DIOGOMMARTINS

CONCORRÊNCIA E PARALELISMO – THREADS, MÚLTIPLOS PROCESOS E ASYNCIO

QUEM SOU EU

- Bacharel em SI
- Programador por profissão e amor a ~10 anos
- ❤️ Python
- Trabalho na SIEVE

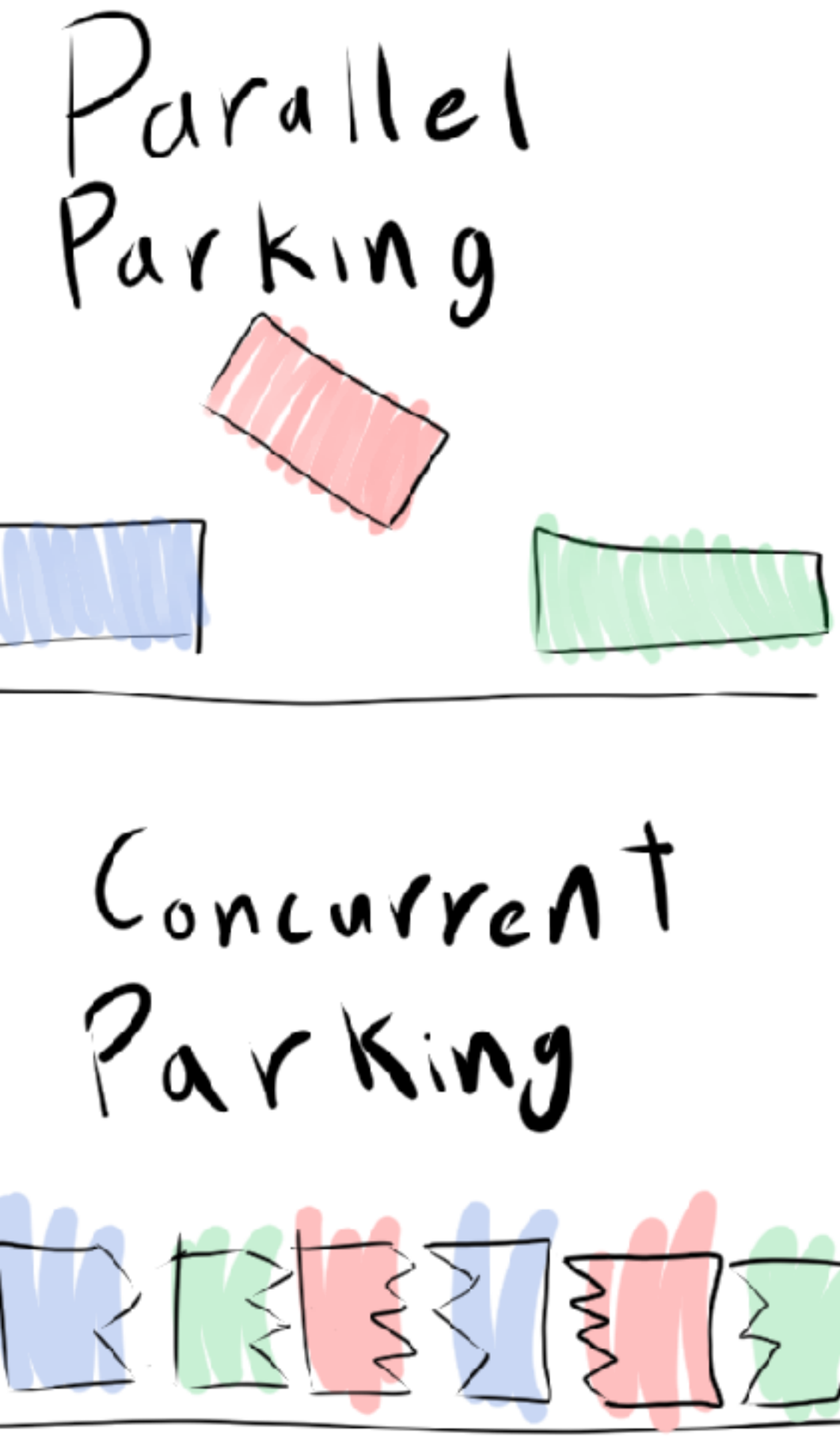
CONTRATAMOS DEVS

JR - PL - SR

diogo.martins@sieve.com.br

OBJETIVO

- Auxiliar desenvolvedores na escolha da abordagem correta pra solução de problemas que exigem performance



CONCORRÊNCIA E PARALELISMO

Pode parecer a mesma coisa, mas não é.

PARALELISMO

- Duas ou mais tarefas são executadas literalmente ao mesmo tempo.
- Necessita de múltiplos CPUs ou múltiplos processadores.

