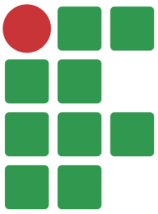

INSTITUTO FEDERAL DA PARAÍBA
CAMPUS CAMPINA GRANDE
TECNOLOGIA EM TELEMÁTICA
DISCIPLINA DE PROGRAMAÇÃO III
PROF. VICTOR ANDRÉ PINHO DE OLIVEIRA

Programação III

Aula 4 - Preparando dados para transmissão - Parte 2

Introdução

Olá,

Na semana anterior, nós vimos 6 abordagens de como podemos implementar framing na troca de dados entre hosts que usam Sockets TCP. Adicionalmente, eu mostrei para vocês formas de enviarmos qualquer tipo de dado via rede, desde os mais básicos (int, char, string, float) até os tipos de dados mais complexos do Python (Lista, Dicionário etc.). E isso foi possível graças aos módulos struct e pickle, respectivamente.

Na aula de hoje nós vamos continuar trabalhando o tema “Preparando dados para transmissão”. Dessa vez, vamos nos desprender do Python e falaremos sobre alguns padrões universais comumente usados na troca de dados através da Internet. Me refiro ao JSON, CSV e XML.

Não poderíamos deixar de falar sobre compactação, mecanismo importantíssimo para reduzir a quantidade de bytes transferidos pela rede.

Bom proveito!

Preparando dados para transmissão

Quando olhamos para trás, desde o momento em que começamos a estudar Sockets, vemos o quanto progredimos. Começamos estudando Sockets UDP. Trabalhávamos com o tipo string somente e a codificávamos antes de enviar pela rede. Depois iniciamos Sockets TCP e continuamos a trabalhar com strings. Na aula anterior, entretanto, nós vimos como podemos enviar qualquer tipo de dados, sejam eles dados básicos, comuns a todas as linguagens, ou dados mais complexos, particulares à linguagem Python.

Agora nós iremos dar mais um passo nessa jornada, sairemos da “caixa Python” e veremos alguns padrões universais que são comumente empregados na troca de dados através da Internet. Falaremos sobre o JSON, CSV e XML.

O padrão JSON

JSON significa *JavaScript Object Notation* (Notação de Objetos JavaScript) e é um padrão legível para humanos. Sim, JavaScript é uma linguagem de programação para Web e o JSON é um subconjunto da linguagem JavaScript, mas você não precisa conhecer-la, pois o padrão é completamente independente da linguagem.

Em essência, o JSON é constituído por:

- uma coleção de pares chave/valor - exatamente igual ao tipo dicionário do Python
- uma lista de valores - exatamente igual ao tipo Lista do Python

Como a linguagem Python possui tipos que coincidem com a estrutura do JSON, acaba ficando muito simples trabalhar com JSON em aplicações Python.

Para mais detalhes sobre o padrão JSON, acesse https://www.json.org/json-pt.html.

Vejamos como podemos manipular o padrão JSON em códigos Python.

Primeiro, precisamos importar o módulo JSON:

```
>>> import json
```

Depois de importado, só precisaremos de duas funções:

- a função **dumps**: que nos permite codificar no formato JSON
- a função **loads**: que nos permite decodificar uma string JSON

Exemplo 1:

```
>>> json.dumps(["dado1",3.5,100])
'["dado1", 3.5, 100]'
>>> json.loads('["dado1", 3.5, 100]')
['dado1', 3.5, 100]
```

No exemplo acima passamos uma Lista contendo uma string, um ponto flutuante e um inteiro. Note que dumps retorna uma string da Lista passada, e loads processa a string e devolve o objeto Lista.

