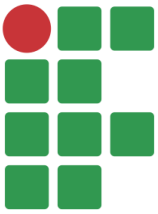

INSTITUTO FEDERAL DA PARAÍBA
CAMPUS CAMPINA GRANDE
TECNOLOGIA EM TELEMÁTICA
DISCIPLINA DE PROGRAMAÇÃO III
PROF. VICTOR ANDRÉ PINHO DE OLIVEIRA

Programação III

Aula 3 - Preparando dados para transmissão - Parte 1

Introdução

Olá,

Na semana anterior, nós estudamos como implementar Sockets TCP. Vimos que o TCP trabalha com *stream* (fluxo) e, portanto, a responsabilidade em preparar os dados, enquadrando-os, antes de transmitir, fica a cargo do programador.

Adicionalmente, até então, havíamos trabalhado em nossos programas apenas com o tipo strings, codificando-as e decodificando-as para enviar e receber, respectivamente, via rede. Na aula de hoje, veremos como enviar mensagens personalizadas, envolvendo qualquer tipo de dado, bem como implementar técnicas de *framing* (enquadramento).

Tudo que será tratado aqui servirá para UDP. As técnicas de framing, entretanto, terão mais utilidade em Sockets TCP.

Bom proveito!

Preparando dados para transmissão

Até então, tínhamos programado nossos Clientes/Servidores para trocar e processar mensagens simples. Em geral, eram mensagens baseadas no tipo de dado string, muito útil em algumas aplicações. Porém outras aplicações vão exigir a troca e o processamento de outros tipos de dados.

Já reiteramos diversas vezes que só podemos enviar e receber o tipo bytes via rede. E por causa disso, aliado à facilidade de realizar a conversão entre string -> bytes e bytes -> string, vínhamos usando o tipo string em nossos programas. Conforme vimos, basta invocarmos os métodos, encode e decode, respectivamente. Como esse já foi um assunto bastante presente em materiais anteriores, vamos voltar a nossa atenção em como podemos trabalhar com outros tipos de dados com Sockets.

O módulo struct

Python nos oferece um módulo chamado struct que nos permite transformar qualquer tipo de dado nativo básico (inteiro, float e string) em bytes. O módulo oferece a possibilidade de trabalharmos com funções, bem como com a possibilidade de trabalharmos com a classe struct. Ambas as possibilidades são válidas e vão funcionar do mesmo jeito. Por questão didática, neste material iremos trabalhar apenas com as funções do módulo.

Em essência, precisaremos de duas funções:

- da função **pack**: usada para codificar os dados passados para bytes
- e da função **unpack**: usada para, de posse dos bytes, obter os dados originais

Ambas as funções recebem como primeiro argumento uma string de formatação, uma espécie de string de controle que instrui a função como construir a sequência de bytes (no caso da pack) ou interpretar a sequência de bytes (no caso da unpack).

O primeiro caractere da string de formatação serve para indicar o padrão de ordem dos bytes. Observe a tabela:

Caractere	Significado
<	little-endian
>	big-endian
!	network (=big-endian)

Tabela 1 - Padrão de ordem dos bytes

O padrão little-endian ordena os bytes colocando os bytes menos significativos mais à esquerda. O padrão big-endian ordena os bytes colocando os bytes mais significativos mais à esquerda. Podemos usar qualquer um dos padrões em nossas aplicações. Mas se desejarmos escrever uma aplicação que se comunica com outra já existente, ou se pretendermos atender ao padrão de redes, devemos usar o padrão big-endian. Para evitar confusão e esquecimento, podemos usar sempre o caractere ! (exclamação), já que estamos usando o módulo struct com vista em gerar bytes para transmitir via rede.

Em seguida, devemos preencher a string de formatação com os caracteres de formato. Cada caractere corresponde a um tipo e produzirá um tamanho fixo de bytes. Observe a tabela abaixo (não exaustiva):

Caractere	Significado	Tipo Python	Tamanho em bytes
c	Um caractere	bytes	1
?	Booleano	bool (True, False)	1
i	Inteiro	integer	4

I	Inteiro sem sinal		4
f	float	float	4
d	double		8
s	string	bytes	-

Tabela 2 - Caracteres de formato

De posse das informações acima, podemos realizar alguns experimentos diretamente no interpretador Python.