CONCORRÊNCIA

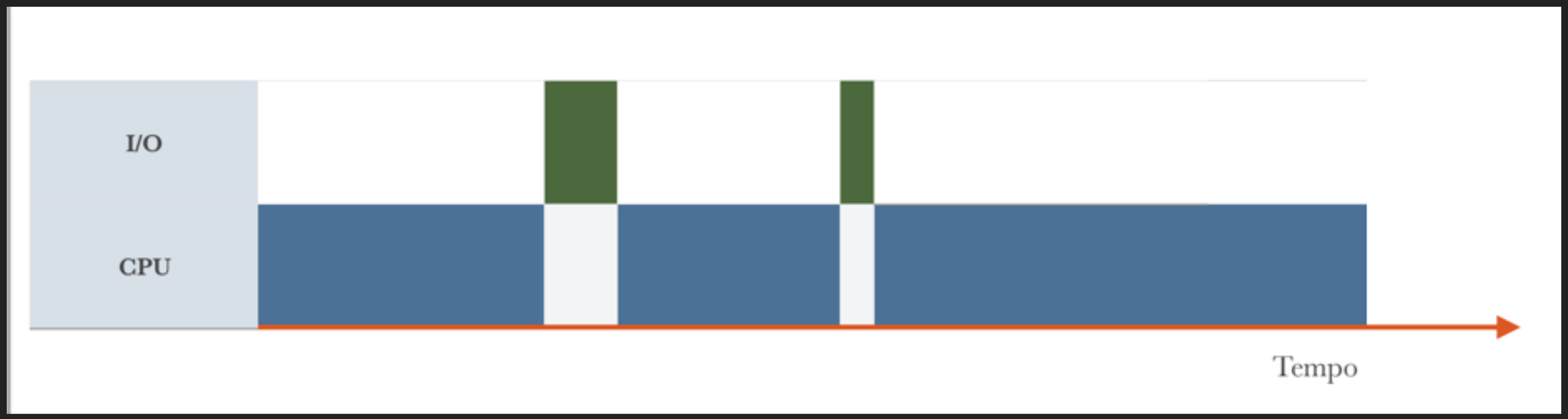
- Duas ou mais tarefas podem começar a ser executadas e terminar em espaços de tempo que se sobrepõem.
- Não significa que elas estejam em execução ao mesmo tempo.
- Mais de uma tarefa é processada em um mesmo intervalo de tempo, não esperando que uma tarefa termine por completo antes de dar início a outra

TIPOS DE PROCESSOS

CPU BOUND E I/O BOUND

CPU BOUND

- Passam a maior parte do tempo em estado de execução, utilizando o processador.
- Aplicações que o fator determinante para o tempo no qual ela vai levar para terminar, é determinado pela velocidade da CPU.

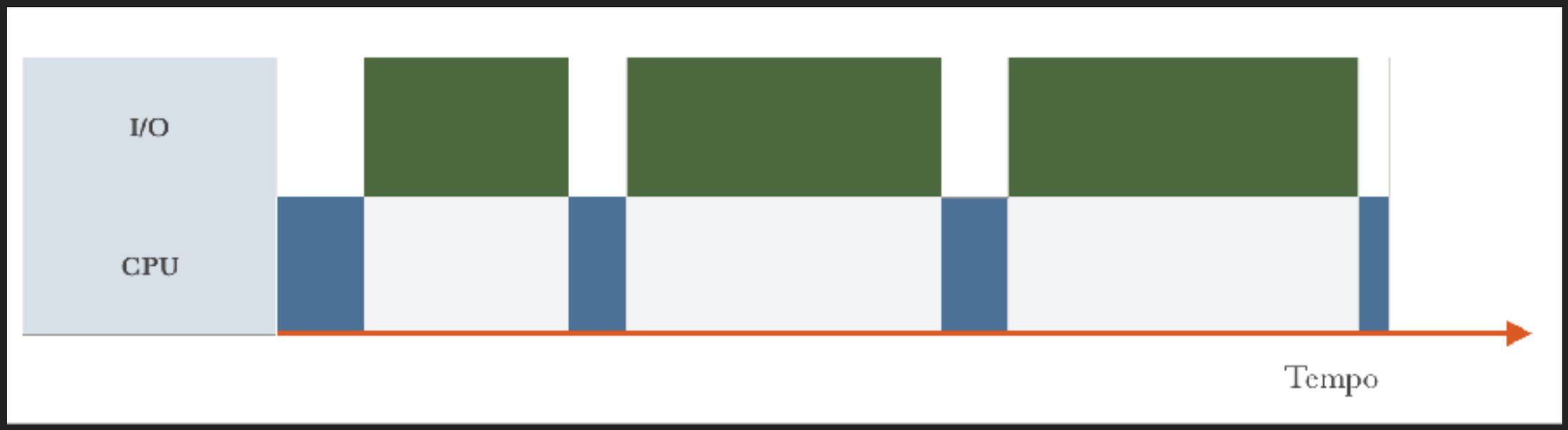


CPU BOUND

- Computações matemáticas intensas
- Algoritmos de busca e ordenação em memória
- Processamento e reconhecimento de imagem

I/O BOUND

- Passa a maior parte em estado de espera por realizar muitas operações de I/O.
- Tempo de execução é determinado pelo tempo gasto esperando por operações de entrada e saída.



I/O BOUND

- Transferência de dados pela rede
- Copiar/Mover arquivo em disco
- Consultar um banco de dados remoto
- Consultar uma API HTTP

I/O BOUND

EXEMPLOS

SEQUENCIAL

- Realizar uma requisição HTTP para www.americanas.com e imprimir o resultado no stdout.

