LorenzoDomenichetti_dataManagement

June 12, 2021

1 Data Management Assignment

1.1 Lorenzo Domenichetti - 2011653

[1]:
```
import os
import time
import requests
import numpy as np
import pandas as pd
from functools import reduce
import matplotlib.pyplot as plt

requests.packages.urllib3.disable_warnings()
```

1.1.1 Fun exercise

1+2 = 3

2+5 = 7

3+7 = 4

4+5 = ?

5+9 = 12

The operation between these numbers of course is the XOR! The answer was truly straight-forward after all the lessons on parity, checksums, redundancy… and error-correction algorithms.

Let's show the proof.

Let's convert the numbers in binaries - I'll keep only the first bits for ease.

1→ 001, 2→ 010, 5→ 101;

1+2 → 001^010 = 011 → can be converted into 3.

2+5 → 010^101 = 111 → can be converted into 7.

3+7 → 011^111 = 100 → can be converted into 4.

4+5 → 100^101 = 001 → can be converted into 1.

5+9 → 0101^1001 = 1100 → can be converted into 12.

[2]:
```
#Or, defining a function..
def XOR(a,b):
    '''
    The XOR function takes as arguments two integer and returns their bitwise␣
 ↪xor.
    '''
    return a ^ b

print(f"The XOR between 1 and 2: {XOR(1,2)}")
print(f"The XOR between 2 and 5: {XOR(2,5)}")
print(f"The XOR between 3 and 7: {XOR(3,7)}")
print(f"The XOR between 4 and 5: {XOR(4,5)}")
print(f"The XOR between 5 and 9: {XOR(5,9)}")
```

```
The XOR between 1 and 2: 3
The XOR between 2 and 5: 7
The XOR between 3 and 7: 4
The XOR between 4 and 5: 1
The XOR between 5 and 9: 12
```

1.2 1. Redundancy

We are programming a file based RAID-4 software algorithm.

For this purpose we are converting a single input (raid4.input) file into four data files raid4.0,raid4.1,raid4.2,raid4.3 and one parity file raid4.4 - the four data and one parity file we call 'stripe files'.

The input file can be downloaded from: http://apeters.web.cern.ch/apeters/pd2021/raid4.input

To do this we are reading in a loop sequentially blocks of four bytes from the input file until the whole file is read:

- in each loop we write one of the four read bytes round-robin to each data file, compute the parity of the four input bytes and write the result into the fifth parity file. (see the drawing for better understanding)

- we continue until all input data has been read. If the last bytes read from the input file are not filling four bytes, we consider the missing bytes as zero for the parity computation.

1.1 Write a program (C,C++, R or Python), which produces four striped data and one parity file as described above using the given input file.

1.2 Extend the program to compute additionally the parity of all bytes within one stripe file.

You can say, that the computed column-wise parity acts as a **CHECKSUM** for each stripe file.

Compute the size overhead by comparing the size of all 5 stripe files with the original file. The size overhead is **25** % !

As expected: we split the initial file in four chunks (each 25% of the initial one). Then we add another chunk that contains parity, and, as computed, it has the same dimension as the previous chunks themselves. So that is why the overhead is 25%.

1.2.1 The q vector acts as a checksum for all stripes.

[3]:
```
#vector containing file_names.
vec_file = ["file4.0", "file4.1", "file4.2", "file4.3", "file4.4"]

#empty checksum vector, updated at each iteration sequentially.
q = [0] * 5

with open("raid4.input", "rb") as file:

    #read 4 bytes blocks in a loop sequentially.
    while (byte := (file.read(4))):

        #check if four or less bytes are read.
        #in case less than four bytes, fill with zeros.
        while(len(byte)<4):
            byte = bytearray(byte)
            byte.append(0)
            byte = bytes(byte)

        #print data on file four by four. Update the checksum.
        for it in range(4):
            with open(vec_file[it], 'ab') as write_file:
                write_file.write(chr(byte[it]).encode('latin-1'))
            q[it] = q[it]^byte[it]

        #parity
        res = ((reduce(lambda i,j: (i^j), byte)))

        #print parity on fifth file. Update the checksum.
        with open(vec_file[4], 'ab') as write_file:
            write_file.write(chr(res).encode('latin-1'))
        q[4] = q[4]^res
```

[4]:
```
files_size = 0
for j in vec_file:
    files_size+=os.path.getsize(j)

overhead = files_size/os.path.getsize('raid4.input')
print(f'The final files summed are {overhead:.2f} times the initial one. The␣
 ↪actual overhead is {(overhead-1)*100:.0f}%')

#print('The actual size is 1.25 times, so 25% overhead')
```

```
The final files summed are 1.25 times the initial one. The actual overhead is
25%
```

1.3 What is the 5-byte parity value if you write it it in hexadecimal format like P5 =0x[q0][q1][q2][q3][q4], where the [qx] are the hexadecimal parity bytes computed by xor-ing all bytes in each stripe file. A byte in hexadecimal has two digits and you should add leading 0 if necessary.

[5]:
```
lista = [str(hex(k))[2:] for k in q]
for it in range(len(lista)):
    if(len(lista[it])!=2):
        lista[it] = '0'+ lista[it]

checksum = '0x '+' '.join(lista)
print(f"The final parity value in hexadecimal format: {checksum}")
```

```
The final parity value in hexadecimal format: 0x a5 07 a0 9c 9e
```

1.4a If you create a sixth stripe file, which contains the row-wise parities of the five stripe files, what would be the contents of this file?

1.4b Write down the equation for R, which is the XOR between all data stripes D0,D1,D2,D3 and the parity P. Remember P was the parity of D0,D1,D2,D3! Reduce the equation removing P from it to get the answer about the contents!

If we compute again the parity between all columns and their parity, the result is going to be zero for each row.

Let's show the proof:

$$R = D0 \oplus D1 \oplus D2 \oplus D3 \oplus P = D0 \oplus D1 \oplus D2 \oplus D3 \oplus (D0 \oplus D1 \oplus D2 \oplus D3)$$

Now we can exploit the commutative and associative properties of the XOR. $a \oplus (b \oplus c) = (a \oplus b) \oplus c = (a \oplus c) \oplus b$.

Such proposition follows directly from the XOR definition, and are intuitive looking at XOR's truth table.

$$R = (D0 \oplus D0) \oplus (D1 \oplus D1) \oplus (D2 \oplus D2) \oplus (D3 \oplus D3)$$

Now we may want again to exploit a property of the XOR operation. Given any X, it holds X^X = 0 . The proof of such proposition follows again from the binary representation - X's bits are all equal, and the bitewise XOR returns always a zero bit.

So, finally, $R = 0 \oplus 0 \oplus 0 \oplus 0 = 0$.

Let's check it. - For the sake of simplicity, instead of removing P one could have noticed that the first four XORs again yield P, and $P \oplus P = 0$.

[6]:
```
double_xor = []
with open("raid4.input", "rb") as file:
    while (byte := (file.read(4))):
        while(len(byte)<4):
            byte = bytearray(byte)
```

```
            byte.append(0)
            byte = bytes(byte)

        res = ((reduce(lambda i,j: (i^j), byte)))
        #d_xor = res^res  - simpler..
        d_xor = res
        for it in range(4):
            d_xor = d_xor^byte[it]

        #appending XOR of res(fifth column) and the original four bytes.
        double_xor.append(d_xor)

#Checking our proposition is correct
double_xor == [0] * len(double_xor)
```

[6]: True

1.5 After some time you recompute the 5-byte parity value as in 1.3. Now the result is P 5 = 0x a5 07 a0 01 9e. Something has been corrupted. You want to reconstruct the original file raid4.input using the 5 stripe files.

