

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Redes de Computadores

Trabalho prático 3

Cloud Text

Aluno: Caíque Bruno Fortunato
Matrícula: 2013062731

Belo Horizonte, Junho de 2018

Introdução

Atualmente aplicações utilizando redes de computadores são extremamente comuns em nosso cotidiano e cada vez mais presentes nas mais diversas aplicações que, conseqüentemente deixam as pessoas mais conectadas e automatiza processos. Como foi desenvolvido nos trabalhos anteriores, a comunicação entre os computadores pode ser feita através de bibliotecas, como o Socket.

Socket é um componente do sistema que permite a comunicação entre processos através da rede por meio dos protocolos de transporte, sendo também uma abstração computacional que meia diretamente a uma porta de transporte mais um endereço de rede. Sendo assim, navegadores web utiliza sockets para requisitar páginas, sistemas também usam o recurso para comunicar com banco de dados, entre várias outras aplicações.

Objetivo e justificativa

O principal objetivo deste trabalho é desenvolver uma aplicação que utilize comunicação entre um servidor e diferentes clientes que permita a permita o usuários a enviarem arquivos de texto para um “pasta pessoal” e solicitar arquivos quando desejar. Para isso será utilizado Socket e o protocolo TCP, que fornece uma conexão bidirecional, sequencial e com fluxo único de dados. Além disso, o programa deverá funcionar tanto em IPv4 quanto em IPv6.

A motivação é colocar em prática alguns conceitos aprendidos na disciplina de Redes de Computadores, como TCP, Socket e protocolos de conexão, por exemplo.

Desenvolvimento

Nesta seção será apresentado como foi tratado a conexão de rede utilizando o Socket e o desenvolvimento da aplicação que roda por trás de tal conexão, chamada de Protocolo Cloud.

Conexão utilizando Socket

Para fazer a comunicação entre o cliente e servidor foi utilizado a biblioteca Socket na linguagem de programação Python 3.6, observando os requisitos:

1. O servidor deverá tratar múltiplas conexões;
2. Os programas deverão funcionar utilizando o TCP;
3. Os programas deverão funcionar com IPv4 e IPv6.

Para tratar a conexão foram criados dois arquivos: cliente.py e servidor.py que atendem os requisitos especificados.

O servidor deverá tratar múltiplas conexões

Para atender a tal requisito foi utilizada a biblioteca `_thread`. Parte do código do servidor encontra-se abaixo, que trata as múltiplas conexões.

```
service_socket = socket(AF_INET6, SOCK_STREAM)
service_socket.bind((host, port))
service_socket.listen(100)
p = ProtocoloCloud()

while True:
    client_socket, addr = service_socket.accept()
    _thread.start_new_thread(con_cliente, (client_socket, addr, p))

service_socket.close()
```

Depois da criação do socket, definindo a porta e o host é chamado o método `listen`, que coloca o socket em modo de espera definindo um máximo de 100 conexões pendentes que devem ser aguardadas.

É instanciado um objeto chamado `p`, da classe `ProtocoloCloud()` que possui a responsabilidade de receber e interpretar uma mensagem recebida pelo cliente, retornando a resposta esperada, conforme especificação. Tal classe será melhor especificada nas próximas seções.

Dentro do loop principal do servidor.py, a cada conexão aceita é criada uma nova thread que irá tratar o recebimento, interpretação e envio de mensagens através do método `con_cliente` que recebe como parâmetro o objeto `p` com as informações já armazenadas durante a conexão, mantendo ativa a base de usuários, arquivos e mensagens, por exemplo.

Os programas deverão funcionar com IPv4 e IPv6

Conversão de IPv4 para IPv6

Neste trabalho o programa utiliza IPv6 como padrão de comunicação entre o cliente e servidor por padrão. A técnica utilizada para não ter problema com as versões foi a seguinte: caso uma entrada do host do cliente seja IPv4 o programa irá converter para IPv6 (Ex: 127.0.0.1 → ::FFFF:127.0.0.1), assim a conexão pode ser estabelecida independente do tipo de IP utilizado.

Como saber a arquitetura do ip digitado?