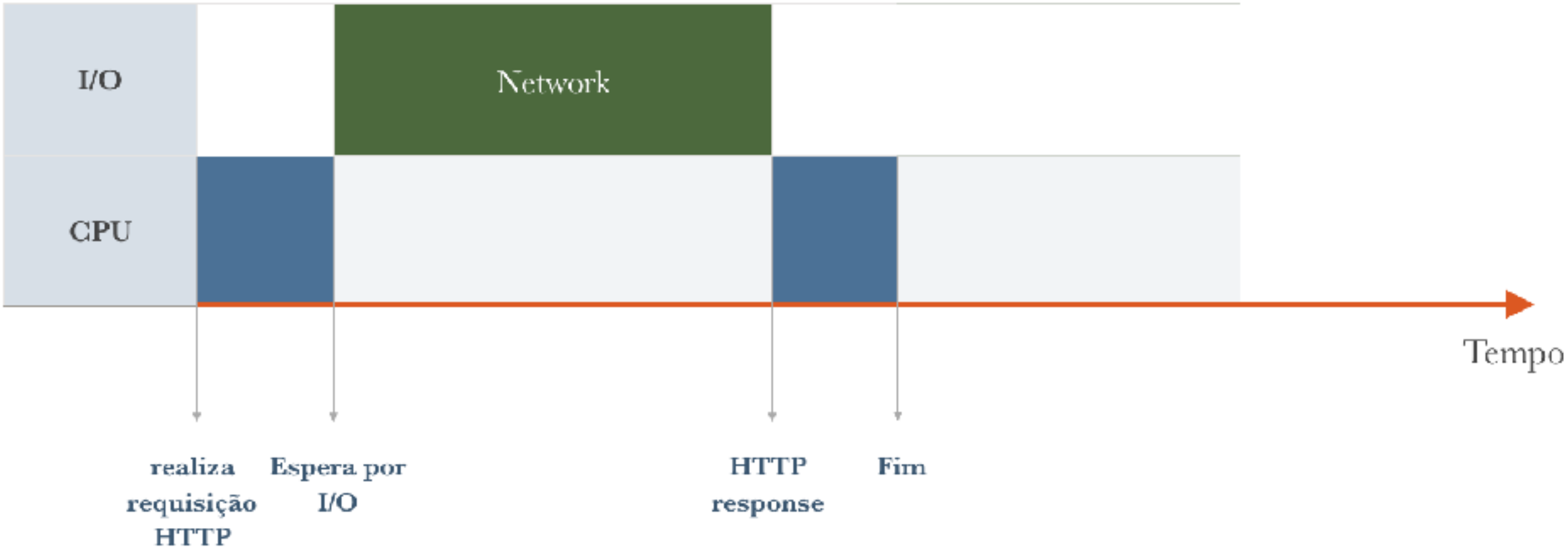
```python
import requests

response = requests.get('http://www.americanas.com')
print(response.text)
```

SEQUENCIAL

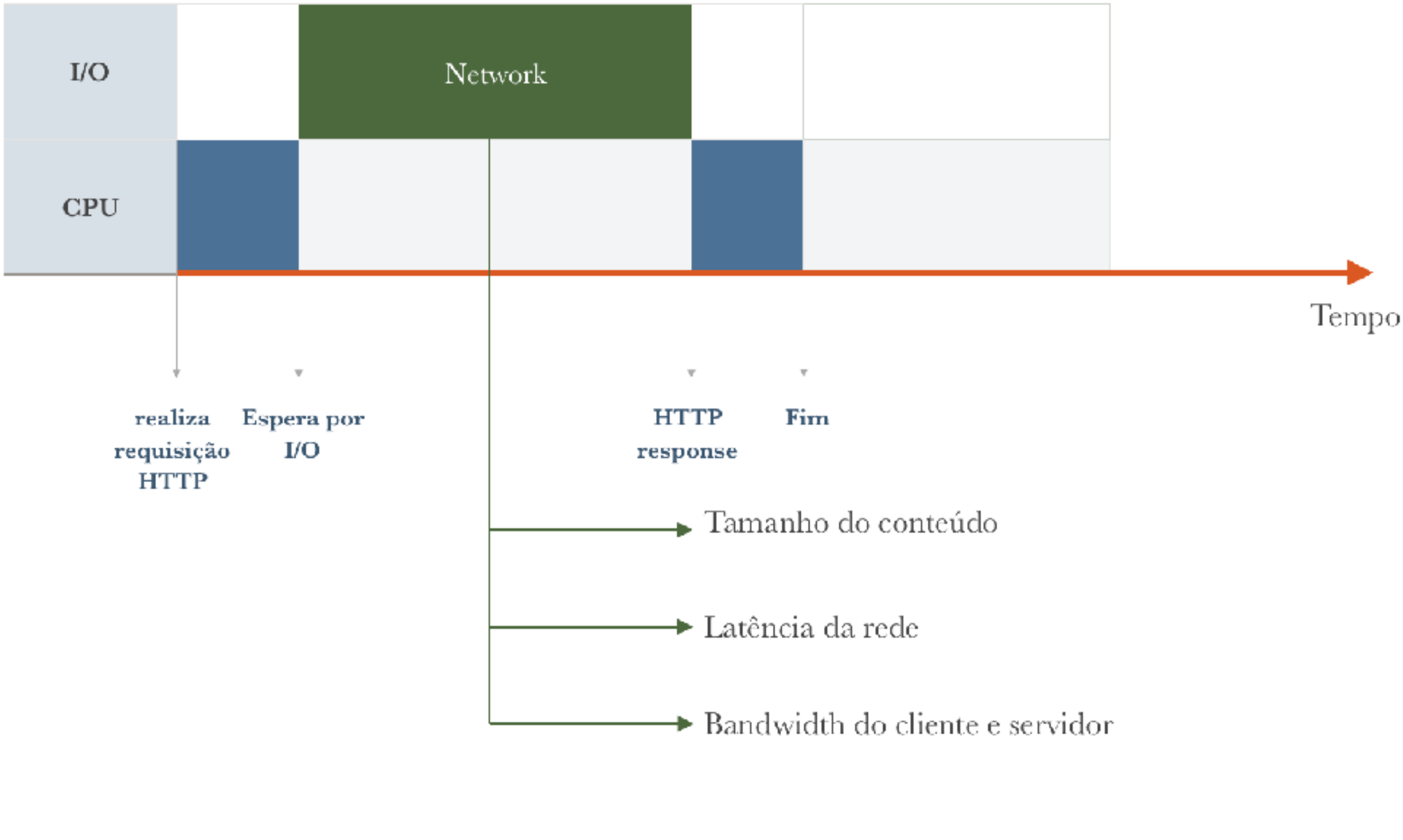
```python
import requests

response = requests.get('http://www.americanas.com')
print(response.text)
```