Describe how you can recreate the original data file. Which stripe files do you use and how do you recreate the original data file with the correct size?

Original checksum: 0x a5 07 a0 9c 9e

New checksum: 0x a5 07 a0 01 9e

First - we need to find which column has been corrupted. In this case, the checksum yield the uncorrect result in the fourth column (9c → 01) - so we may conclude that the corrupted data lay in the fourth column.

The fourth column then has to be discarded. Fortunately, only one column seems to be corrupted. So, with a single parity, we are still able to reconstruct the initial file. The procedure is the following: as we want to reconstruct the lost column, we take the remaining four (1,2,3,5). Now, thanks to the properties of the XOR, we can reconstruct the column element by element taking the XOR of the row elements of the four columns that return the right checksum. The result of this operation will be then the original value.

With this approach one is finally able to reconstruct the initial data, and the checksum should finally return to the original value. If two or more columns would have been corrupted, we would have not been able to use this simple algorithm to reconstruct the original data.

$$D3 = D0 \oplus D1 \oplus D2 \oplus P$$

Let's check it.

[7]:
```
fourth_original = []
fourth_reconstructed = []
with open("raid4.input", "rb") as file:
```

```
    while (byte := (file.read(4))):
        while(len(byte)<4):
            byte = bytearray(byte)
            byte.append(0)
            byte = bytes(byte)

        res = ((reduce(lambda i,j: (i^j), byte))) #parity

        recover_xor = res
        for it in range(3):
            recover_xor = recover_xor^byte[it] #computing the XOR between the␣
 ↪first three and the parity entries.

        fourth_reconstructed.append(recover_xor)
        fourth_original.append(byte[3])

#Checking our proposition is correct. Check if the two list are equal element␣
 ↪by element.
print(fourth_original == fourth_reconstructed)
print(fourth_original[:5])
print(fourth_reconstructed[:5])
```

```
True
[70, 51, 229, 167, 196]
[70, 51, 229, 167, 196]
```

1.3 2. Cryptography

The Caesar cipher is named for Julius Caesar, who used an alphabet where decrypting would shift three letters to the left.

A friend has emailed you the following text: K]amua!trgpy She told you that her encryption algorithm works similar to the Caesar cipher:

- to each ASCII value of each letter I add a secret key value. (note that ASCII values range from 0 to 255)

- additionally to make it more secure I add a variable (so called) nonce value to each ASCII number.

The nonce start value is 5 for the first character of the message. For each following character add 1 to the nonce of the previous character, e.g. for the second letter the nonce added is 6, for the third letter it is 7 aso.

2.1 Is this symmetric or asymmetric encryption and explain why?

The encryption is symmetric - ONE single key is used for both encrypting and decriptying the secret message. And this is exactly the definition of symmetric encryption. In asymmetric encryption, instead, there would have been differnt keys for encoding and decoding. In this case, as will be shown, an algorithm can be designed for decoding the initial message using the same keys (nonce start value and *"the secret key"*) used for encryption.

Moreover, there are no public nor private keys neiter for sender nor receiver - as it happens in asymmetric encryption.

2.2 Write a small brute force program which tests keys from 0..255 and use a dictionary approach to figure out the original message.

The **used key is 246**. The original message text is **Padova rocks**.

[8]:
```
message = 'K]amua!trgpy'
#conversion to integer and subtraction of the "nonce" values
to_int = [ord(k) for k in message] - np.arange(5,5+len(message))

#original message
original_message = ''

#message key
key = -1

for i in range(256):
    #subtracting the values one by one to the original string - bruteforce␣
 ↪algorithm.
    dec = to_int - i

    #ASCII values only range from 0 to 255. If in the subtraction a negative␣
 ↪numeber is found,
    # the count restarts from the biggest value possible, i.e. 255.
    dec[dec < 0 ] = 256 + dec[dec < 0 ]

    #interpreting the dictionary_approach as "the string that makes sense is␣
 ↪the right one"!
    if(chr(dec[0])=='P'):
        original_message = ''.join(([chr(k) for k in dec]))
        key = i
        break


print(f"As our friend told us that the encryption algorithm includes sums, I am␣
 ↪assuming the decryption algorithm works by subtraction.\nThis way we can␣
 ↪really use the same keys (246, 5-nonce) with slightly different algorithms␣
 ↪(subtractions instead of sums).\n")
print(f"The original message: {original_message}\n")
```

```
As our friend told us that the encryption algorithm includes sums, I am assuming
the decryption algorithm works by subtraction.
This way we can really use the same keys (246, 5-nonce) with slightly different
algorithms (subtractions instead of sums).

The original message: Padova rocks
```

2.3 What is the decryption algorithm/formula to be used?

The decryption algoritm follows the following steps:

1) convert the initial characters to their corresponding ASCII;
2) subtract the nonce values, ranging from 5 to 5+length(message), accordingly to the position in the array;
3) subtract to the key (246 in our case) to the obtained values. Mind that negative numbers have to be converted to numbers on top of the ASCII range.
4) convert the so obtained integers into the corresponding characters and join them into a single string.

1.4 3. Object Storage

In an object storage system we are mapping objects by name to locations using a hash table. Imagine we have a system with ten hard disks (10 locations). We enumerate the location of a file using an index of the hard disk [0..9].

Our hash algorithm for placement produces hashes, which are distributed uniform over the value space for a flat input key distribution. We want now to **simulate the behaviour of our hash algorithm without the need to actually compute any hash value.**

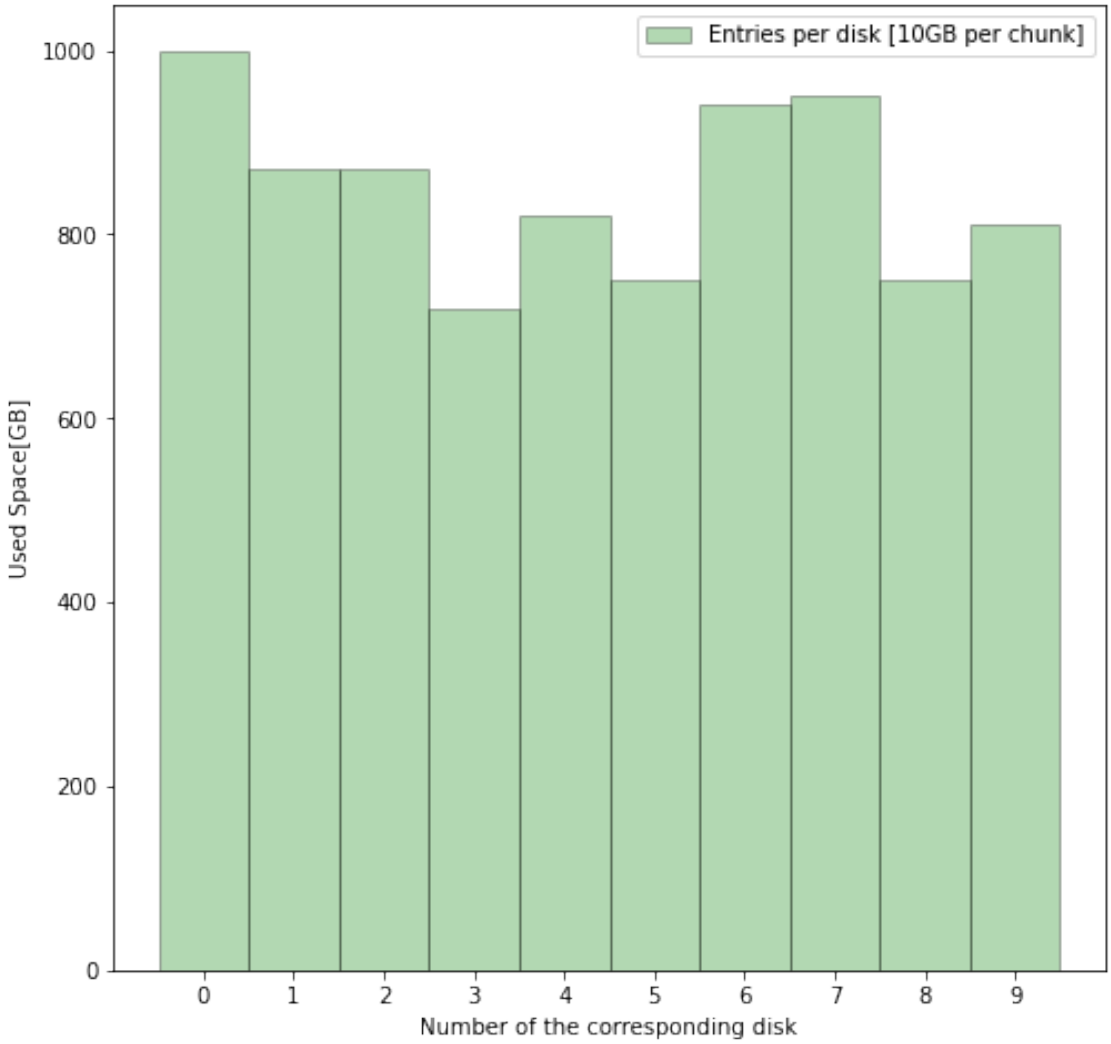
Instead of using real filenames, which we would hash and map using a hash table to a location (as we did in the exercise), we are 'computing' a **location** for 'any' file by **generating a random number for the location** in the range [0..9] to assign a file location. To place a file in the storage system we use this random location where the file will be stored and consumes space.