Conforme descrito na documentação, o IP pode ser informado por número ou por nome, exemplo: login.dcc.ufmg.br. Além disso, pode ser tanto na arquitetura IPv4 quanto IPv6. A biblioteca `socket` do Python 3 possui uma função chamada `getaddrinfo` que informa diversas informações sobre o endereço informado: `getaddrinfo(host, port[, family[, socktype[, proto[, flags]]]])`

Segundo a documentação a função: *“Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service ((family, socktype, proto, canonname, sockaddr)). host is a domain name, a string representation of an IPv4/v6 address or None. port is a string service name such as 'http', a numeric port number or None. (...)”*

Sendo assim, o primeiro argumento possui o tipo de arquitetura, IPv4 ou IPv6 e o último a sequência numérica do IP. Se o IP informado for somente do tipo IPv4 (exemplo 127.0.0.1) é realizada a conversão, caso contrário é mantido o endereço obtido.

Funcionamento do Cloud Text

Foram criados três classes para fazer o funcionamento do programa Cloud Text: *Usuário*, *ConteudoMsg* e *ProtocoloCloud*. Em síntese, *ProtocoloCloud* recebe uma mensagem e trata ela, identificando o tipo. Nesta mesma classe possui uma lista de usuários (*Usuário*) que possuem uma lista de arquivos que possuem uma mensagem específica (*ConteudoMsg*) .

ProtocoloCloud

Tal classe possui como objetivo receber as mensagens que foram enviadas pelo cliente, identificando o tipo de mensagem e seu respectivo tratamento de modo a enviar uma resposta para que o servidor envie para o cliente. É criada uma lista de usuários cadastrados, que é manipulada em praticamente todos os métodos. Basicamente são quatro tipo de mensagens: registro(N), envio(S), recebimento (R) e lista(L).

Alguns métodos são:

- *verificaUsuário*: Verifica se o usuário já existe na lista de usuários cadastrados no sistema.
- *verificaSenha*: Uma vez que o usuário exista, o método verifica se a senha enviada corresponde com a cadastrada.
- *interpretaDadosRegistro*: Possui como objetivo verificar se já não existe um usuário de acordo com os parâmetros passados.
- *interpretaDadosEnvio*: Verifica se existe o usuário e verifica a senha. Possui a principal função de gravar mensagens, caso já exista ele substitui o dado e informa ao usuário.
- *interpretaDadosRecebimento*: Verifica se a mensagem existe, se o usuário existe e se a senha confere e enviar para o cliente o resultado do comando e o conteúdo da mensagem.
- *interpretaDadosLista*: Lista os arquivos existentes de um determinado usuário, caso o usuário e senha estejam corretos.

- *retornaResposta*: Dado as respostas dos quatro 4 métodos listados nos tópicos anteriores, o método interpreta e envia uma string de resposta para que o servidor envie a mensagem para o cliente.

Usuário

Classe que representa um determinado usuário. Cada usuário possui um *username*, *senha* e uma *lista de arquivos*. Sendo assim, a classe possui tais atributos e os seguintes métodos:

- *verificaArq*: Verifica se existe um arquivo específico na lista de arquivos cadastrados.
- *setMensagemArq*: Define uma mensagem a ser gravada em um arquivo específico.
- *consultaMensagem*: Uma vez que o arquivo existe, ele consulta a mensagem presente nele.
- *getListaArquivos*: Lista os arquivos do usuário, caso existam ou não.

ConteudoMsg

Classe que possui como objetivo ser a abstração de uma mensagem de um arquivo possuindo um conteúdo específico. Os métodos existentes são praticamente getters e setrs.

Testes

Os testes foram realizados em uma máquina Linux com distribuição Ubuntu 16.04 LTS de 64 bits, 8GB de memória RAM, 1 TB de HD e processador Intel Core i5 5º geração. O servidor foi inicializado e, posteriormente o cliente.

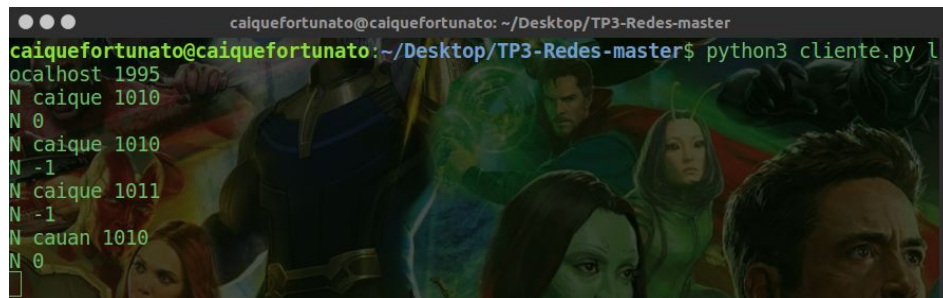
Teste da mensagem de registro

Ao receber uma mensagem (N), o servidor precisará verificar se já não existe um usuário com esse identificador. O servidor deverá responder o cliente informando o sucesso ou não da operação:

N 0 Sucesso, usuário criado.