Exemplo 2:

```
>>> json.dumps({'Pýthon':False,'Python':True})
'{"P\\u00fdthon": false, "Python": true}'
>>> json.loads('{"P\\u00fdthon": false, "Python": true}')
{'Pýthon': False, 'Python': True}
```

No exemplo acima passamos um Dicionário para dumps. Note que a função faz alguns ajustes ao converter para uma string JSON. Os caracteres acentuados são codificados (observe a string Pýthon), espaços são colocados e os booleanos False e True se

transformam em false e true, respectivamente. Quando decodificamos com loads, obtemos nosso Dicionário exatamente como esperado.

Exemplo 3:

```
>>> dados = {"dict":{"primos":[2,3,6,7],"amigos":[284,220]}}
>>> json.dumps(dados)
'{"dict": {"amigos": [284, 220], "primos": [2, 3, 6, 7]}}'
>>> json.loads('{"dict": {"amigos": [284, 220], "primos": [2,
3, 6, 7]}}')
{'dict': {'amigos': [284, 220], 'primos': [2, 3, 6, 7]}}
```

O objetivo do exemplo acima é simplesmente ilustrar a possibilidade de empregar o formato JSON com um objeto mais complexo, combinando Dicionários e Listas.

Se for preciso ler os dados JSON de um arquivo, algo comum caso esteja implementando um Servidor, não há segredo nenhum. Basta usar a função nativa open para abrir o arquivo .json e invocar o método read do objeto arquivo, pois ele já retorna todo o conteúdo como string. De posse da string JSON, você é livre para enviar pela rede ou simplesmente carregar em um objeto Python:

```
>>> file = open("arquivo.json")
>>> dados = json.loads(file.read())
>>> file.close()
```

E caso queira armazenar uma string JSON num arquivo, basta abrir o arquivo em modo escrita e invocar o método write do objeto arquivo passando a string gerada pelo método dumps do módulo json.

```
>>> file = open("arquivo.json", "w")
>>> file.write(json.dumps(dados))
>>> file.close()
```

O padrão CSV

CSV significa comma-separated values (valores separados por vírgulas). É como se fosse uma planilha de dados em texto puro, com as vírgulas separando as colunas. Portanto, a tabela (planilha):

João	100	70	90	85	65
Pedro	95	80	75	60	70
Tiago	90	90	70		100

seria equivalente ao seguinte CSV:

João,100,70,90,85,65
Pedro,95,80,75,60,70

Tiago,90,90,70,,100

O exemplo acima diz respeito a 3 alunos e suas 5 notas. Note que o aluno Tiago não possui a quarta nota na tabela, pois ele não entregou a atividade :). Note também o campo vazio no CSV referente à nota na linha de Tiago.

Para ilustrar a manipulação do CSV em Python, vamos começar salvando os dados acima em um arquivo “notas.csv”.

Exemplo 1:

```
>>> import csv
>>> file = open("notas.csv")
>>> dados = csv.reader(file)
>>> for i in dados:
...   print(i)
...
...
['João', '100', '70', '90', '85', '65']
['Pedro', '95', '80', '75', '60', '70']
['Tiago', '90', '90', '70', '', '100']
```

O exemplo acima foi rodado diretamente no interpretador Python. Primeiramente, importamos o módulo csv. Depois, abrimos o arquivo com a função open, nativa do Python. Na linha seguinte, invocamos a função reader do módulo csv para ler e interpretar o arquivo CSV. A função retorna um objeto csv.reader que fica armazenado em dados. Depois, acessamos as linhas do CSV em um for. Note que cada linha vem transformada em uma Lista correspondendo ao CSV original. Adicionalmente, veja que todos os campos vêm convertidos em string.

Exemplo 2:

```
>>> import csv
>>> file = open("notas.csv","w")
>>> escritor = csv.writer(file)
>>> escritor.writerow(["João", 100, 70, 90, 85, 65])
22
>>> escritor.writerow(["Pedro", 95, 80, 75, 60, 70])
22
>>> escritor.writerow(["Tiago", '90', '90', '70', '', '100'])
21
>>> file.close()
```

O exemplo acima ilustra como podemos criar um arquivo CSV por meio do Python, usando o módulo csv. Na realidade, estamos produzindo o mesmo arquivo notas.csv. Depois que importamos o módulo, abrimos o arquivo em modo escrita. Invocamos a função writer do módulo csv. Ela retorna um objeto csv.writer que nos permite

escrever os dados no arquivo csv. Depois, utilizamos o método writerow com o objeto escritor para inserir uma linha no arquivo CSV. Note que a linha é passada como uma Lista de valores. O uso de aspas nos valores numéricos são opcionais.