Assume each disk has 1TB of space, we have 10TB in total. Place as many files of 10GB size as possible to hard disks choosing random locations until one hard disk is full. Hint: a hard disk is full once you have stored hundred 10GB files.

3.1 Write a program in Python, R or using ROOT, which simulates the placement of 10GB files to random locations and account the used space on each hard disk. Once the first hard disk is full, you stop to place files. Possibly visualise the distribution.

[9]:
```
# as suggested, we'll keep 1000 as ratio between different data sizes (MB, GB,␣
 ↪TB), instead of 1028...
counts = [0]*10
x = np.arange(10)
np.random.seed(123456)


for i in np.random.randint(0,10,1000):
    counts[i] += 10
    if(counts[i] == 1000):
        break

plt.figure(figsize = (8,8))
```

```
plt.hist(x, bins = range(11), align='left',  weights = counts, alpha = 0.3,␣
 ↪color = 'green',
         histtype='bar', label = 'Entries per disk [10GB per chunk]',␣
 ↪ec='black')
plt.xlabel("Number of the corresponding disk")
plt.ylabel("Used Space[GB]")
plt.legend()
plt.xticks(np.arange(0,10))
plt.show()
```



3.1a How many files did you manage to place?

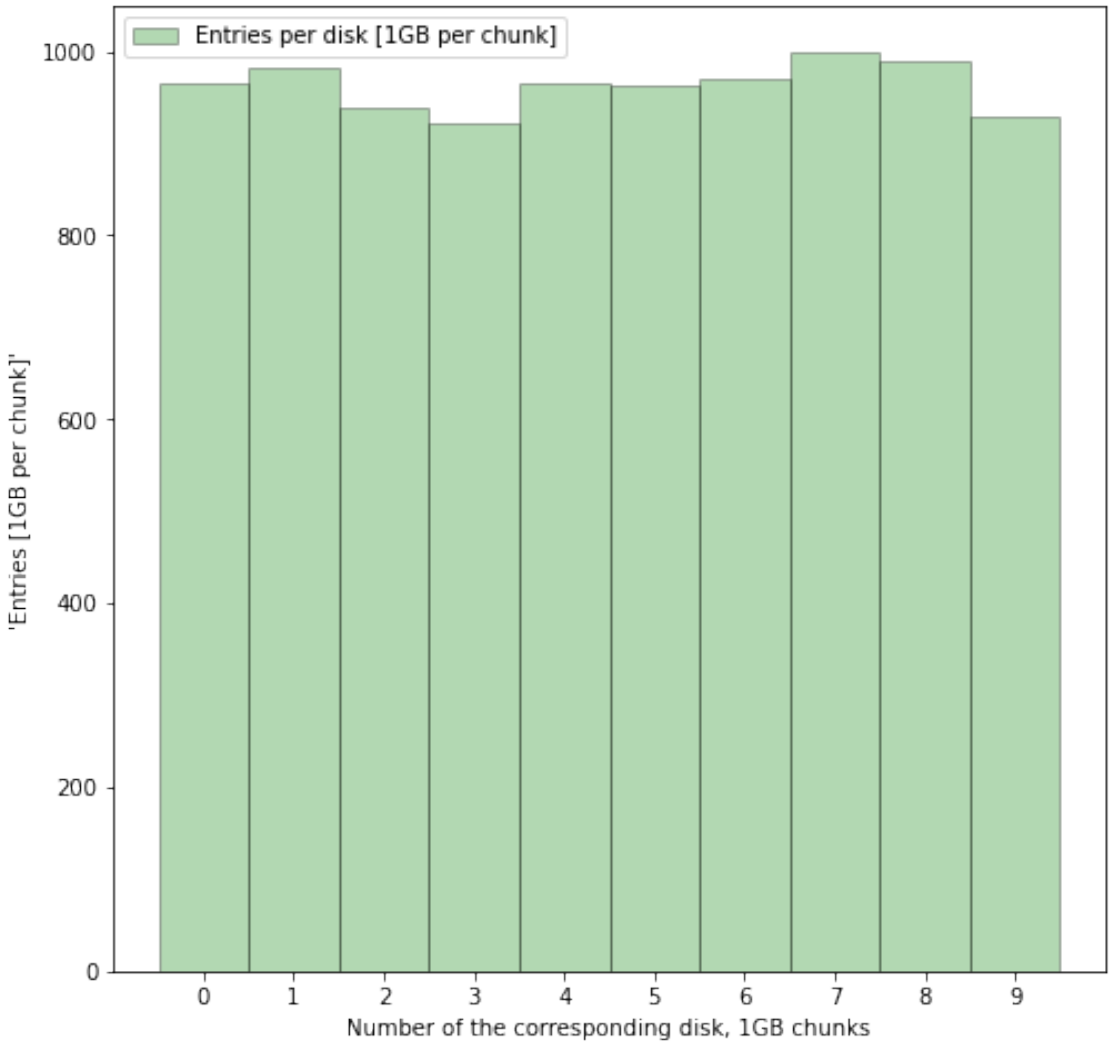
3.1b What is the percentage of total used space on all hard disks in the moment the first disk is full?

