Universidade Federal de Uberlândia

Faculdade de Engenharia Elétrica

1º trabalho de Redes de Computadores

Servidor web simples em Python

Aluno
Roní G. Gonçalves
10921EEL026

Professor
Paulo R. Guardieiro

Uberlândia, 20 de julho de 2014

Sumário

1 Introdução **3**

2 Implementação, testes e resultados **4**
2.1 Estabelecer uma conexão por meio de sockets 4
2.2 Receber o pedido de conexão por meio de sockets 5
2.3 Fazer o *parsing* da mensagem recebida 5
2.4 Obter o arquivo solicitado na mensagem recebida 5
2.5 Criar a mensagem de resposta 5
2.6 Enviar a mensagem de resposta 8

3 Conclusões **10**

4 Referências **11**

1 Introdução

Neste trabalho é proposta uma solução para a tarefa número 1 da sexta edição do livro de redes de computadores de James Kurose e Keith Ross [3], que se trata de programação de *sockets* para desenvolver um servidor WEB simples em Python que seja capaz de: (i) estabelecer uma conexão com o socket quando contatado pelo cliente (navegador da internet); (ii) receber o pedido HTTP desta conexão; (iii) fazer o *parsing* da mensagem para definir qual é o arquivo solicitado; (iv) obter o arquivo pedido do sistema de arquivos do servidor; (v) criar uma mensagem HTTP de resposta composta do arquivo pedido precedido de linhas de cabeçalho e, finalmente, (vi) enviar a resposta através da conexão TCP para o cliente. Se o navegador pede por um arquivo que não se encontra no diretório de seu servidor, ele deve retornar uma mensagem de erro 404.

2 Implementação, testes e resultados

Nesta seção, é apresentada a implementação do servidor WEB para cada um dos seis ítens pedidos. Os testes realizados que mostram o cumprimento do que foi pedido são também mostrados, muitos deles por meio do programa Wireshark.

2.1 Estabelecer uma conexão por meio de sockets

Python é uma linguagem de programação que permite o desenvolvimento de aplicações complexas muito rapidamente por causa de suas vastas bibliotecas ou, como os *pythonistas* as chamam, módulos. Usando o módulo `socket` facilita a criação do servidor. Para se estabelecer uma conexão é necessário criar um *socket*, atrelá-lo a uma porta e fazê-lo "ouvir" as conexões feitas a ele.

```
1  from socket import *

   .
4  .
   .

7  conexao = socket(AF_INET, SOCK_STREAM)

   conexao.bind(('', 65000))
10

   conexao.listen(10)
```

Nesse caso, o *socket* criado se chama `conexao`[1], os parâmetros AF_INET e SOCK_STREAM dizem respeito ao endereço IPv4 – AF: *address family* – e ao tipo de *socket* criado.

A porta escolhida para prestar o serviço WEB foi a de número 65000. A porta 80 foi deixada de lado apenas por segurança. Não há nenhum problema em usá-la, no entanto.

O método `listen()` limita quantas conexões feitas com o *socket* serão ouvidas. Segundo a documentação do Python, o valor máximo de seu argumento depende do sistema operacional e comumente é igual a 5. Mesmo assim, o valor usado nesse caso foi 10.

[1] Este é o famoso *Welcoming socket*...

2.2 Receber o pedido de conexão por meio de sockets

Para, depois de criar o *socket*, receber pedidos de conexão, basta usar o método `accept()`. Tal método retorna dois objetos: o primeiro, trata-se de um novo *socket* que é usado para enviar e receber dados pela conexão estabelecida; o segundo é o endereço do cliente que fez o pedido de conexão.

```
1         client,addr = conexao.accept()
          print 'Conexao estabelecida com ', addr
```

2.3 Fazer o *parsing* da mensagem recebida

Uma vez que a conexão é estabelecida, basta receber por meio do *socket* `client` as mensagens enviadas pelo cliente usando o médodo `recv()`.

```
1 msg = client.recv(4096)

  msglist = msg.split()
```

O argumento do método `recv()` diz o tamanho máximo, em bytes, da mensagem que o cliente pode enviar ao servidor. Nesse caso, o tamanho é de 4096 bytes.

A mensagem recebida é guardada como uma *string*. Para fazer o *parsing* dessa mensagem, cada palavra deve ser separada ordenadamente, por isso o método `split()` é usado na *string* `msg` e seu resultado é armazenado em forma de uma lista denominada `msglist`.

2.4 Obter o arquivo solicitado na mensagem recebida

Considerando que a primeira palavra na lista `msglist` é o método HTTP GET, a segunda palavra desta lista deve ser o endereço do objeto pedido pelo navegador da internet. Por isso, para se obter o arquivo solicitado, basta conhecer qual é a *string* que o representa.

```
object_path = msglist[1]
```

Como os índices em Python começam em 0 – assim como em C – o segundo elemento de uma lista é acessado pelo índice 1.

