obtida a partir da entrada padrão, para o servidor e então desconectará. O servidor imprimirá a mensagem recebida na saída padrão. Se múltiplos clientes conectarem-se ao servidor, devem ser manipulados sequencialmente, isto é, o servidor deve imprimir a mensagem completa de um cliente e fechar a conexão antes de manipular o próximo cliente.

O servidor deve aceitar um argumento de linha de comando, definindo a porta que irá utlizar.

```
$ python basic_server.py 12345
```

O cliente deve aceitar dois argumentos em linha de comando: o hostname (ou endereço IP) do servidor ao qual se conectará e a porta:

```
$ python basic_client.py localhost 12345
```

O servidor deve ser alcançável a partir de qualquer IP associado com a máquina.

0.1.1 Sockets bloqueantes

Nesta parte do desenvolvimento, é interessante utilizar sockets bloqueantes. Por bloqueante, entenda que o socket não retornará até que a chamada esteja completa. Casos em que chamadas ao socket não retornarão resultados imediatamente incluem:

- send: se o buffer interno do socket está completo, então não há espaço para os dados serem escritos.
- recv: se o buffer interno está vazio, não há mais nada a ser lido.
- accept: se não há mais clientes querendo estabelecer uma conexão.

0.1.2 Exemplos de uso

Aqui um exemplo de como seu cliente e servidor devem trabalhar. Suponha dois clientes diferentes conectados sequencialmente:

```
$ python basic_client.py localhost 12345
I am a student in CS168. This class is awesome!
$ python basic_client.py localhost 12345
Why is Shenker so bad at drawing?
```

Se um servidor executando e escutando a porta 12345 antes de um cliente se conectar, então ele deve imprimir como se segue:

```
$ python basic_server.py 12345 I am a student in CS168. This class is awesome! Why is
Shenker so bad at drawing?
```

0.2 Parte 1

0.2.1 Funcionalidade do servidor

O servidor deve aceitar um único argumento em linha de comando, que é a porta em que o servidor deveria escutar conexões.

Diferentemente do servidor na parte 0, o servidor nessa parte do trabalho deve aceitar conexões de vários clientes e receber mensagens concorrentemente dos mesmos. Cada cliente deve ter um nome associado, para que outros clientes conectados saibam de quem a mensagem partiu, e o canal de comunicação ao qual estão inscritos. A primeira mensagem que o cliente recebe deve ser utilizada como nome do cliente.

Mensagens futuras do cliente ao servidor podem ter uma das duas formas. O primeiro tipo de mensagem é de controle, mensagens de controle sempre iniciam com "/". Há três mensagens de controle diferentes que o servidor deve manipular:

- /join <canal> deve adicionar um cliente a um determinado canal. Clientes podem estar associados a um canal por vez, assim, se o cliente já está associado a um canal, este comando deve remover o cliente do canal. Quando um cliente é adicionado ou removido do canal, uma mensagem deve ser enviada indicando o ocorrido a todos os outros membros do canal.

- /create <canal> deve criar um novo canal com o nome fornecido. Juntamente com a criação do canal, o cliente deve sair do canal ao qual pertence atualmente para entrar no canal criado.

- /list deve enviar uma mensagem de resposta ao cliente com os nomes de todos os canais atuais, separados em linhas diferentes.

O segundo tipo de mensagem são mensagens comuns enviadas ao canal corrente do cliente. Todas mensagens que não são precedidas de / são consideradas mensagens comuns. Essas mensagens devem ser enviadas a todos os outros clientes no canal, precedida pelo nome do cliente em colchetes, veja exemplo mais abaixo. Mensagens não devem ser entregues a clientes que estão em outros canais. Se um cliente não está em um canal, o servidor deve enviar uma mensagem de erro ao cliente.

Quando um cliente desconecta, uma mensagem deve ser entregues a todos os outros clientes do canal, dizendo que o cliente se desconectou.

Sockets proveem a funcionalidade de fluxo de dados, mas eles não diferenciam que dado pertence a que mensagem. Quando a função recv retorna algum dado, o socket não dirá se é uma única mensagem, partes de uma mensagem ou múltiplas mensagens. Como resultado, você precisa determinar um meio de como as mensagens terminam. Para este projeto, utilize mensagens de tamanho fixo (200 caracteres), incluindo as mensagens do cliente. Se uma mensagem é menor que 200 caracteres, complete a mensagem com espaços em branco, e o receptor deve retirar todos os espaços em branco que foram utilizados para completar o tamanho da mensagem. Você deve assumir que nenhuma mensagem é maior que 200 caracteres.