[10]:
```
print(f"The number of files placed: {sum(counts)/10:.0f}")
print(f"The total amount of space available is 10 x 1TB and we have placed␣
 ↪{sum(counts)/10:.0f} x 10GB files.",
      f"\nSo we have used {sum(counts)/1000}TB of the 10TB available. The␣
 ↪percentage: {sum(counts)/100}%")
```

```
The number of files placed: 848
The total amount of space available is 10 x 1TB and we have placed 848 x 10GB
files.
So we have used 8.48TB of the 10TB available. The percentage: 84.8%
```

3.2 Repeat the same task placing 1GB files until the first hard disk is full.

[11]:
```
#In case of 1GB files, we need to have 1000 files to fill a 1TB disk.
counts_1GB = [0]*10
np.random.seed(123456)

for i in np.random.randint(0,10,10000):
    counts_1GB[i] += 1
    if(counts_1GB[i] == 1000):
        break

plt.figure(figsize = (8,8))
plt.hist(x, bins = range(11), align='left',  weights = np.array(counts_1GB),␣
 ↪alpha = 0.3, color = 'green',
         histtype='bar', label = 'Entries per disk [1GB per chunk]', ec='black')
plt.xlabel("Number of the corresponding disk, 1GB chunks")
plt.ylabel("'Entries [1GB per chunk]'")
plt.legend()
plt.xticks(np.arange(0,10))
plt.show()
```



3.2a How many files did you manage to place?

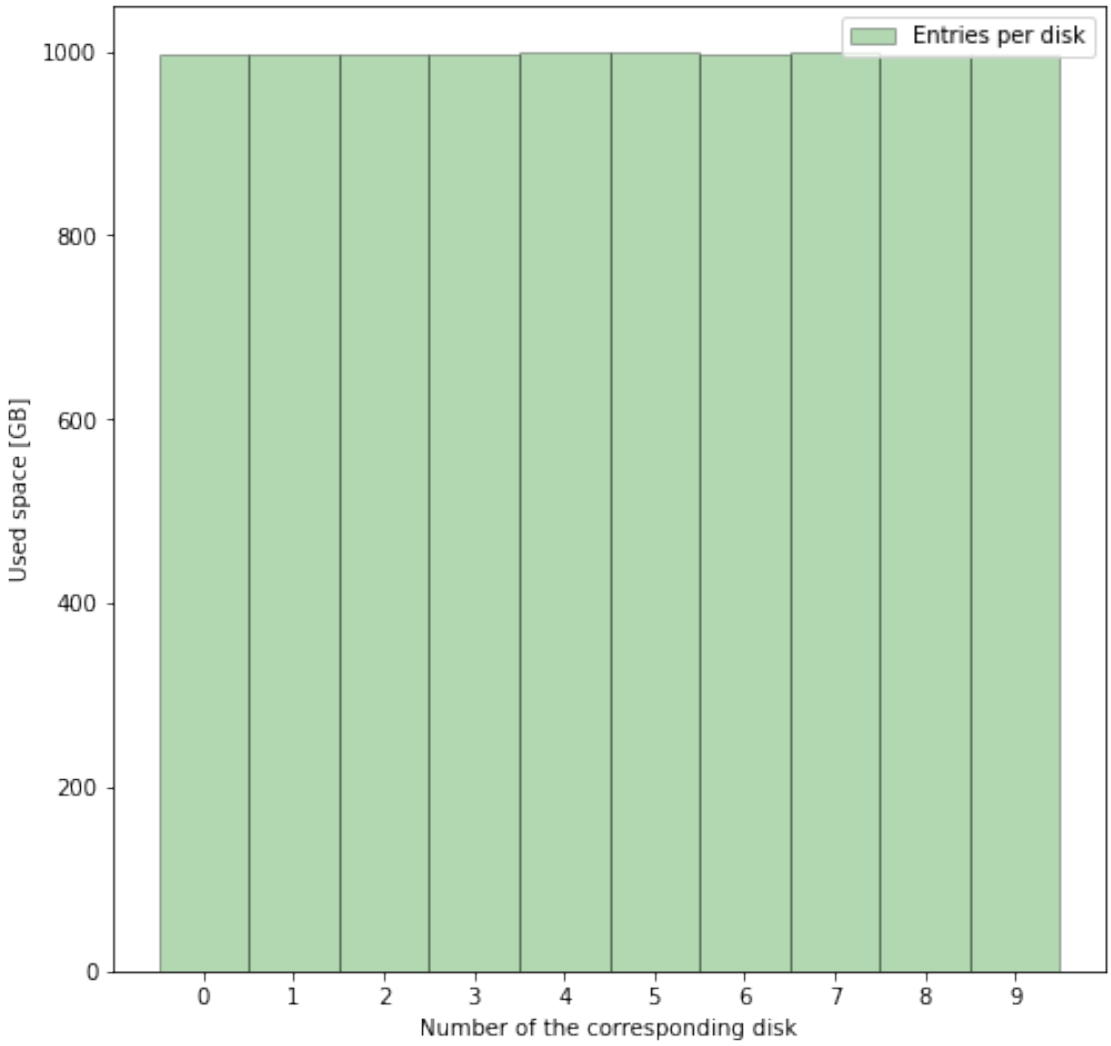
3.2b What is the percentage of total used space on all hard disks in the moment the first disk is full?

[12]:
```
print(f"The number of files placed: {sum(counts_1GB)}")
print(f"The total amount of space available is  10 x 1TB and we have placed␣
 ↪{sum(counts_1GB)} x 1GB files.",
      f"\nSo we have used {sum(counts_1GB)/1000}TB of the 10 available. The␣
 ↪percentage: {sum(counts_1GB)/10000*100}%")
```

```
The number of files placed: 9626
The total amount of space available is  10 x 1TB and we have placed 9626 x 1GB
files.
So we have used 9.626TB of the 10 available. The percentage: 96.26%
```

3.3 Based on this observation: why do you think object storage typically stores fixed size blocks of 4M and not files of GBs size as a whole? (so called block storage approach) Run the same program for 4M block sizes and demonstrate the benefits

[13]:
```
#250*4MB = 1GB => 1000GB = 1TB
counts_4MB = [0]*10
np.random.seed(123456)

for j in np.random.randint(0,10, 2500000):
    counts_4MB[j]+=0.004
    if(counts_4MB[j] >= 1000): break


plt.figure(figsize = (8,8))
plt.hist(x, bins = range(11), align='left',  weights = np.array(counts_4MB),␣
 ↪alpha = 0.3, color = 'green',
         histtype='bar', label = 'Entries per disk', ec='black')
plt.xlabel("Number of the corresponding disk")
plt.ylabel("Used space [GB]")
plt.legend()
plt.xticks(np.arange(0,10))
plt.show()
```



[14]:
```
print(f"The number of files placed: {sum(counts_4MB)/0.004:.0f}")
print(f"The total amount of space available is 10 x 1 TB and we have placed␣
↪{sum(counts_4MB)/0.004:.0f} x 4MB files.",
      f"\nSo we have used {sum(counts_4MB)/1000:.3f}TB of the 10 available. The␣
↪percentage: {sum(counts_4MB)/100:.2f}%")
```

```
The number of files placed: 2494566
The total amount of space available is 10 x 1 TB and we have placed 2494566 x
4MB files.
So we have used 9.978TB of the 10 available. The percentage: 99.78%
```

We can clearly see that with this simple algorithm the space that we manage to fill increases reducing the dimension of the chunks stored, and the distribution of the entries per disk tend to be more and more uniform as we increase the number of random numbers(i.e. file placed) generated.

Moreover, some application may need to read small chuncks of data coming from bigger files (ex.

last 100 calls to a call center). With this chunk reduction, waiting times are cut by orders of magnitude.

In principle, one could split files in smaller and smaller chunks, optimizing the used space (hypothetically down to bits). But also the time to handle each chunk (read/write/send..) has to be considered. 4MB chunks are in most applications a reasonable tradeoff between filling efficiency and handling ease.