Se você não precisar ler/escrever o formato CSV de/para um arquivo, é possível realizar a manipulação diretamente no código Python usando strings. Nesse caso, não se faz necessário importar nenhum módulo, visto que o formato CSV é muito simples.

Exemplo 3:

```
>>> str_csv = ",".join(["João", "100", "70", "90", "85",
"65"])
>>> str_csv
'João,100,70,90,85,65'
>>> str_csv += "\n" + ",".join(["Pedro", "95", "80", "75",
"60", "70"])
>>> str_csv
'João,100,70,90,85,65\nPedro,95,80,75,60,70'
>>> str_csv += "\n" + ",".join(["Tiago", "90", "90", "70", "",
"100"])
>>> str_csv
'João,100,70,90,85,65\nPedro,95,80,75,60,70\nTiago,90,90,70,,1
00'
```

No exemplo acima usamos o método join em uma string “,” para unir os elementos de uma lista em uma só string. Concatenamos um “\n” e depois seguimos o mesmo raciocínio para cada aluno.

Exemplo 4:

```
>>> str_csv.splitlines()
['João,100,70,90,85,65', 'Pedro,95,80,75,60,70',
'Tiago,90,90,70,,100']
dados = []
>>> for i in str_csv.splitlines():
...   dados.append(i.split(","))
...
...
>>> dados
[['João', '100', '70', '90', '85', '65'], ['Pedro', '95',
'80', '75', '60', '70'
], ['Tiago', '90', '90', '70', '', '100']]
```

Seguindo o raciocínio do exemplo 3, podemos extrair os dados da string gerando uma lista de lista. O exemplo acima começa exemplificando o método splitlines, que cria uma lista a partir da quebra do “\n” na string original. Depois criamos uma lista vazia

(dados) e entramos num loop para varrer cada elemento (string) da lista gerada por str_csv.splitlines(). Dentro do loop, invocamos o método split para quebrar cada string da lista a partir do caractere “,”. Finalmente, exibimos o resultado da lista dados.

O padrão XML

XML significa *Extensible Markup Language* (Linguagem de Marcação Extensível). É uma recomendação da W3C, cujo fim principal é facilitar o compartilhamento de dados através da Internet.

Entrar nos detalhes da XML foge do escopo desta aula, mas, de modo geral, um arquivo XML é formado por um conjunto de tags aninhadas. O formato XML também é legível para humanos. Abaixo temos o arquivo notas.xml que armazena os mesmos dados de notas.csv, porém no formato XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<alunos>
    <João>
      <n1>100</n1><n2>70</n2><n3>90</n3><n4>85</n4><n5>65</n5>
    </João>
    <Pedro>
      <n1>95</n1><n2>80</n2><n3>75</n3><n4>60</n4><n5>70</n5>
    </Pedro>
    <Tiago>
      <n1>90</n1><n2>90</n2><n3>70</n3><n4></n4><n5>100</n5>
    </Tiago>
</alunos>
```

A biblioteca padrão do Python fornece alguns módulos para a análise e manipulação do XML: xml.dom e xml.sax, por exemplo. São módulo excelentes, porém é preciso de bastante código para poder percorrer a estrutura e obter os dados que queremos. Por questão de simplicidade, para os nossos propósitos, iremos utilizar o módulo xmltodict. Como o próprio nome sugere, o módulo nos permite converter um XML direto em um dicionário Python, bem como permite gerar um arquivo XML a partir de um dicionário Python.

O módulo não faz parte da biblioteca padrão, mas pode ser instalado via pip:

```
user@~# pip3 install xmltodict
```

Depois de instalado e importado, só precisaremos de duas funções:

- a função **parse**: que recebe a string contendo o XML
- a função **unparse**: que recebe o dicionário e, opcionalmente, um objeto arquivo caso queiramos salvar a string contendo o XML direto num arquivo

Exemplo 1:

```
>>> import xmltodict
>>> file = open("notas.xml")
>>> notas = xmltodict.parse(file.read())
>>> notas['alunos']['João']['n1']
'100'
>>> notas['alunos']['João']['n2']
'70'
>>> notas['alunos']['João']['n3']
'90'
>>> notas['alunos']['João']['n4']
'85'
>>> notas['alunos']['João']['n5']
'65'
```

Primeiro importamos o módulo xmltodict. Em seguida, abrimos o arquivo. Depois, passamos todo o conteúdo do arquivo (usando o método read do objeto arquivo) à função parse do módulo xmltodict. Essa função converte a string contendo o XML em um dicionário.

Exemplo 2:

```
>>> notas['alunos']['Tiago']['n4']
>>> notas['alunos']['Tiago']['n4'] = 80
>>> file.close()
>>> file = open("notas.xml","w")
>>> xmltodict.unparse(notas, file)
>>> file.close()
```

O trecho acima conclui o exemplo anterior. Verificamos que Tiago está sem a quarta nota. Digamos que ele conversou com o professor e entregou o trabalho que estava faltando e, depois de corrigido, tenha ficado com 80. Atribuímos, então, 80 à quarta nota. Fechamos o arquivo que estava somente leitura e reabrimos no modo escrita. Em seguida usamos a função unparse, para salvar e converter o dicionário notas atualizado no arquivo notas.xml.

Assim como comentamos no CSV, se você não precisar ler/escrever o padrão XML de/para um arquivo, é possível manipular diretamente no código Python por meio de strings, mas, nesse caso, ainda vamos precisar do módulo xmltodict.

Exemplo 3:

```
>>> str_xml = xmltodict.unparse(notas)
>>> str_xml
'<?xml version="1.0"
encoding="utf-8"?>\n<alunos><João><n1>100</n1><n2>70</n2><n
```

```
3>90</n3><n4>85</n4><n5>65</n5></João><Pedro><n1>95</n1><n2>80
</n2><n3>75</n3><n
4>60</n4><n5>70</n5></Pedro><Tiago><n1>90</n1><n2>90</n2><n3>7
0</n3><n4></n4><n5
>100</n5></Tiago></alunos>'
```

Considerando o mesmo dicionário notas usado nos dois exemplos anteriores, podemos invocar a função unparse, passando apenas o dicionário como argumento, que ela vai retornar uma string com os dados no padrão XML.

Exemplo 4:

```
>>> notas = xmltodict.parse(str_xml)
>>> notas
OrderedDict([('alunos', OrderedDict([('João',
OrderedDict([('n1', '100'), ('n2',
 '70'), ('n3', '90'), ('n4', '85'), ('n5', '65')])), ('Pedro',
OrderedDict([('n1
', '95'), ('n2', '80'), ('n3', '75'), ('n4', '60'), ('n5',
'70')])), ('Tiago', O
rderedDict([('n1', '90'), ('n2', '90'), ('n3', '70'), ('n4',
None), ('n5', '100'
)]))]))])
```

De posse da string contendo o XML, podemos simplesmente invocar a função parse para obtermos o nosso dicionário de volta. Note que, na verdade, o dicionário gerado pela função parse é um OrderedDict e não um dicionário comum. Em um OrderedDict os dados sempre permanecerão na ordem em que foram inseridos, diferente do que acontece ao usarmos um dicionário comum.

Para mais informações sobre XML, acesse https://www.w3.org/standards/xml/core.

É importante que você compreenda bem os padrões, pois precisaremos deles mais à frente na disciplina.

Compactação

A compactação é um recurso imprescindível para transmissão de dados através da Internet. Compactar quer dizer reduzir e, quando compactamos os dados antes de enviá-los, estamos reduzindo a quantidade de bytes que irão trafegar pela rede.

A biblioteca padrão do Python fornece o módulo zlib. Vamos precisar de apenas duas funções:

- **compress**: recebe uma sequência de bytes e retorna uma sequência de bytes compactados
- **decompress**: recebe os bytes compactados e retorna os bytes originais.

Não será necessário mais que um exemplo para compreender o uso das funções.

Exemplo 1:

```
>>> import zlib
>>> zlib.compress(b"teste")
b'x\x9c+I-.I\x05\x00\x06\x83\x02&'
>>> zlib.compress(b"testeteste")
b'x\x9c+I-.I-\x01\x11\x00\x17\xbf\x04K'
>>> zlib.compress(b"testetesteteste")
b'x\x9c+I-.I-\x81\x13\x003\xb4\x06p'
>>> zlib.compress(b"testetestetesteteste")
b'x\x9c+I-.I-A%\x00Zb\x08\x95'
>>> zlib.compress(b"testetestetestetesteteste")
b'x\x9c+I-.I-\xc1B\x00\x00\x8b\xc9\n\xba'
>>> zlib.decompress(b'x\x9c+I-.I-A%\x00Zb\x08\x95')
b'testetestetesteteste'
```

Primeiro importamos o módulo zlib e depois invocamos sucessivas vezes a função compress. Observe que, à semelhança de quase todos os esquemas de compactação, quando a função compress recebe dados menores tendem a torná-los mais longos ao invés de mais curtos. Isso acontece porque a compactação gera um overhead, uma espécie de informação sobre a compactação que acaba ocupando algum espaço.

Desse modo, ao compactar b"testeteste" obtemos uma saída maior que b"testetesteteste". Mas atente que isso ocorre apenas para pequenas quantidades de dados.

Para descompactar uma sequência de bytes compactados, basta invocar decompress.

Como, então, integrar o que foi visto na aula de hoje com o que foi trabalhado na aula passada? Simples! Quer você esteja usando JSON, CSV, XML e/ou esteja compactando seus dados, escolha a melhor abordagem de *framing* antes de transmiti-los.

Abaixo temos uma aplicação onde o Cliente recebe o nome e três notas de vários alunos. Depois envia para o Servidor no formato JSON e compactado. O Servidor calcula a média, acrescenta à estrutura e retorna o formato JSON compactado ao Cliente. Utilizamos a abordagem 5 de framing vista na aula anterior. Se não lembra, a abordagem envia a quantidade de bytes da mensagem antes de enviar a mensagem propriamente dita.

Servidor:

```
import socket
import struct
import json
```

```
import zlib

def recvall(sock, length):
    data = b""
    while len(data) < length:
      more = sock.recv(length - len(data))

      if not more:
           raise EOFError("Falha ao receber os dados
esperados.")
      data += more
    return data

def recvmsg(sock):
    tam = struct.unpack("!i",recvall(sock,4))[0]
    return recvall(sock,tam)

def sendmsg(sock,msg):
    sock.sendall(struct.pack("!i",len(msg)))
    sock.sendall(msg)

def compress(msg):
    return zlib.compress(msg)

def decompress(msg):
    return zlib.decompress(msg)

def toJSON(msg):
    return json.dumps(msg)

def fromJSON(msg):
    return json.loads(msg)