- Espera até que tenha uma resposta completa
- Bloqueante. Nenhuma operação pode ser executada enquanto a anterior não terminar.

SEQUENCIAL


I/O
Network
CPU
Tempo
realiza requisição HTTP
Espera por I/O
HTTP response
Fim

SEQUENCIAL

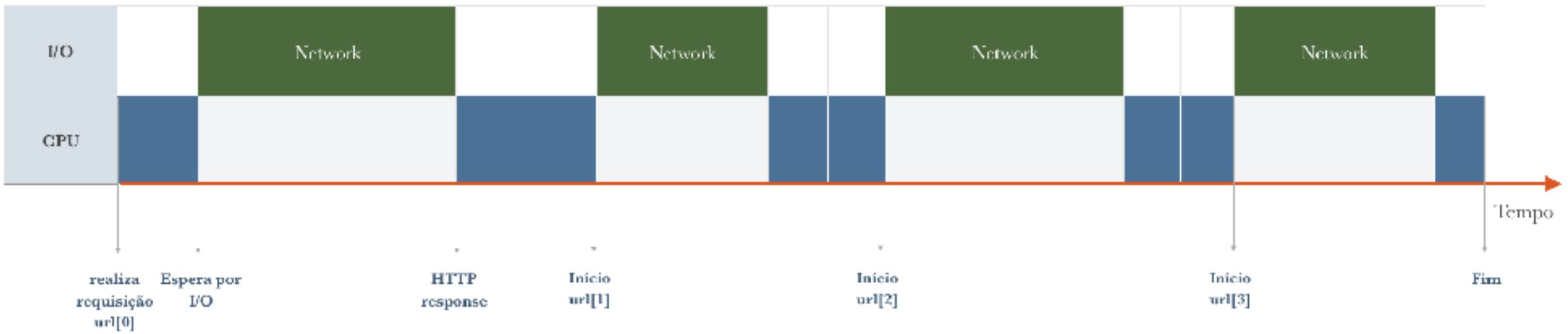

I/O
Network
CPU
Tempo
realiza requisição HTTP
Espera por I/O
HTTP response
Fim
Tamanho do conteúdo
Latência da rede
Bandwidth do cliente e servidor

SEQUENCIAL – 4 URLS

```
1 import requests
2
3 urls = (
4     'http://www.americanas.com',
5     'http://www.submarino.com',
6     'http://www.shoptime.com',
7     'http://www.soubarato.com',
8 )
9
10 for url in urls:
11     response = requests.get(url)
12     print(response.text)
```

- CPU ociosa enquanto espera pela resposta
- Cada tarefa precisa esperar pela finalização da anterior
- Tempo gasto em I/O fica mais aparente.