3.4a Compute the average used space on all hard disks and the standard deviation for the average used space for 10 GB and 1GB and 4M files. How is the standard deviation correlated to the block size and why?

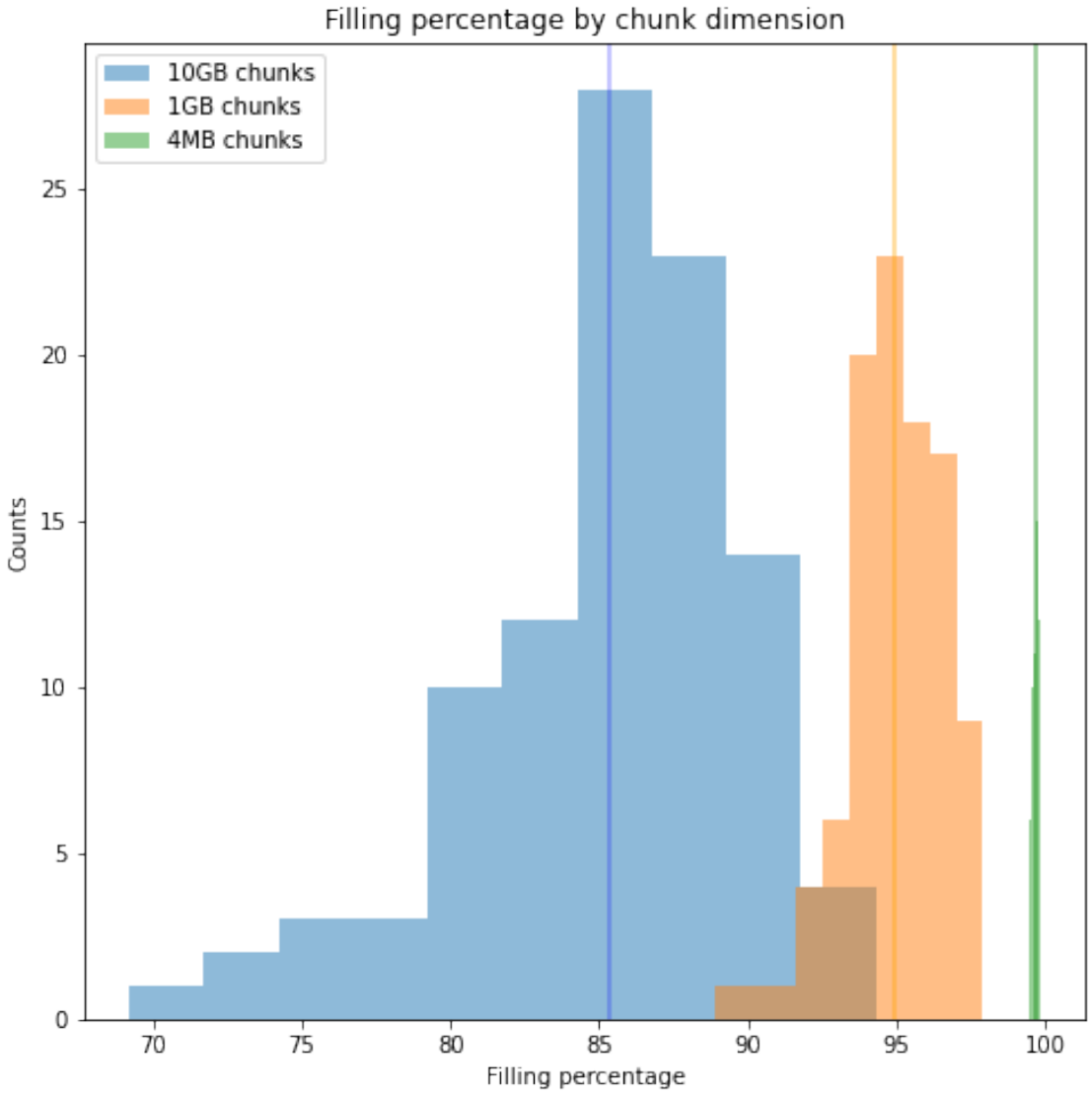
Let's re-run the previous experiment for 100 times and look at the average filled space per 10-disk-set, plotting the distributions and computing means and std_devs.

[15]:
```
k = np.arange(10)
results_10GB = []
results_1GB = []
results_4MB = []
np.random.seed(123456)

for it in range(100):

    counts = [0]*10
    for j in np.random.randint(0,10, 1000):
        counts[j]+=1
        if(counts[j] == 100): break

    results_10GB.append(sum(counts))

    counts = [0]*10
    for j in np.random.randint(0,10, 10000):
        counts[j]+=1
        if(counts[j] == 1000): break

    results_1GB.append(sum(counts))

    counts = [0]*10
    for j in np.random.randint(0,10, 2500000):
        counts[j]+=1
        if(counts[j] == 250000): break

    results_4MB.append(sum(counts))
```

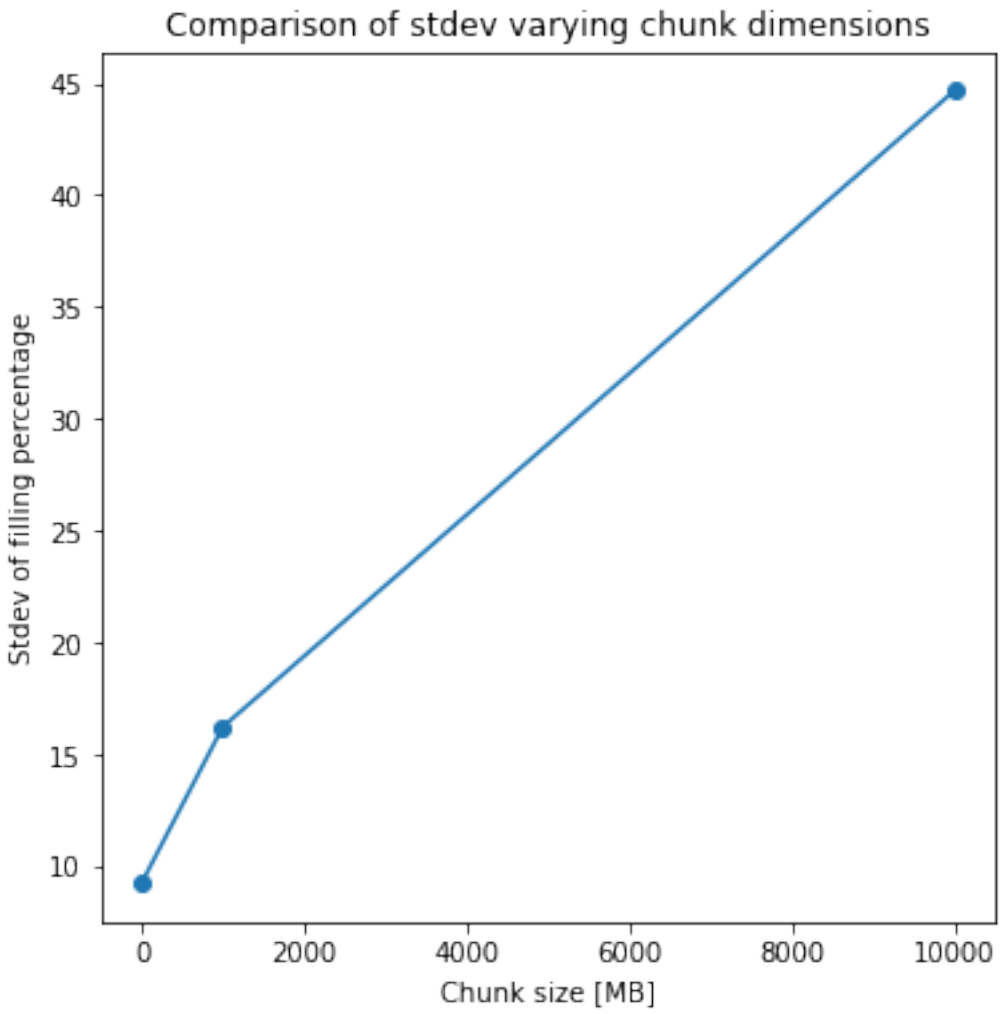
[16]:
```
results_10GB = np.array(results_10GB)
results_1GB = np.array(results_1GB)
results_4MB = np.array(results_4MB)

m_10 = results_10GB.mean()
```

```
std_10 = (results_10GB/100).std()

m_1  = results_1GB.mean()
std_1  = (results_1GB/1000).std()

m_4  = results_4MB.mean()
std_4  = (results_4MB/25000).std()

print(f"Filling percentages - means: 10 GB - {m_10/10:.2f},  1 GB - {m_1/100:.
 ↪2f},  4 MB - {m_4/25000:.2f}")

plt.figure(figsize = (8,8))
plt.hist(results_10GB/10, alpha = 0.5, label = "10GB chunks")
plt.hist(results_1GB/100, alpha = 0.5, label = "1GB chunks")
plt.hist(results_4MB/25000, alpha = 0.5, label = "4MB chunks")
plt.xlabel("Filling percentage")
plt.ylabel("Counts")
plt.title("Filling percentage by chunk dimension")
plt.legend()
plt.axvline(m_10/10, color= 'blue', alpha = 0.3)
plt.axvline(m_1/100, color = 'orange', alpha = 0.5)
plt.axvline(m_4/25000, color = 'green', alpha = 0.5)

plt.show()
```

```
Filling percentages - means: 10 GB - 85.37,  1 GB - 94.97,  4 MB - 99.68
```