N -1 Erro, usuário já existe.

Foi criado um usuário de nome *caique* e senha *1010*. Quando adicionado no sistema foi retornado N 0. Uma vez que já existia esse usuário foram feitos dois testes: com os mesmos usuários e senha e com usuário igual e senha diferente. Por fim, foi adicionado outro usuário.



```
calquefortunato@calquefortunato: ~/Desktop/TP3-Redes-master
calquefortunato@calquefortunato:~/Desktop/TP3-Redes-master$ python3 cliente.py l
localhost 1995
N caique 1010
N 0
N caique 1010
N -1
N caique 1011
N -1
N cauan 1010
N 0
```

Conforme pode ser visualizado na imagem acima, os resultados foram os esperados para as duas situações possíveis.

Teste da mensagem de envio

Ao receber uma mensagem (S), o servidor precisará verificar se já existe uma mensagem com este mesmo nome (e mesmo assim substituir seu conteúdo), verificar se existe o usuário e verificar a senha.

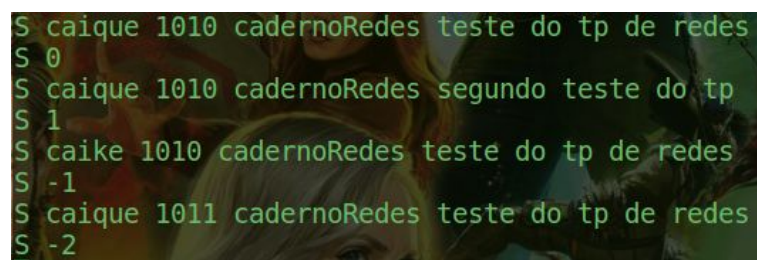
S 1 Sucesso, mensagem foi salva porém substituiu um conteúdo

S 0 Sucesso, mensagem salva.

S -1 Erro, usuário não existe.

S -2 Erro, senha incorreta.

Para realizar o teste foram criados 4 casos, cada um para uma mensagem específica. Como o programa iniciou do 0, não existe nenhum conteúdo. As mensagens enviadas e seus respectivos resultados encontram-se abaixo:



```
S caique 1010 cadernoRedes teste do tp de redes
S 0
S caique 1010 cadernoRedes segundo teste do tp
S 1
S caike 1010 cadernoRedes teste do tp de redes
S -1
S caique 1011 cadernoRedes teste do tp de redes
S -2
```

Observa-se que o comportamento do sistema segue o esperado.

Teste da mensagem de recebimento

Ao receber uma mensagem (R), o servidor precisará verificar se a mensagem existe, se o usuário existe e se a senha confere e enviar para o cliente o resultado do comando e o conteúdo da mensagem.

R 0 [mensagem] (*conteúdo da mensagem)*

R -1 Erro, usuário não existe.

R -2 Erro, senha incorreta.

R -3 Erro, mensagem não encontrada

Foram feitos testes para cada uma das mensagens que possam existir, de modo que as entradas e saídas encontram-se na imagem abaixo. Ressaltando que os testes são feitos em sequência, ou seja, mensagens e usuários são os mesmos que foram mostrados anteriormente.



```
R caique 1010 cadernoRedes
R 0 segundo teste do tp
R caique 1010 cadernoRC
R -3
R caique 1011 cadernoRedes
R -2
R caike 1010 cadernoRedes
R -1
```

Teste da mensagem de lista

*Ao receber uma mensagem (L), o servidor precisará verificar se o usuário existe e se a senha confere. (*Nome dos arquivos)*

L 0 [arq] [...*]. Se não houver nenhum, deixe apenas "L 0".*

L -1 Erro, usuário não existe.

L -2 Erro, senha incorreta.