SEQUENCIAL – 4 URLS

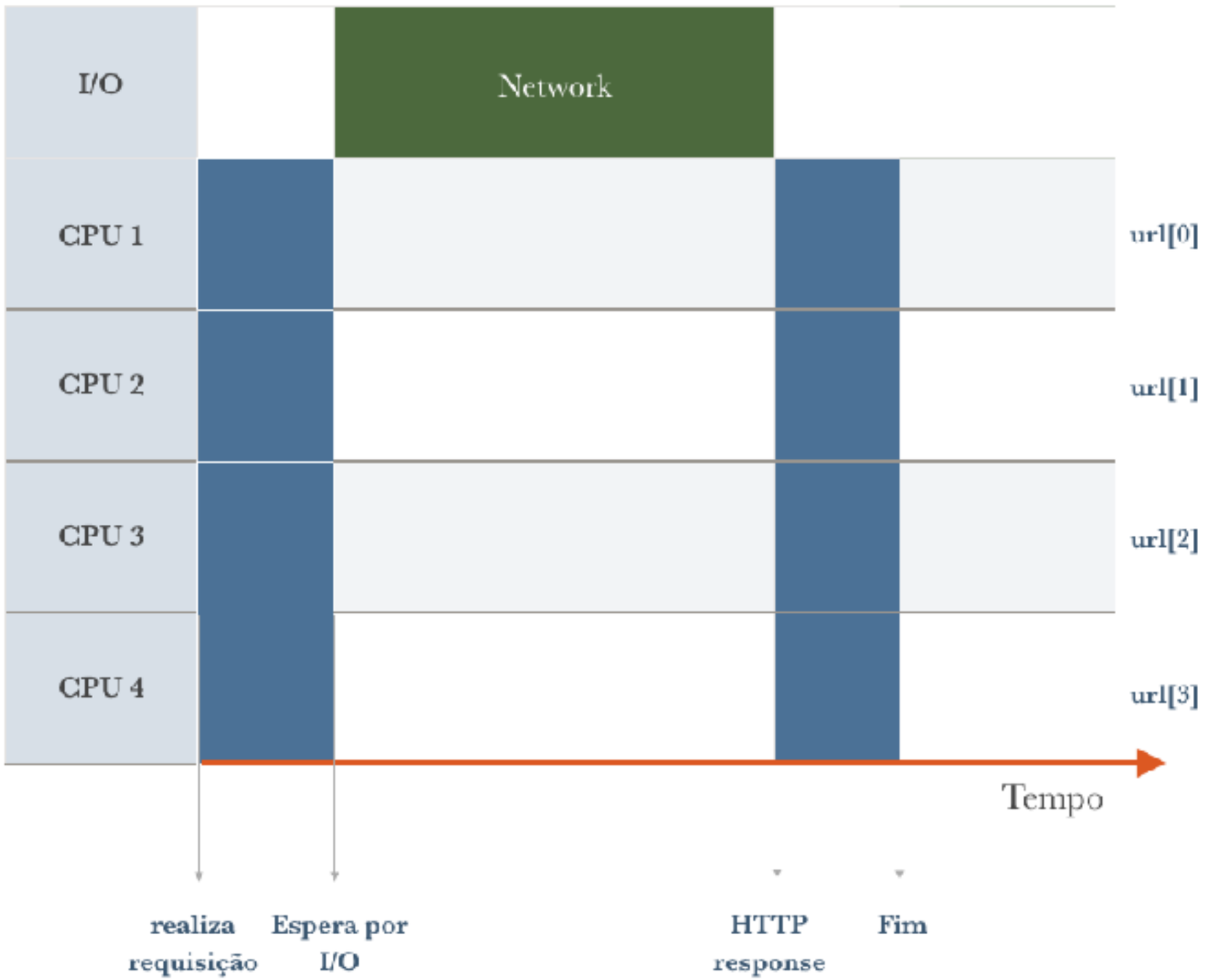


MÚLTIPLOS PROCESSOS - 4 URLS - 4 CPUS

```python
import os
import requests
from multiprocessing import Pool


urls = (
    'http://www.americanas.com',
    'http://www.submarino.com',
    'http://www.shoptime.com',
    'http://www.soubarato.com',
)


def get_and_print(url):
    response = requests.get(url)
    print(response.text)


pool = Pool(processes=os.cpu_count())
pool.map(get_and_print, urls)
```

- Cada umas das CPUs utilizadas continua ociosa enquanto espera pela resposta
- Tempo gasto em I/O é distribuído pelos processos.
- Sem compartilhamento de memória. Comunicação entre processos é algo custoso.
- Overhead de criar novos processos