[17]:
```
size = [4, 1000, 10000]
fill_size = [std_4*100, std_1*100, std_10*100]
plt.figure(figsize = (6,6))
plt.plot(size, fill_size, marker = 'o')
plt.title("Comparison of stdev varying chunk dimensions")
plt.xlabel("Chunk size [MB]")
plt.ylabel("Stdev of filling percentage")
plt.show()
```



The standard deviation shrinks dramatically as the dimension of the chunks decreases. This effect is related to the fact that, as shown in the previous histograms, the filling percentage of EACH single disks tends to be uniform as files are smaller.

This property leads then the total filling percentage to be constant throughout different sets of disks, and so the final percentage tends to be more reliable and solid.

If we were to repeat such task with thousands of disks with such properties, for sure the distribution of the disks filled with bigger chunks will be far from uniform, while reducing the chunks would lead to a flat distribution - again, smaller files are to be preferred in the case in which we stop filling once one single disk is full.

Moreover, it's also interesting to consider, in this case, the upper limits of such algorithm.

In the case of 10GB scenario, with 10x 1TB disks, the best case scenario is represented by 9 disks with 99 files, and one with 100 ones.

The best result for 10GB chunks is 99.1% - Actually lower than the average we get in case of 4MB

files! Really poor.

In the case of 1GB files, instead, it is 99.91%.

In the last 4MB case, 99.9964%!

The uniformity of the samples is also related on the ratio between the actual averages we got and the upper limits. In the case of 4MB files we are not far from our best estimate - 0.3%. Increasing the file size increases then variability, and so the actual means are very far from the best estimate. In such cases using a predetermined filling algorithm instead of a random generator may really help!

3.4b If we now repeat such an experiment for many more (thousands) of hard disks, which kind of distribution do you get when you do a histogram of the used space of all hard disks?

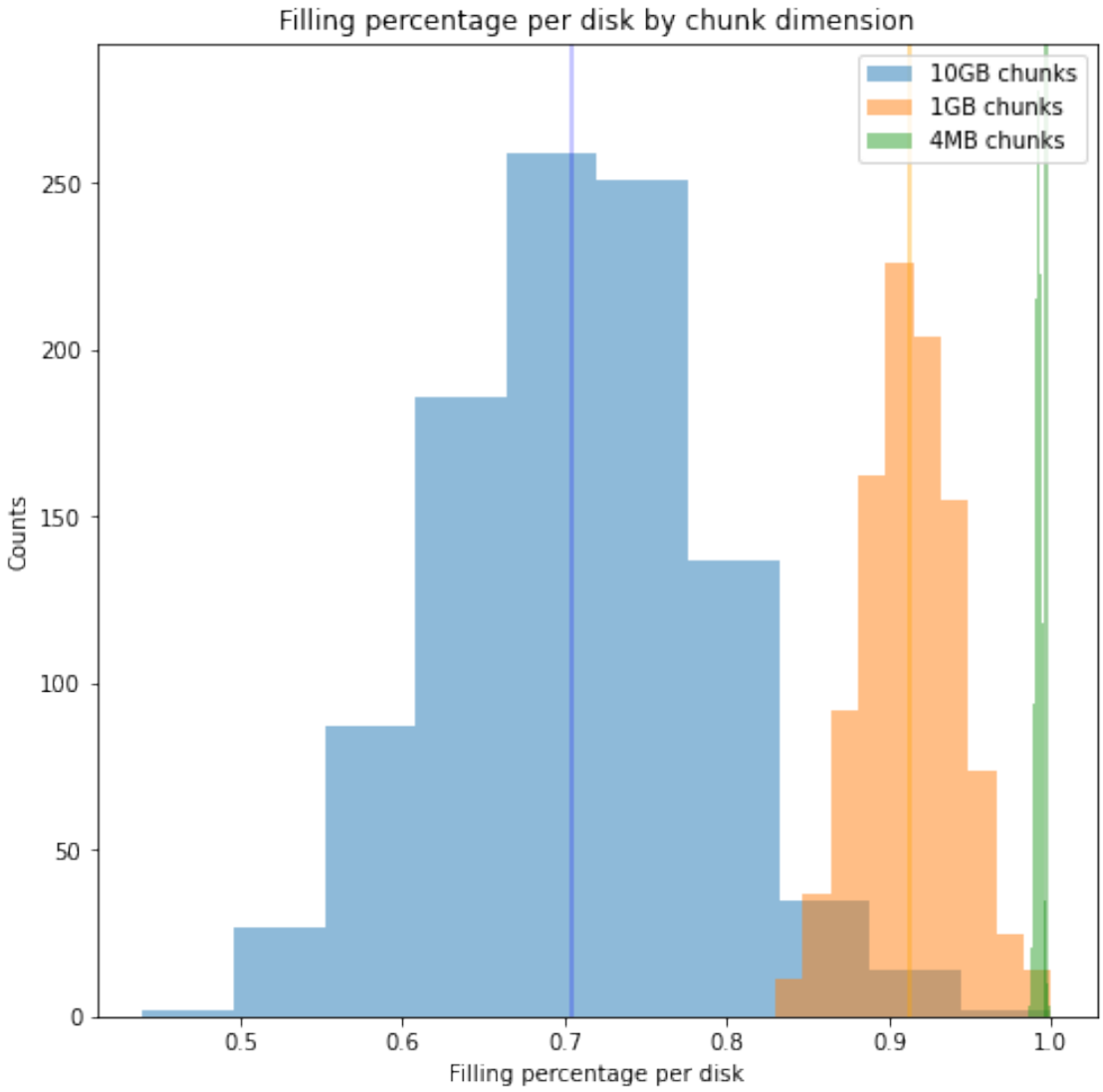
The distribution is different from the previous one - previously we were looking for the average filling space in the whole 10 sets, and repeating the same experiments for a hundred times.