Para realizar esse teste, primeiro foi inserido um usuário que não possuía nenhum arquivo cadastrado de forma que o resultado fosse L 0. Posteriormente foi testado com usuário não existente e senha incorreta. Por fim, para testar a lista foi criado um novo documento para o usuário *caique* utilizando a mensagem de envio.

As entradas e saídas podem ser visualizadas na imagem abaixo. A ordem dos testes são são respectivos à descrição do parágrafo acima.


```

S caique 1010 cadernoMSI teste de monografia
S 0
S caique 1010 cadernoFinanceira teste de financeira
S 0
L caique 1010
L 0 cadernoRedes cadernoMSI cadernoFinanceira
L cauan 1010
L 0
L caique 1011
L -2
L caik 1010
L -1

```

As saídas estão de acordo com o esperado.

Teste com múltiplas conexões

Também foi testado múltiplas conexões, como pode ser visto na imagem abaixo:

```

caiquefortunato@caiquefortunato: ~/Desktop/TP3-Redes-master
caiquefortunato@caiquefortunato:~/Desktop/TP3-Redes-master$ p
python3 cliente.py localhost 1995
N caique 1010
N -1
N cauan 1010
N -1
N caique 1010
L 0 diario
L 0

```

```

caiquefortunato@caiquefortunato:~/Desktop/TP3-Redes-master
caiquefortunato@caiquefortunato:~/Desktop/TP3-Redes-master$ p
python3 cliente.py localhost 1995
N caique 1010
N 0
S caique 1010 diario isso eh um teste
S 0

```

```

caiquefortunato@caiquefortunato:~/Desktop/TP3-Redes-master
caiquefortunato@caiquefortunato:~/Desktop/TP3-Redes-master$ p
python3 cliente.py localhost 1995
N cauan 1010
N 0
R caique 1010 diario
R 0 isso eh um teste

```

```

caiquefortunato@caiquefortunato:~/Desktop/TP3-Redes-master
caiquefortunato@caiquefortunato:~/Desktop/TP3-Redes-master$ python3 servidor.py 1995

```

Foram criados três conexões com o cliente / servidor, de modo que foram inseridos dados aleatoriamente em cada cliente e o servidor respondeu de maneira correta.

Por exemplo, inicialmente foi inserido o usuário caique 1010 no primeiro quadrante e o servidor respondeu: N 0, usuário não existe. Quando a consulta foi feita no segundo quadrante o servidor respondeu N -1, ou seja: o usuário já existe. O mesmo ocorreu quando o usuário cauan foi inserido no terceiro quadrante, quando consultado no segundo a resposta foi N -1.

Conclusão

Com o desenvolvimento do presente trabalho foi possível criar um programa que aceita múltiplas conexões, de modo que vários clientes conseguem comunicar com o servidor simultaneamente, o que é bem próximo de aplicações utilizadas no dia-a-dia. Para isso foi possível revisar e aplicar os conceitos de paralelismo e como aplicar na linguagem Python, muito usada no mercado.

Além disso, depois de ter utilizado o TPC e UDP em trabalhos anteriores foi possível entender melhor a diferença entre os dois e porque o TCP foi uma boa escolha para o programa. Precisamos aqui, por exemplo, garantir a confiabilidade no recebimento íntegro das mensagens, garantindo a existência de uma conexão.

Devido do paradigma escolhido para desenvolver o trabalho, de orientação a objetos, foi possível aprender melhor como é utilizado em Python e quanto foi válido separar a conexão entre cliente / servidor do programa, uma vez que isso melhora a modularização e manutenção, caso necessário.

As principais dificuldades encontradas foi determinar a estratégia utilizada, a modelagem de classes e seus relacionamentos, além de ter que pesquisar sobre paralelismo e como aplicar tal técnica no socket.

Referências Bibliográficas

1. ***How to convert IPv4-mapped-IPv6 address to IPv4 (string format)?***

Disponível em:

<https://stackoverflow.com/questions/11233246/how-to-convert-ipv4-mapped-ipv6-address-to-ipv4-string-format?utm_medium=organic&utm_source=google_e_rich_ga&utm_campaign=google_rich_ga> Último acesso: 6 de maio de

2018.

2. Documentação do Python 3. Disponível em: <<https://docs.python.org/3>>

Último acesso: 24 de junho de 2018.

3. Wiki Python Brasil. Disponível em: <<https://wiki.python.org.br/SocketBasico>>

Último acesso: 24 de junho de 2018.