MÚLTIPLOS PROCESSOS – 4 URLS – 4 CPU

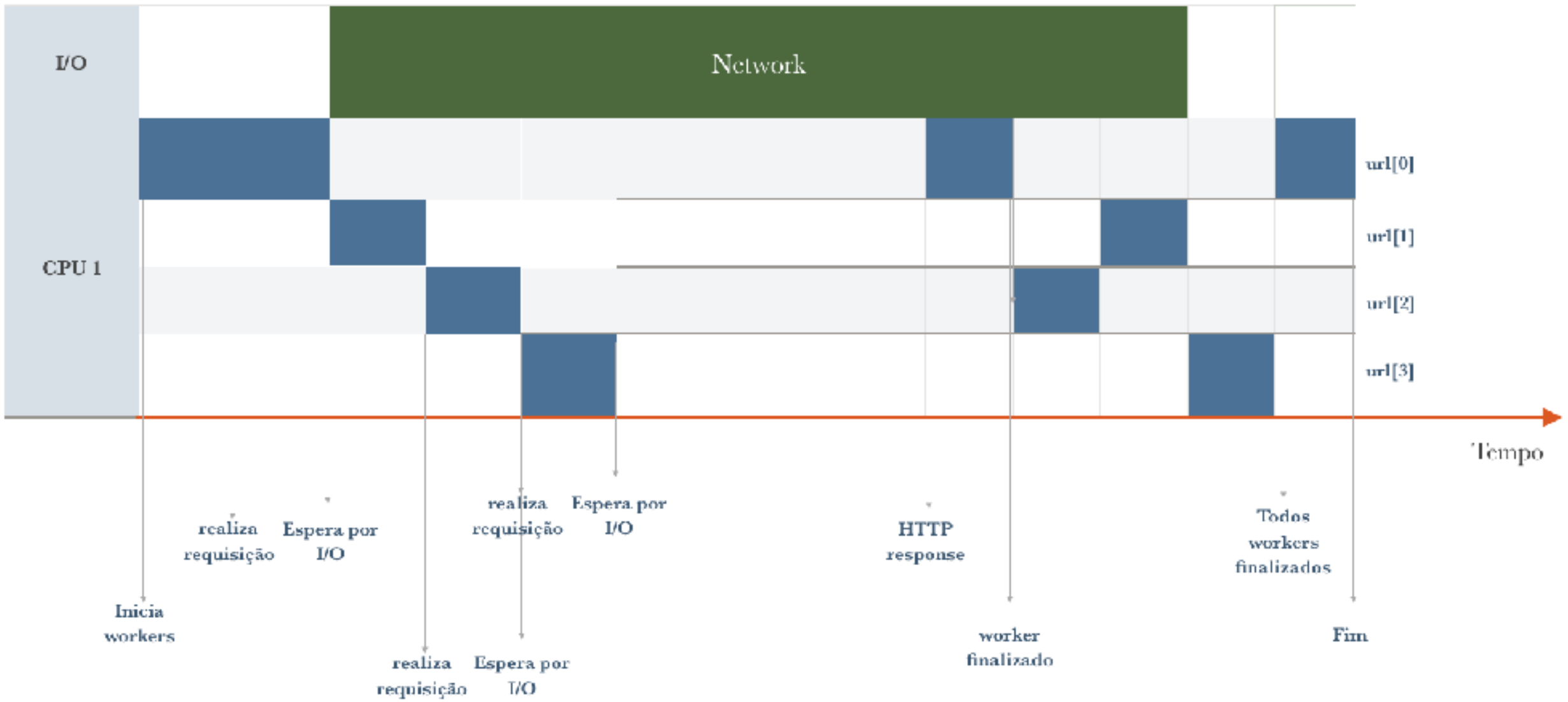


MÚLTIPLAS THREADS – 4 URLS – 4 THREADS

```python
import os
import requests
from multiprocessing.pool import ThreadPool as Pool

urls = (
    'http://www.americanas.com',
    'http://www.submarino.com',
    'http://www.shoptime.com',
    'http://www.soubarato.com',
)


def get_and_print(url):
    response = requests.get(url)
    print(response.text)


pool = Pool(processes=os.cpu_count())
pool.map(get_and_print, urls)
```

- Compartilhamento de memória
- Sincronização de uso de recursos
- Custo de criação e manutenção mais “leve" se comparado a processos
- São pthreads reais. Número de threads é limitado pelo SO

MÚLTIPLAS THREADS – 4 URLS – 4 THREADS



MÚLTIPLAS THREADS – 4 URLS – 4 THREADS

```python
import os
import requests
from multiprocessing.pool import ThreadPool as Pool

urls = (
    'http://www.americanas.com',
    'http://www.submarino.com',
    'http://www.shoptime.com',
    'http://www.soubarato.com',
)


def get_and_print(url):
    response = requests.get(url)
    print(response.text)


pool = Pool(processes=os.cpu_count())
pool.map(get_and_print, urls)
```

- Compartilhamento de memória
- Sincronização de uso de recursos
- Custo de criação e manutenção mais “leve" se comparado a processos
- São pthreads reais. Número de threads é limitado pelo SO

4.000 URLS ?

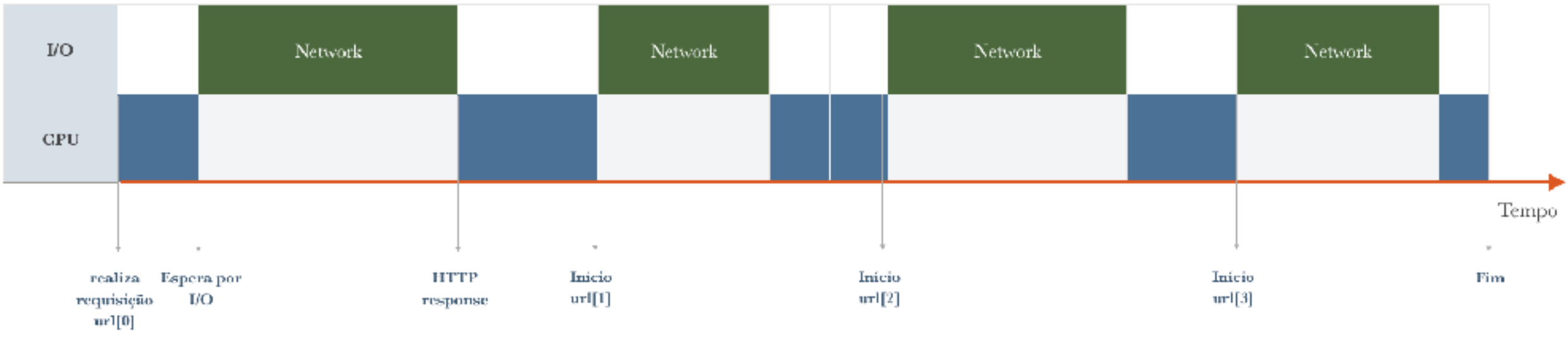
E SE PRECISÁSSEMOS REALIZAR 4.000 REQUISIÇÕES HTTP?

- Digamos que cada umas das 4000 requisições tenham o mesmo custo **T**

E SE PRECISÁSSEMOS REALIZAR 4.000 REQUISIÇÕES HTTP?