Em cada exemplo abaixo, considere que o módulo struct já foi importado:

```
>>> import struct
```

Exemplo 1:

```
>>> struct.pack("!ici",2,b"+",2)
b'\x00\x00\x00\x02+\x00\x00\x00\x02'
>>> struct.unpack("!ici",b'\x00\x00\x00\x02+\x00\x00\x00\x02')
(2, b'+', 2)
```

No exemplo acima, configuramos a string de formatação para enviar dados via rede (!), codificar um inteiro (i), seguido de um caractere (c), seguido de outro inteiro (i). Na função pack, a string de formatação é sucedida dos argumentos seguindo a ordem dada na própria string de formatação. Assim, passamos primeiro o inteiro 2, um bytes b"+", e outro inteiro 2. Note que o caractere deve ser passado no formato bytes.

Na função unpack usamos a mesma string de formatação, seguida dos bytes "recebidos". Note que a função retorna uma tupla, onde cada elemento é um dado extraído. A função unpack sempre retorna uma tupla, mesmo que só tenha um dado a ser extraído.

Exemplo 2:

```
>>> struct.pack("!f",9.5)
b'A\x18\x00\x00'
>>> struct.unpack("!f",b'A\x18\x00\x00')
(9.5,)
```

No exemplo acima resolvemos codificar apenas um float. Note que a função unpack retornou uma tupla com apenas um elemento.

Exemplo 3:

```
>>> struct.pack("!3?",True,False,True)
b'\x01\x00\x01'
>>> struct.unpack("!???",b'\x01\x00\x01')
```

```
(True, False, True)
```

Caso queira ler o mesmo tipo em sequência, você pode indicar isso na string de formatação, precedendo o formato por um número. Isso serve para qualquer tipo.

Exemplo 4:

```
>>> struct.pack("!6s","Python".encode())
b'Python'
>>> struct.unpack("!6s",b'Python')
(b'Python',)
```

Para ler strings você deverá indicar o formato com s e preceder pelo total de bytes da string. A string deve ser passada em bytes. Mas muito cuidado! Atenção para um detalhe! Caracteres diferentes da tabela ASCII (acentuados, por exemplo) irão produzir mais de um byte por caractere.

Veja:

```
>>> struct.pack("!6s","Pýthon".encode())
b'P\xc3\xbdtho'
>>> dados = struct.unpack("!6s",b'P\xc3\xbdtho')
>>> dados
(b'P\xc3\xbdtho',)
>>> dados[0].decode()
'Pýtho'
```

Mas o que houve? Pýthon tem exatos 6 caracteres! Exatamente, mas, por existir um deles acentuado, a codificação resultante acaba tendo mais bytes que o tamanho da string.

Podemos solucionar esse problema contando os bytes da string codificada antes de empacotar. Dá-le Python!

Exemplo 5:

```
>>> string = "Pýthon".encode()
>>> struct.pack("!{}s".format(len(string)),string)
b'P\xc3\xbdthon'
>>> dados =
struct.unpack("!{}s".format(len(string)),b'P\xc3\xbdthon')
>>> dados[0].decode()
'Pýthon'
```

Primeiro atribuímos à variável string o texto “Pýthon” codificado. Invocamos o método format para substituir “{}” na string de formatação pela quantidade de bytes. Agora tudo funcionou como esperado.

Abra o interpretador Python em um terminal e repasse os exemplos acima.

O módulo pickle

Se você gostou do módulo struct, vai se apaixonar pelo módulo pickle.

Como vimos, o módulo struct permite transformar os tipos básicos de dados do Python em uma cadeia de bytes. Isso é extremamente importante quando estamos desenvolvendo um Cliente em Python, por exemplo, para se comunicar com um Servidor desenvolvido em C. Mas pode acontecer de estarmos desenvolvendo tanto o Servidor quanto o Cliente em Python. Numa situação como essa, seria muito conveniente se pudéssemos transmitir outros tipos Python mais complexos, como Listas, Dicionários, Tuplas etc.

Pois bem, isso é possível através do módulo pickle. Além de ser possível transmitir tipos mais complexos, o uso do pickle é ainda mais simples.