2.5 Criar a mensagem de resposta

Criar a mensagem de resposta foi talvez a parte mais trabalhosa do servidor. Primeiramente, só existem três possibilidades de mensagens HTTP

que o servidor implementado cria:

1. OK 200: caso o arquivo solicitado exista no sistema de arquivos do servidor e o método de leitura deste arquivo consiga abrí-lo
2. 404 Not Found: caso o arquivo solicitado não exista
3. 400 Bad Request: caso o método enviado pelo cliente seja diferente de GET

Antes de prosseguir, duas funções auxiliares precisam ser apresentadas. A primeira a partir do nome do arquivo solicitado define qual o tipo de arquivo que será enviado ao cliente; a segunda a partir do tipo de arquivo definido, cria um cabeçalho específico para dois tipos de arquivos: html e jpeg.

```
def descobre_tipo_conteudo(string):

        tipo_conteudo = []
        auxiliar = ''

        for i in range(len(string)):

                auxiliar = string[(len(string) - 1) -
                   i]

                if (auxiliar == '.'):

                        break
                else:

                        tipo_conteudo.append(auxiliar)

        tipo_conteudo.reverse()

        return ''.join(tipo_conteudo)

def make_header(TYPE_OF_CONTENT):

        header = ''
```

```
        header = ('Connection: ' + CNX + 'Date: ' +
            strftime("%a, %d %b %Y %X GMT", gmtime()) +
             '\r\n' + 'Server: ' + SERVERNAME)

        if (TYPE_OF_CONTENT == 'JPG' or
            TYPE_OF_CONTENT == 'jpg'):

                header = header + CONTENT_TYPE_JPG + '
                    \r\n'
        else:
                header = header + CONTENT_TYPE_TXT + '
                    \r\n'

        return header
```

Por meio dessas funções, parte da mensagem é construída. Agora, uma vez tendo o nome do arquivo pedido, o programa procura por ele e tenta abrí-lo. Se não houver arquivo, ele tentará abrí-lo mesmo assim e um erro de exceção acontecerá, o que causará a construção da mensagem de erro 404 e o envio da página de erro correspondente.

```
existearquivo = True

if msglist[0] == 'GET':

        try:
                arquivo_pedido = open(('root' +
                    object_path), 'rb')

        except IOError:

                existearquivo = False

        if (existearquivo):

                client.send(OK_RESPONSE)
                client.send(make_header(content_type))
                arquivo_lido = arquivo_pedido.read()
                arquivo_pedido.close()
                client.send(arquivo_lido)
                client.send('\r\n')
```

```
        else:

                client.send(NOTFOUND_RESPONSE)
                client.send(make_header('text/html'))

                arquivo_pedido = open('root/404/404.
                    htm', 'r')
                arquivo_lido = arquivo_pedido.read()
                arquivo_pedido.close()

                client.send(arquivo_lido)
                client.send('\r\n')
```

2.6 Enviar a mensagem de resposta

O envio da mensagem, como pode ser visto logo acima, é feito pelo método `send()`. A resposta HTTP é enviado primeiramente (OK_RESPONSE, NOT-FOUND_RESPONSE ou BAD_REQUEST)[2]. Em seguida, as informações de cabeçalho são enviadas e, por fim, o próprio arquivo solicitado é enviado. Feito isso, só resta encerrar a conexão por meio do método `close()`.

```
HOST = 'localhost'

OK_RESPONSE = 'HTTP/1.1 OK 200 \r\n'

NOTFOUND_RESPONSE = 'HTTP/1.1 404 Not Found \r\n'

BAD_REQUEST = 'HTTP/1.1 400 Bad Request \r\n'

HEAD_END = '\r\n\r\n'

CNX = 'close\r\n'

SERVERNAME = 'ronizoide/0.1 (Linux)\r\n'

CONTENT_TYPE_TXT = 'text/html; charset=utf-8\r\n'
```

[2]Elas estão definidas num arquivo a parte chamado de constants.py. Isso foi feito por motivos de clareza e legibilidade do código.

```
17 CONTENT_TYPE_JPG = 'image/jpeg\r\n'

   CONTENT_TYPE_PNG = 'image/png\r\n'
```

3 Conclusões

Os algoritmos genéticos apresentam aplicações interessantes e importantes também em áreas onde não há ainda soluções matemáticas analíticas ou numéricas para certos problemas. Por exemplo, num caso de otimização de funções, a única forma de se encontrar realmente um mínimo ou máximo global seria esgotando todas as possibilidades, o que num problema com domínio nos números reais seria uma tarefa impossível dada a limitação computacional existente hoje. Os algoritmos genéticos permitem encontrar aproximações para esses problemas percorrendo de uma forma metódica boa parte do espaço de busca onde a resposta correta se encontra. A esse metodismo é acrescentada certa aleatoriedade por meio dos mecanismos de cruzamento e mutação que não deixam o algoritmo procurar por respostas somente num conjunto restrito de supostas soluções. Como se pôde ver na seção **??**, ainda que não haja garantias de se obter a resposta correta, um resultado próximo do ideal é encontrado.

4 Referências

[1] Dave Kulhman, *A Python book: Beginning Python, Advanced Python, and Python Exercises*. Abril de 2012.

[2] David Beazley, *Python Essential Reference*. Sams Publishing, Indiana – EUA, 2006.

[3] James Kurose e Keith Ross, *Computer Networking: A Top-Down Approach*. Pearson Education, 2013.

[4] Kenneth Lambert, *Fundamentals of Python: From First Programs through Data Structures*. Cengage Learning, 2010.

[5] Paulo Roberto Guardieiro, *Notas de aula da disciplina Redes de Computadores*. Faculdade de Engenharia Elétria, Universidade Federal de Uberlândia, 2014.

[6] Roy Fielding et al., *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*. The Internet Society, 1999.