The expected distribution in this case contains some larger tails, as some disk must reach the 100% space filling.

[18]:
```
k = np.arange(10)
results_10GB_1000 = []
results_1GB_1000 = []
results_4MB_1000 = []
np.random.seed(123456)

counts = [0]*1000
for j in np.random.randint(0, 1000, 1000*100):
    counts[j]+=1
    if(counts[j] == 100): break

results_10GB_1000.append(counts)

counts = [0]*1000
for j in np.random.randint(0, 1000, 1000*1000):
    counts[j]+=1
    if(counts[j] == 1000): break

results_1GB_1000.append(counts)

counts = [0]*1000
for j in np.random.randint(0,1000, 1000*250000):
    counts[j]+=1
    if(counts[j] == 250000): break

results_4MB_1000.append(counts)
```

[19]:
```
results_10GB_1000 = np.array(results_10GB_1000).flatten()
results_1GB_1000 = np.array(results_1GB_1000).flatten()
```

```
results_4MB_1000 = np.array(results_4MB_1000).flatten()

m_10 = results_10GB_1000.mean()
std_10 = (results_10GB_1000/100).std()

m_1  = results_1GB_1000.mean()
std_1  = (results_1GB_1000/1000).std()

m_4  = results_4MB.mean()
std_4  = (results_4MB/250000).std()


plt.figure(figsize = (8,8))
plt.hist(results_10GB_1000/100, alpha = 0.5, label = "10GB chunks")
plt.hist(results_1GB_1000/1000, alpha = 0.5, label = "1GB chunks")
plt.hist(results_4MB_1000/250000, alpha = 0.5, label = "4MB chunks")

plt.xlabel("Filling percentage per disk")
plt.ylabel("Counts")
plt.title("Filling percentage per disk by chunk dimension")
plt.legend(loc = 1)
plt.axvline(m_10/100, color= 'blue', alpha = 0.3)
plt.axvline(m_1/1000, color = 'orange', alpha = 0.5)
plt.axvline(m_4/2500000, color = 'green', alpha = 0.5)

plt.show()


print(f"Filling percentage: mean and std: 10GB - {m_10:.2f},  {std_10:.2f}")
print(f"Filling percentage: mean and std: 1GB  - {m_1/10:.2f},  {std_1:.2f}")
print(f"Filling percentage: mean and std: 4MB  - {m_4/25000:.2f},  {std_4:.2f}")
```




```
Filling percentage: mean and std: 10GB - 70.44,  0.08
Filling percentage: mean and std: 1GB  - 91.28,  0.03
Filling percentage: mean and std: 4MB  - 99.68,  0.01
```

As can be also seen from the graph, the usage of 4MB files still brings major advantages in terms of filling capacity. Moreover, the stadard deviation of the 4MB distribution is really small - all disk are almost equally filled and the algorithm used shows a very high efficiency.

1.5 4. Rest APIs & Block Chain Technology

1.5.1 I have two usernames - CANE (which means "dog" in italian, my usual one online), and LorenzoDomenichetti

Under https://pansophy.app:8443 you find a (hopefully running) Crypto Coin Server exporting a simple Block Chain. You can open this URL in any web browser to see the current Block Chain

status and the account information. At the time of writing the initial birth account of the Block Chain contained 1M coins (“genesis” : 1000000).

The REST responses are given in JSON format. Our REST API uses secure HTTP protocol and it is based on two HTTP methods:

GET

POST

GET requests are used, to retrieve any kind of information, POST requests are used to change state in the server.The task is to implement a client and use a simple REST API to submit transactions to the Block Chain. Your goal is to book coins from other people’s accounts to your own account. The server implements a Proof Of Time algorithm. To add a transaction to move coins to your account, you have to submit a merit request and you have to let time pass before you can send a claim request to execute your transaction on the Block Chain. If you claim your transaction too fast after a merit request, your request is discarded. The server enforces a Proof Of Time of a minimum of 10 seconds!

4.1.1 Use the REST API and the curl command to transfer coins of the genesis or any other account on your own team account.

You can use the -d option to POST a document. You have to indicate in your request, that the content type of the document is JSON. To do this you can add an HTTP header for this command curl … -H ”Content-Type: application/json” …

If you prefer, you can use a Python program, doing the same HTTPS requests respecting Proof of Time. If you want to have some more fun, you can also load the current state into your Python script using GET requests and programatically steal from accounts which are reported. Be aware, that you can never steal the last coin of an account and if at the time of a claim there are not enough coins left on an account, your transaction is discarded.

To you will have to add at least one successful transaction to the Block Chain.

[20]:
```
# defining the api-endpoint
API = "https://pansophy.app:8443/"

# data to be sent to api

data_merit = {
    "operation": "merit",
    "team": "LorenzoDomenichetti",
    "coin": 1,
    "stealfrom": "CANE"

}

data_claim= {
    "operation": "claim",
    "team": "LorenzoDomenichetti",
```

```
}


# sending post request and saving response as response object
r = requests.post(url = API, json = data_merit, verify = False)

# extracting response text
pastebin_url = r.text
print("The pastebin URL is:%s"%pastebin_url)


time.sleep(10)

# sending post request and saving response as response object
r = requests.post(url = API, json = data_claim, verify = False)


# extracting response text
pastebin_url = r.text
print("The pastebin URL is:%s"%pastebin_url)
```

```
The pastebin URL is:{"msg": "accepted"}
The pastebin URL is:{"msg": "request claimed"}
```

[21]:
```
#AUTOMATIC STEALING ALGORITHM.
#ONE GET REQUEST PER CYCLE, THAN RUN OVER ALL ACCOUNTS STEALING 1k COINS PER␣
 ↪CLAIM.
while(1):

    break #comment this line for starting automatic steal!

    data = requests.get(url = "https://pansophy.app:8443/", verify = False)

    data = data.json()

    API = "https://pansophy.app:8443/"

    for accounts in data['accounts'].items():

        if(accounts[0] == 'CANE'): continue
        money = accounts[1]

        for it in range(int(accounts[1]/1000)+1):
            if(money == 1): continue
            if(money<=1000):
                data_merit = {
                    "operation": "merit",
```

```
            "team": "CANE",
            "coin": money-1,
            "stealfrom": accounts[0]
        }
    else:
        data_merit = {
            "operation": "merit",
            "team": "CANE",
            "coin": 1000,
            "stealfrom": accounts[0]
        }


    data_claim= {
        "operation": "claim",
        "team": "CANE",
    }


    print(accounts[1], it, money )
    print(data_merit)

    # sending post request and saving response as response object
    r = requests.post(url = API, json = data_merit, verify = False)

    # extracting response text
    pastebin_url = r.text
    print("The pastebin URL is:%s"%pastebin_url)


    time.sleep(10)

    # sending post request and saving response as response object
    r = requests.post(url = API, json = data_claim, verify = False)


    # extracting response text
    pastebin_url = r.text
    print("The pastebin URL is:%s"%pastebin_url)


    money = money - 1000
```

4.2 What is the maximum number of transactions one given team can add to the Block Chain in one day?

8640! As in a day there are 86400 seconds and we can only send one claim successfully every 10s.