Só vamos precisar de duas funções:

- da função **dumps**: usada para serializar os dados em bytes
- e da função **loads**: usada para, de posse dos bytes, obter os dados originais

Em cada exemplo abaixo, considere que o módulo pickle já foi importado:

```
>>> import pickle
```

Exemplo 1:

```
>>> pickle.dumps(1)
b'\x80\x03K\x01.'
>>> pickle.loads(b'\x80\x03K\x01.')
1
>>> pickle.dumps(True)
b'\x80\x03\x88.'
>>> pickle.loads(b'\x80\x03\x88.')
True
>>> pickle.dumps("Pýthon")
b'\x80\x03X\x07\x00\x00\x00P\xc3\xbdthonq\x00.'
>>>
pickle.loads(b'\x80\x03X\x07\x00\x00\x00P\xc3\xbdthonq\x00.')
'Pýthon'
```

No exemplo acima, ilustramos o uso do pickle com os tipos básicos. Note que não foi preciso indicar o tipo passado. O próprio Pickle se encarrega disso.

Exemplo 2:

```
>>> pickle.dumps([[1,2],{"chave":"value"}])
```

```
b'\x80\x03]q\x00(]q\x01(K\x01K\x02e}q\x02X\x05\x00\x00\x00chav
eq\x03X\x05\x00\x00\x00valueq\x04se.'
>>>
pickle.loads(b'\x80\x03]q\x00(]q\x01(K\x01K\x02e}q\x02X\x05
\x00\x00\x00chaveq\x03X\x05\x00\x00\x00valueq\x04se.')
[[1, 2], {'chave': 'value'}]
```

Neste exemplo passamos uma lista, cujo primeiro elemento é outra lista e o segundo elemento é um dicionário. Usando o pickle não houve segredo nenhum.

Framing

Como já mencionado, se você estiver usando UDP na comunicação, o próprio protocolo se encarregará de enviar os dados em blocos discretos e identificáveis. Porém, se estiver usando TCP, que trabalha por fluxo, deverá se preocupar em como delimitar suas mensagens de modo que o destinatário saiba onde uma mensagem termina e uma nova começa. É por essa última razão que agora vamos estudar técnicas de *framing*.

Abordagens

Há várias abordagens. Vejamos algumas:

1. Se um host apenas envia e o outro apenas recebe, nesse caso o emitente pode enviar todos os dados com sendall e o destinatário poderá chamar repetidamente recv até a chamada retornar uma string vazia, indicando que o emitente encerrou o socket.
2. Se um host primeiro envia para só depois receber, nesse caso temos uma variante da abordagem anterior. O emitente pode enviar todos os dados com sendall e, em seguida, fechar o socket na direção de ida. Em seguida, o mesmo procedimento deverá ocorrer do outro lado da rede.
3. Usar mensagens de tamanho fixo.
4. Delimitar as mensagens com caracteres ou bytes especiais. Nesse caso, o destinatário deverá chamar recv até encontrar o caractere delimitador.
5. Prefixar cada mensagem com seu tamanho em bytes.
6. Prefixar blocos da mensagem com seu tamanho em bytes. Esse caso é uma variante do anterior, no caso de o emitente não conhecer de antemão o tamanho total da mensagem a ser transmitida. O emitente pode sinalizar o fim da transmissão através do envio de um bloco de tamanho zero.

Implementando as abordagens

A estrutura do código utilizada em cada abordagem é semelhante à estrutura que vimos na aula anterior. Portanto, os comentários serão voltados para o que realmente importa para esta aula.

Obs: Nos códigos abaixo, foi suprimido a parte que invoca a função main.

Abordagem 1

Nesta primeira abordagem vamos simular uma situação em que o Servidor apenas recebe os dados climáticos de um Cliente. A única função do Cliente é, de posse dos dados, enviar tudo para o Servidor.

Servidor:

```
import socket
import pickle

def recvall(sock):
    data = b""
    while True:
       more = sock.recv(1024)

       if not more:
            break
       data += more
    return data

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()
       sc.shutdown(socket.SHUT_WR)

       msg = recvall(sc)
       print("Dados recebidos: ",pickle.loads(msg))
       sc.close()

    return 0
```

O início da main já deve ser conhecido por você. Dentro do loop while, logo após que o Servidor atende a conexão que chega, invocamos o método shutdown. Nesse ponto é importante se lembrar de que o protocolo TCP é bidirecional e full-duplex. Isso quer dizer que, podemos ter transmissão indo e vindo ao mesmo tempo. Mas pode

acontecer de querermos apenas enviar dados ou apenas receber. O método shutdown nos permite fechar apenas uma das direções (ou ambas).

O método shutdown recebe um dos 3 argumentos:

- SHUT_WR: usado para desativar a saída de dados, desativar envios
- SHUT_RD: usado para desativar a chegada de dados, desativar recebimentos
- SHUT_RDWD: usado para desativar ambas as direções

Entendido isso, como nosso Servidor apenas recebe o dado do Socket desativamos sua saída. Esse método não é necessário nessa abordagem, mas atua como medida protetiva.