- Solução sequencial

4000 * **T**

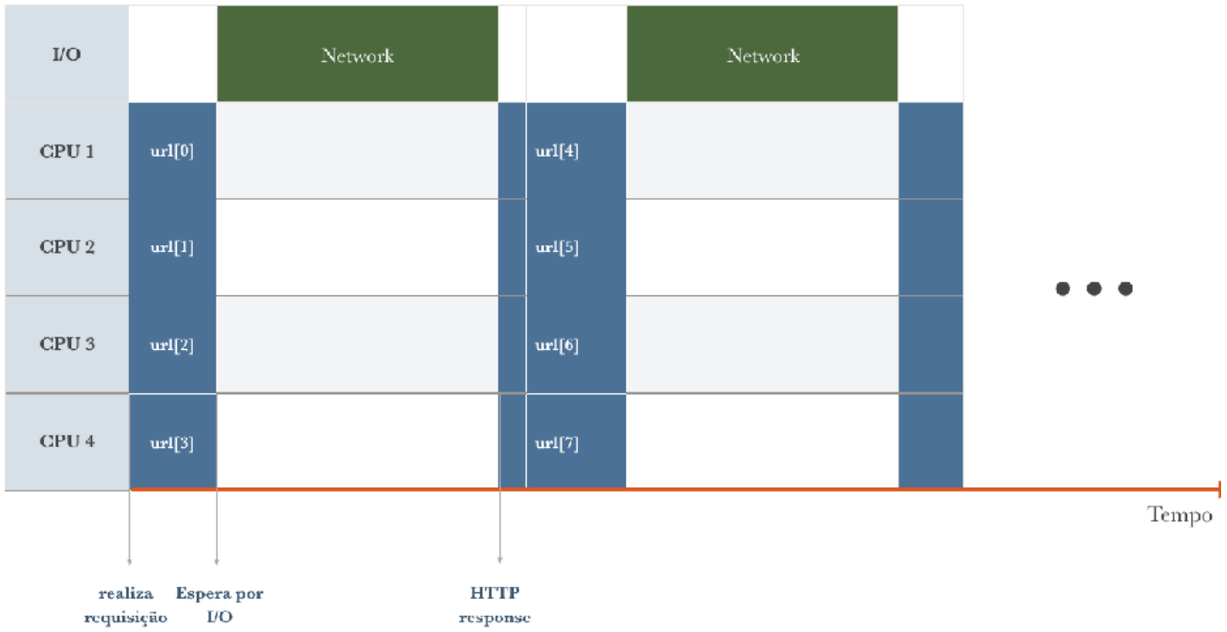


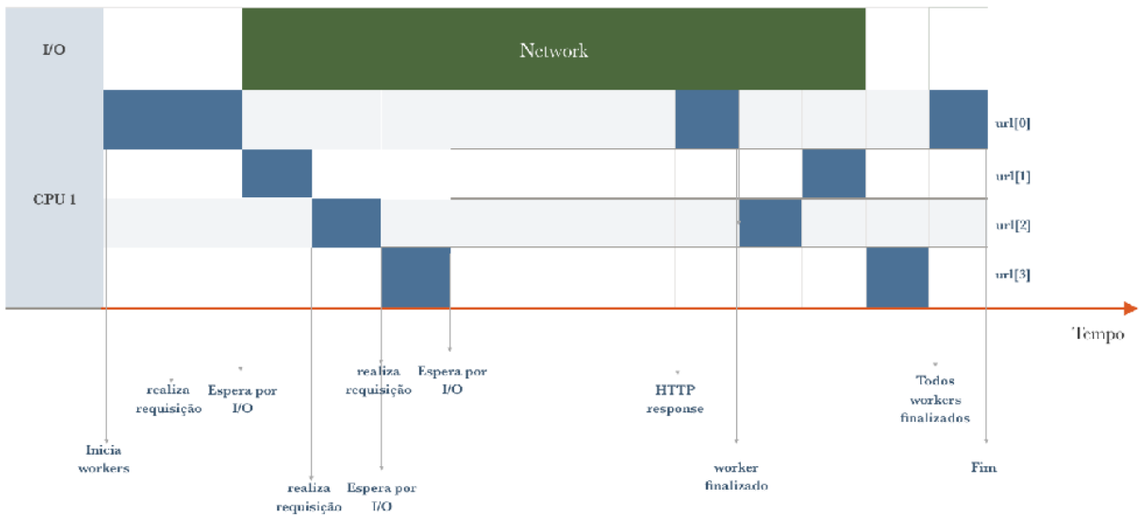
E SE PRECISÁSSEMOS REALIZAR 4.000 REQUISIÇÕES HTTP?

- Solução com múltiplos processos

E SE PRECISÁSSEMOS REALIZAR 4.000 REQUISIÇÕES HTTP?

- Solução com múltiplos processos e múltiplas threads





Custo = ((4000 * **T**) / **P**) / **P**)

E SE FOSSEM 400.000 OU 4.000.000 DE URLS?

Custo: ((n * **T**) / **Processos**) / **Threads**)

- Elevamos bastante o valor de **N**;
- Não conseguimos elevar a quantidade de **Processos** e **Threads** na mesma proporção;
- Benefício de múltiplos processos/threads não cresce linearmente.

Há algo de errado ou subutilizado na arquitetura da nossa solução, se na maior parte do tempo em que nossa solução está rodando, a CPU está ociosa ou trabalhando em questões não relacionadas a aplicação em si.

ASYNCHRONOUS I/O

Provê uma forma de se escalonar eficientemente aplicações I/O bound com código concorrente, em uma única thread

- Introduzida por Guido na PEP 3156 (em 2012!)
- asyncio na Standard Library
- Somente Python 3 (>=3.4)
- Sintaxe mais agradável em Python >=3.5

COROUTINES

- São funções que podem ser suspensas em determinados pontos de execução para serem retomadas depois, mantendo todos os estados de quando foi suspensa;
- Conjunto de sub-rotinas ou instruções, que permitem pontos de entrada, suspensão e retomada de execução em certas partes do código, realizando uma troca de contexto.