4.2.1 Explain what this function does and why is this 'the key' for Block Chain technology?

```
[ ]: def calculate_hash(self):
     block_of_string = "{}{}{}{}{}{}".format( self.index,
     self.team,
     self.prev_hash,
     self.coins,
     self.timestamp)
     self.my_hash =
     hashlib.sha256(block_of_string.encode()).hexdigest()
     return self.my_hash
```

This function computes the hash of the current block in the blockchain, given the information contained in the block itself (team, coins, timestamp, index) AND the previous valid hash. It basically concatenates the information inside a string and then hashed it through the SHA-256 algorithm, returning a 256bit string containing all the information of the considered block.

It is truly the key of the blockchain as every block is concatenated usign the same procedure, and of course each of them have to be consistent with the previous one in order to be properly attached to the chain. Infact, if one tries to hack any block of the chain (in principle), then the neighbouring blocks will have inconsistent hashes, breaking the whole chain.

4.2.2 If you have the knowledge of the hash function, how can you validate the contents of the Block Chain you received using a GET request to make sure, nobody has tampered with it? You don't need to implement it! Explain the algorithm to validate a Block Chain!

As each block is related to the previous one, it is possible to check if all blocks are coherent with the previous one.

- As the index feature counts the position of each block, we may want to check weather the next block has index = prev_index+1
- Each block contains the hash of the previous one, so it is possibile to check that the two actually are equal.
- One last check can be done asking that the timestamp of the considered block is greater than the timestamp of the previous one.

IF all these requirement are satisfied (at least in this simple case) for each block of the received chain, we may then conclude that the chain has not been tampered.

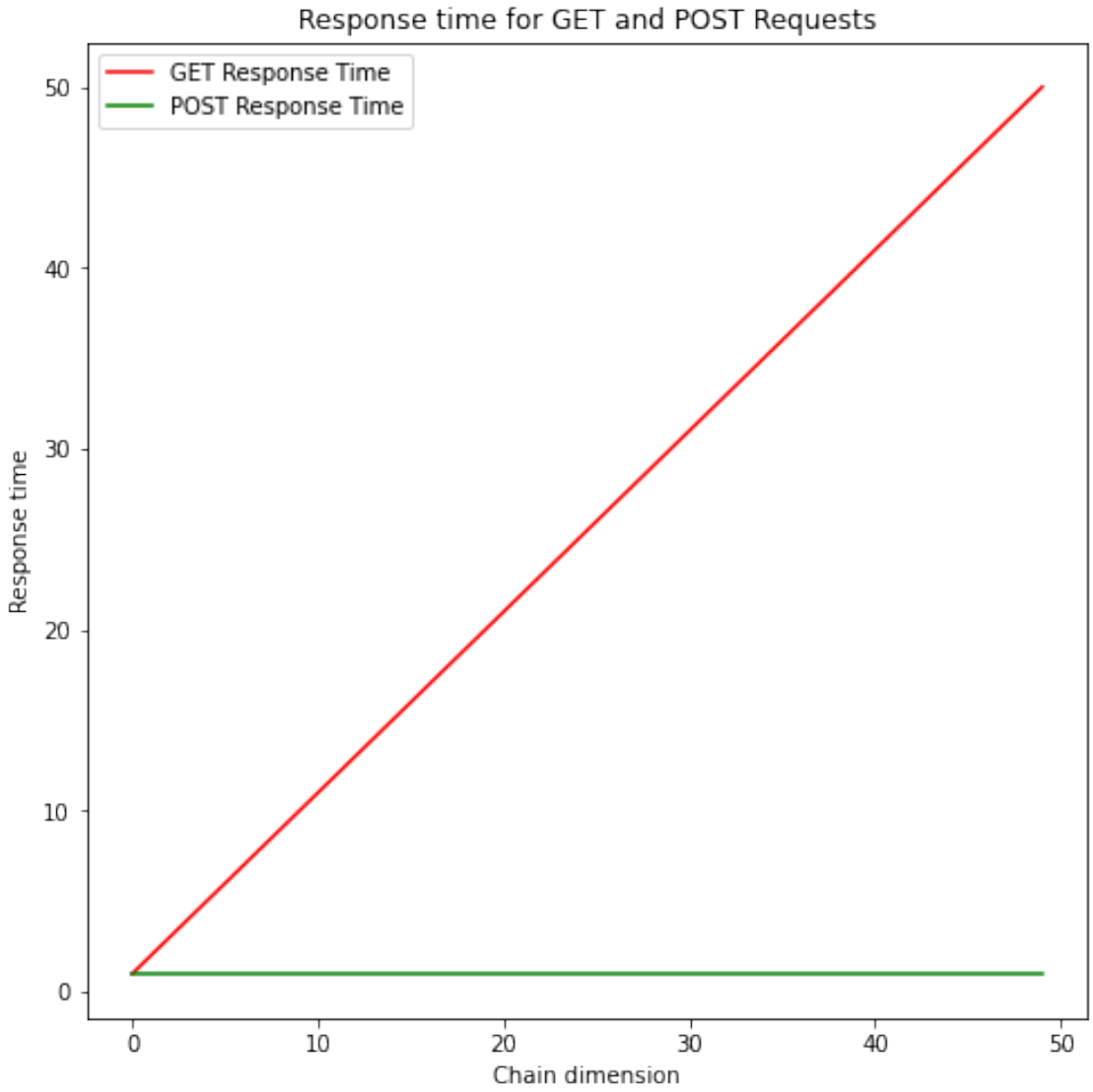
OBS: these are the steps computed in the hands-on session. More advanced settings can be then added to make the chain even more secure. (For example the proof_no feature, which was implemented in our example, which would try to prevent users from mining blocks easily).

4.2.3 Why might the GET REST API run into scalability problems? Express the scalability behaviour of execution times of GET and POST requests in Big O notation in relation to the number of transactions recorded in the Block Chain! Draw execution time vs transactions for GET and POST requests.

We may now assume that THE time the GET request takes is constant for each block in the chain, so it may scale as O(N) for big values of N. On the other hand, posting a request requires only

the knowledge of the last hash. This may lead us to believe the POST time to be constant, so independent from the dimension of the chain - in bigO notation O(1).

[22]:
```
plt.figure(figsize= (8,8))
plt.plot(np.array(range(50))+1, label = 'GET Response Time', c = 'red')
plt.plot(np.repeat(1,50), label = 'POST Response Time', c = 'green')
plt.xlabel("Chain dimension")
plt.ylabel("Response time")
plt.title("Response time for GET and POST Requests")
plt.legend()
plt.show()
```



4.2.4 If the Crypto server goes down, the way it is implemented it loses the current account balances. How can the server recompute the account balances after a restart from the saved Block

Chain?

If the server goes down and loses the balances, the simplest way to recover the balances is to start reading the chain again from bottom to top, so starting from the first genesis block up to the last executed transaction.

4.2.5 What are the advantages of using a REST API and JSON in a client-server architecture? What are possible disadvantages?

Advantages:

REST API is easy to understand and learn. Moreover, thanks to this user-friendly library "requests", the usage is super-easy also in python.

With the possibility to caching and using HTTP proxy, high loads can be efficiently managed.

Users can exploit standard HTTP procedure call-outs to retrieve data and requests.

Brings flexibility formats by serializing data in JSON format.

Allows Standard-based protection with the use of OAuth protocols to verify your REST requests.

Disadvantages:

Lack of state: most web applications require stateful mechanisms. The burden of maintaining the state lies on the client, which makes the client application heavy and difficult to maintain.

Last of security: REST doest impose security such as other APIs. That is the reason REST is appropriate for public URLs, but it is not good for confidential data passage between client and server.

In general, RESI APIs reduce complexity and lets users handle resources with ease and few operations.