Depois invocamos a função recvall. Note que a função foi implementada sem receber o total de bytes a ser recebido do host remoto. Analisando a função vemos que ela invoca recv infinitamente até que não tenha mais dados no buffer de entrada. Essa situação ocorrerá quando o Cliente encerrar o Socket do outro lado.

Note que usamos o módulo pickle. Simplesmente invocamos a função loads para interpretar msg.

Cliente:

```
import socket
import pickle

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

    print(" Socket: ",sock.getsockname())

    msg = {"Dia 1":"30C","Dia 2":"29C", "Dia 3":"28C"}
    sock.sendall(pickle.dumps(msg))

    sock.close()

    return 0
```

Aqui no Cliente também não há segredos. Invocamos shutdown para encerrar o fluxo de entrada. Depois criamos uma msg com os dados climáticos e enviamos tudo. Note que os bytes foram gerados usando pickle, mais precisamente a função dumps. Logo após enviar tudo, fechamos o socket. Quando o socket é fechado, qualquer tentativa de leitura no buffer de entrada do host remoto vai resultar em uma leitura vazia (indicando fim de arquivo).

Abordagem 2

Nesta segunda abordagem vamos simular um Servidor que realiza o serviço de ordenação de uma Lista. O Cliente envia os bytes e encerra o fluxo fechando a conexão de ida. O Servidor recebe os dados, processa e envia a resposta ao Cliente.

Servidor:

```
import socket
import pickle

def recvall(sock):
    data = b""
    while True:
       more = sock.recv(1024)

       if not more:
            break
       data += more
    return data

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()

       msg = recvall(sc)
       lista = sorted(pickle.loads(msg))

       sc.sendall(pickle.dumps(lista))
       sc.shutdown(socket.SHUT_WR)

       sc.close()

    return 0
```

Por ser uma variante da abordagem anterior, o código ficou semelhante, inclusive a função recvall é a mesma. No atendimento da conexão, o Servidor recebe todo o fluxo

de entrada, interpreta os bytes com pickle (loads), ordena a Lista (função sorted) e envia os bytes (dumps) da Lista ordenada. Por fim, encerra a direção de envio e encerra o socket.

Cliente:

```
import socket
import pickle

def recvall(sock):
    data = b""
    while True:
       more = sock.recv(1024)

       if not more:
            break
       data += more
    return data

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

    print(" Socket: ",sock.getsockname())

    msg = [100,200,600,0,400,1000,250,900]
    print("Lista enviada: ", msg)
    sock.sendall(pickle.dumps(msg))
    sock.shutdown(socket.SHUT_WR)

    res = recvall(sock)
    print("Lista recebida: ", pickle.loads(res))

    sock.close()

    return 0
```

Também muito semelhante ao código do Cliente da abordagem anterior. O Cliente armazena a Lista desordenada em msg. Converte tudo em bytes (dumps), envia e fecha o fluxo de saída (shutdown). Em seguida, invoca recvall para obter todos os bytes enviados. Carrega a lista (loads) e exibe na tela.

Abordagem 3

Nesta terceira abordagem vamos simular um Servidor que realiza o serviço de soma de dois inteiros. Como um inteiro ocupa 4 bytes, conseguimos transmitir e receber as mensagens considerando um tamanho fixo.

Servidor:

```
import socket
import struct

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 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()

      msg = recvall(sc,8)
      nums = struct.unpack("!ii",msg)

      sc.sendall(struct.pack("!i",nums[0]+nums[1]))

      sc.close()

    return 0
```

Note que a função recvall do código acima foi implementada na aula passada. Ela recebe a quantidade de bytes a ser lido do buffer de entrada por recv.

Na função main, no atendimento da conexão, passamos 8 bytes para recvall, já que queremos receber dois inteiros de 4 bytes. Dessa vez, utilizamos o módulo struct para interpretar (unpack) e converter (pack) em bytes nossos dados. Atenção para a string de formatação. Usamos “!ii” para receber dois inteiros. Usamos “!i” para enviar um inteiro. Fizemos nums[0] + nums[1], pois os números recebidos ficam armazenados em uma tupla.