Assegure que seu código manipula o caso onde menos que uma mensagem está disponível no buffer do socket (recv retornando menos que 200 caracteres de dados) e o caso onde mais de uma mensagem está disponível no buffer do socket. Você deve manipular mensagens parciais bufferizando: se recv recebe parte de uma mensagem, seu código deve manter a parte recebida até que receba o complemento e a mensagem possa ser manipulada por completo. Exemplo, se um cliente recebe 150 caracteres de um servidor, o cliente deve armazenar os 150 caracteres até que os 50 restantes sejam recebidos. O cliente deve mostrar a mensagem recebida somente que ela estiver completa. Para checar se o seu servidor está manipulando tais casos, um cliente que quebra mensagem em parte menores é disponibilizado (client_split_messages.py). O cliente só testa alguns cenários possíveis. Você pode modificar o código para testar casos adicionais.

0.2.2 Manipulação de erros

Seu servidor deve manipular os casos onde cliente envia mensagens invalidas, retornando uma mensagem de erro apropriada. Por exemplo, se um cliente utiliza /join, mas não provê o nome do canal ao qual se juntará, o servidor deve retornar uma mensagem de erro. O arquivo utils.py inclui as strings com todas as mensagens que deverá manipular. Por exemplo, utils.py define a seguinte mensagem de erro:

```
CLIENT_SERVER_DISCONNECTED = "Server at 0:1 has disconnected"
```

Você pode utilizar .format para substituir as chaves com os parâmetros que queira:

```
error_message = CLIENT_SERVER_DISCONNECTED.format("localhost", 12345)
```

Você deverá utilizar as mensagens de erro definidas em utils.py. Se não utilizar essas mensagens de erro, não obterá pontos por manipulação de erros.

Quando um comando ocasiona um erro, o comando não deve causar qualquer mudança no servidor. Por exemplo, se o cliente está atualmente no canal cs168_tas e tenta se juntar a um canal inexistente, o cliente **não** deve ser removido do canal atual.

0.2.3 Funcionalidade cliente

Cada cliente se conecta a servidor chat particular e está associado a um nome. Seu cliente deve iniciar como:

```
$ python client.py Scott 127.0.0.1 55555
```

Esse comando deve conectar o cliente ao servidor executando nesse IP e porta, e enviar uma mensagem com o nome Scott.

Depois de iniciado, o cliente deve escutar por mensagens vindo do servidor e da entrada padrão. Mensagens do cliente devem ser impressas na saída padrão, depois de removidos os espaços no final, e mensagens da entrada padrão devem ser enviadas ao servidor, depois de adicionados espaços ao final, quando necessários. O cliente deve imprimir [Me] a cada nova linha, para ficar claro no histórico de mensagens quais enviou. Quando o cliente recebe mensagem do servidor, ele deve escrever juntamente ao [Me] a mensagem enviada pelo servidor. Veja o vídeo exemplo já disponibilizado nos recursos.

Vajamos um exemplo de interação do cliente com um servidor executando localmente na porta 55555:

```
python chatv3_client.py Panda localhost 55555
Me Hello world!
Not currently in any channel. Must join a channel before sending messages.
[Me] /list
[Me] /create 168_tas
[Me] /list
168_tas
[Me] Hello world!
Alice has joined
[Alice] Hi everyone! Does anyone know what we're doing on the first day of lecture?
```

Depois de ver a mensagem de Alice, Panda encerrou seu cliente.

```
python chatv3_client.py Alice 127.0.0.1 55555
[Me] /list
168_tas
[Me] /join 168_tas
[Me] Hi everyone! Does anyone know what we're doing on the first day of lecture?
Panda has left
```

0.2.4 Socket não bloqueantes

Você precisará utilizar socket não bloqueantes, porque seu cliente e servidor necessita receber dados de múltiplas fontes, em ordem não conhecida. Considere o que aconteceria se o cliente utilizasse um socket bloqueante como na parte 0:

```
message_from_server = client_socket.recv(200)
```

Suponha que o servidor não envie nenhuma mensagem. Enquanto o cliente está bloqueado esperando o retorno de dados de recv, o usuário digita alguma coisa na entrada padrão. O cliente deve ler esses dados digitados pelo cliente e enviar ao servidor, mas o cliente continua bloqueado esperando dados vindos do servidor. Para resolver essa situação, você deve utilizar sockets não bloqueantes. O uso de sockets não bloqueantes depende da linguagem de programação escolhida para implementação.

Agradecimentos

Prof. Scott Schenker por disponibilizar o trabalho na internet.

Referências

[1] Schenker, Scott. 2016. *CS 168*[online]. University of California at Berkeley. Disponível em: (https://github.com/NetSys/cs168_student/blob/master/projects/proj1_chat/spec.md).