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1',50000))
    sock.listen(1)

    print("Ouvindo em", sock.getsockname())
```

```
    while True:
        sc, sockname = sock.accept()

        notas = fromJSON(decompress(recvmsg(sc)).decode())

        for aluno in notas.values():
            n1 = float(aluno[0])
            n2 = float(aluno[1])
            n3 = float(aluno[2])
            aluno.append((n1+n2+n3)/3)

        sendmsg(sc,compress(toJSON(notas).encode()))

        sc.close()

    return 0
```

O código possui a mesma estrutura do código apresentado na abordagem 5 de framing da aula anterior. Para cumprir com o propósito, acrescentamos mais 4 funções:

- compress: a função simplesmente invoca a função compress do módulo zlib. A função espera o tipo bytes e retorna bytes.
- decompress: a função simplesmente invoca a função decompress do módulo zlib. A função espera o tipo bytes e retorna bytes.
- toJSON: a função simplesmente invoca a função dumps do módulo json. A função vai receber o nosso dicionário e retorna uma string convertida em JSON.
- fromJSON: a função simplesmente invoca a função loads do módulo json. A função espera receber um string JSON e retorna o dicionário.

No atendimento da conexão, recebemos os dados e descompactamos. Quando descompactamos, obtemos a string JSON codificada. Decodificamos e passamos para a função fromJSON nos retornar o dicionário. De posse do dicionário com as notas, calculamos as médias de todos os alunos.

Com o dicionário atualizado, convertemos para JSON e codificamos a string resultante. Campactamos os bytes antes de enviar ao Cliente.

Cliente:

```
import socket
import struct
import json
import zlib

def recvall(sock, length):
```

```
    data = b""
    while len(data) < length:
       more = sock.recv(length - len(data))

       if not more:
            raise EOFError("Falha ao receber os dados
esperados.")
       data += more
    return data

def recvmsg(sock):
    tam = struct.unpack("!i",recvall(sock,4))[0]
    return recvall(sock,tam)

def sendmsg(sock,msg):
    sock.sendall(struct.pack("!i",len(msg)))
    sock.sendall(msg)

def compress(msg):
    return zlib.compress(msg)

def decompress(msg):
    return zlib.decompress(msg)

def toJSON(msg):
    return json.dumps(msg)

def fromJSON(msg):
    return json.loads(msg)

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(('127.0.0.1',50000))

    notas = {}

    while True:
       nome = input("Entre com o nome do Aluno: ")
       if not nome:
            break
       n1 = input("Entre com a Nota 1: ")
```

```
        n2 = input("Entre com a Nota 2: ")
        n3 = input("Entre com a Nota 3: ")
        notas[nome] = [n1,n2,n3]

    sendmsg(sock,compress(toJSON(notas).encode()))

    res = fromJSON(decompress(recvmsg(sock)).decode())

    for nome, nota in res.items():
        print("Aluno ",nome,"obteve média ",nota[3])

    sock.close()

    return 0
```

De igual modo, aproveitamos o código do Cliente da abordagem 5 de framing da aula passada. Como o Cliente possui as mesmas funções que o Servidor, então as explicações já foram dadas. O que mais importa é o trecho da função main onde coletamos as notas dos alunos. Note que estamos armazenando as notas em uma lista que será colocada em um dicionário tendo o nome do aluno como a chave.

Depois de obtidas todas as notas de todos os alunos, convertemos o dicionário para JSON e codificamos a string retornada. Comprimimos os dados e enviamos.

Logo na sequência, recebemos a mensagem do buffer de entrada e a descompactamos. Os bytes descompactados são decodificados para obtermos a string JSON, que por sua vez é passada à função fromJSON para obtermos o dicionário.

De posse do dicionário, apresentamos as médias dos alunos.

Chegamos ao final de mais uma aula. 🤗

Nesta aula estudamos 3 padrões universais comumente usados na troca de dados através da Internet. Vimos também como podemos compactar nossas mensagens antes de enviá-las pela rede.

Já progredimos bastante, mas os nossos Servidores ainda possuem uma grave limitação: eles atendem apenas um Cliente por vez. Na aula que vem, veremos como empregar técnicas para atender a mais de um Cliente por vez. Eu acho simplesmente demais!

Bons estudos e até a próxima!