Cliente:

```
import socket
import struct

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

    print(" Socket: ",sock.getsockname())

    msg = struct.pack("!ii",100,200)
    print("Enviando dados (100,200) para servidor somar.")
    sock.sendall(msg)

    res = recvall(sock,4)
    print("Resultado da soma: ", struct.unpack("!i",res)[0])

    sock.close()

    return 0
```

A função recvall é a mesma do Servidor. Em main, o Cliente converte dois números em bytes (pack) e os envia ao Servidor. Em seguida, recebe 4 bytes que serão interpretados como um inteiro resultante da soma e o exibe na tela.

Abordagem 4

Nesta quarta abordagem quero apenas demonstrar a possibilidade de usar um caractere (byte) como delimitador para as mensagens de tamanho variável.

Servidor:

```
import socket

def recvall(sock, delim):
    data = b""
    while True:
      more = sock.recv(1)

      if more == delim:
           break
      data += more
    return data

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()

      print(recvall(sc,b"\n").decode())
      sc.sendall("É sim!\n".encode())

      print(recvall(sc,b"\n").decode())
      sc.sendall("É sim!\n".encode())

      print(recvall(sc,b"\n").decode())
      sc.sendall("É sim!\n".encode())

      sc.close()

    return 0
```

O Servidor recebe três mensagens do Cliente e responde com uma resposta padrão. As mensagens têm tamanhos diferentes. Usamos o caractere “\n” como delimitador das mensagens.

O coração do código está na função recvall. Invocamos recv para ler byte a byte em um loop infinito, e encerramos quando o byte delimitador é lido.

Cliente:

```
import socket

def recvall(sock, delim):
    data = b""
    while True:
      more = sock.recv(1)

      if more == delim:
           break
      data += more
    return data

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

    print(" Socket: ",sock.getsockname())

    sock.sendall("Python é bom demais.\n".encode())
    print(recvall(sock,b"\n").decode())

    sock.sendall("Sockets é bom demais.\n".encode())
    print(recvall(sock,b"\n").decode())

    sock.sendall("Framing é bom demais.\n".encode())
    print(recvall(sock,b"\n").decode())

    sock.close()

    return 0
```

Acredito que, com as explicações do Servidor, o código do Cliente esteja óbviu.

Abordagem 5 e 6

Decidi unir as duas últimas abordagens por serem bastante semelhantes em termos de codificação. Implementamos aqui um velho serviço conhecido em nossa disciplina: um Servidor que converte em maiúsculas tudo que é enviado pelo Cliente.

Servidor:

```
import socket
import struct

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 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()

      msg = recvmsg(sc)
      sendmsg(sc,msg.decode().upper().encode())
```

```
        sc.close()

    return 0
```

A estratégia desta abordagem consiste em transmitir primeiro o tamanho da mensagem para depois transmitir a mensagem. Para realizar este intento aproveitamos a função recvall, que recebe a quantidade de bytes a serem lidos, e criamos mais duas funções: sendmsg e recvmsg.

A função sendmsg calcula o tamanho da msg (len) e usa struct.pack para converter esse tamanho (inteiro) em bytes. Então ela primeiro envia esses bytes para depois enviar a mensagem (que já está em bytes)..

A função recvmsg trabalha de forma espelhada. Primeiramente ela lê 4 bytes do buffer de entrada, interpretando-os com struct.unpack. Porque unpack retorna sempre uma tupla, pegamos o primeiro elemento. Em seguida, lemos a quantidade de bytes em função do tamanho recém-recebido.

Cliente:

```
import socket
import struct

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 main():
```

```
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(('127.0.0.1',50000))

    print(" Socket: ",sock.getsockname())

    msg = input("Entre com uma mensagem:").encode()
    sendmsg(sock,msg)

    res = recvmsg(sock)
    print("Servidor Responde: ", res.decode())

    sock.close()

    return 0
```

O Cliente usa uma estratégia bem semelhante ao Servidor. Portanto, dispensamos os comentários.

Chegamos ao final de mais uma aula. 🤗

Nesta aula estudamos como podemos enviar tipos de dados variados via rede. Adicionalmente, vimos algumas estratégias sobre como implementar *framing*. Agora você está pronto para transmitir o que quiser por meio de uma rede, seja através de Sockets UDP ou Sockets TCP.

Na aula que vem, daremos continuidade ao tema “Preparando dados para transmissão”, onde veremos alguns padrões universais (JSON, CSV e XML) que podem ser empregados na transmissão de dados, e também como podemos compactar os dados a serem transmitidos.

Bons estudos e até a próxima!