COROUTINES

- Em funções python, a facilidade da troca de contexto já existia através da sintaxe yield

```
1  from time import sleep
2
3
4  def ham():
5      letters = ('A', 'B')
6      for letter in letters:
7          yield letter
8
9  letters_gen = ham()
10
11 print(next(letters_gen))
12 sleep(1)
13 print(next(letters_gen))
```

COROUTINES

- Em funções assíncronas, a troca de contexto se dá utilizando yield from em python (>=3.4)

```python
import asyncio


@asyncio.coroutine
def soma(x, y):
    print("Calculando {x} + {y} ...".format(x=x, y=y))
    yield from asyncio.sleep(1.0)
    return x + y


@asyncio.coroutine
def print_soma(x, y):
    result = yield from soma(x, y)
    print("{x} + {y} = {result}".format(x=x,
                                        y=y,
                                        result=result))
```

COROUTINES

- E em python >=3.5, utilizando await

```python
import asyncio


async def soma(x, y):
    print(f"Calculando {x} + {y} ...")
    await asyncio.sleep(1.0)
    return x + y

async def print_soma(x, y):
    result = await soma(x, y)
    print(f"{x} + {y} = {result}")

```

LOOP DE EVENTOS

- Corotinas são executadas dentro
- Controlar e manter o agendamento de corotinas e seus pontos de suspensão e retomada toda vez que uma corotina necessitar de tempo para execução de uma tarefa;
- Lidar com sinais do sistemas operacional;
- Transmitir dados através da rede;
- Prover abstrações para transporte para diversos canais de comunicação;...

LOOP DE EVENTOS

```python
import asyncio


async def soma(x, y):
    print(f"Calculando {x} + {y} ...")
    await asyncio.sleep(1.0)
    return x + y

async def print_soma(x, y):
    result = await soma(x, y)
    print(f"{x} + {y} = {result}")

loop = asyncio.get_event_loop()
loop.run_until_complete(print_soma(1, 666))
loop.close()
```

CADE O I/O ?

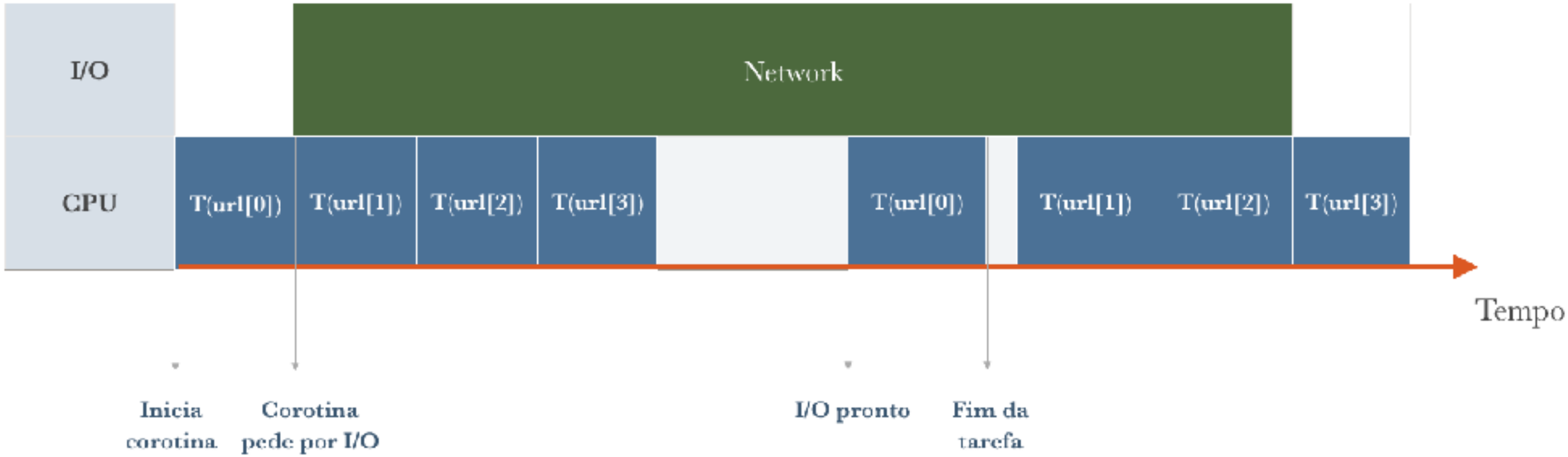
REALIZAR 1 REQUISIÇÃO HTTP

```python
import asyncio
from aiohttp import ClientSession


async def get_and_print():
    async with ClientSession() as session:
        async with session.get("http://www.americanas.com") as response:
            print(await response.text())

loop = asyncio.get_event_loop()
loop.run_until_complete(get_and_print())
```

REALIZAR 4 REQUISIÇÕES HTTP

```python
import asyncio
from aiohttp import ClientSession


urls = (
    'http://www.americanas.com', 'http://www.submarino.com',
    'http://www.shoptime.com', 'http://www.soubarato.com',
)

async def get_and_print(session, url):
    async with session.get(url) as response:
        print(await response.text())


async def fetch(urls):
    async with ClientSession() as session:
        tasks = (get_and_print(session, url) for url in urls)
        await asyncio.gather(*tasks, return_exceptions=True)


loop = asyncio.get_event_loop()
loop.run_until_complete(fetch(urls))
```

O QUE ISSO SIGNIFICA?



E 400.000 URLS?

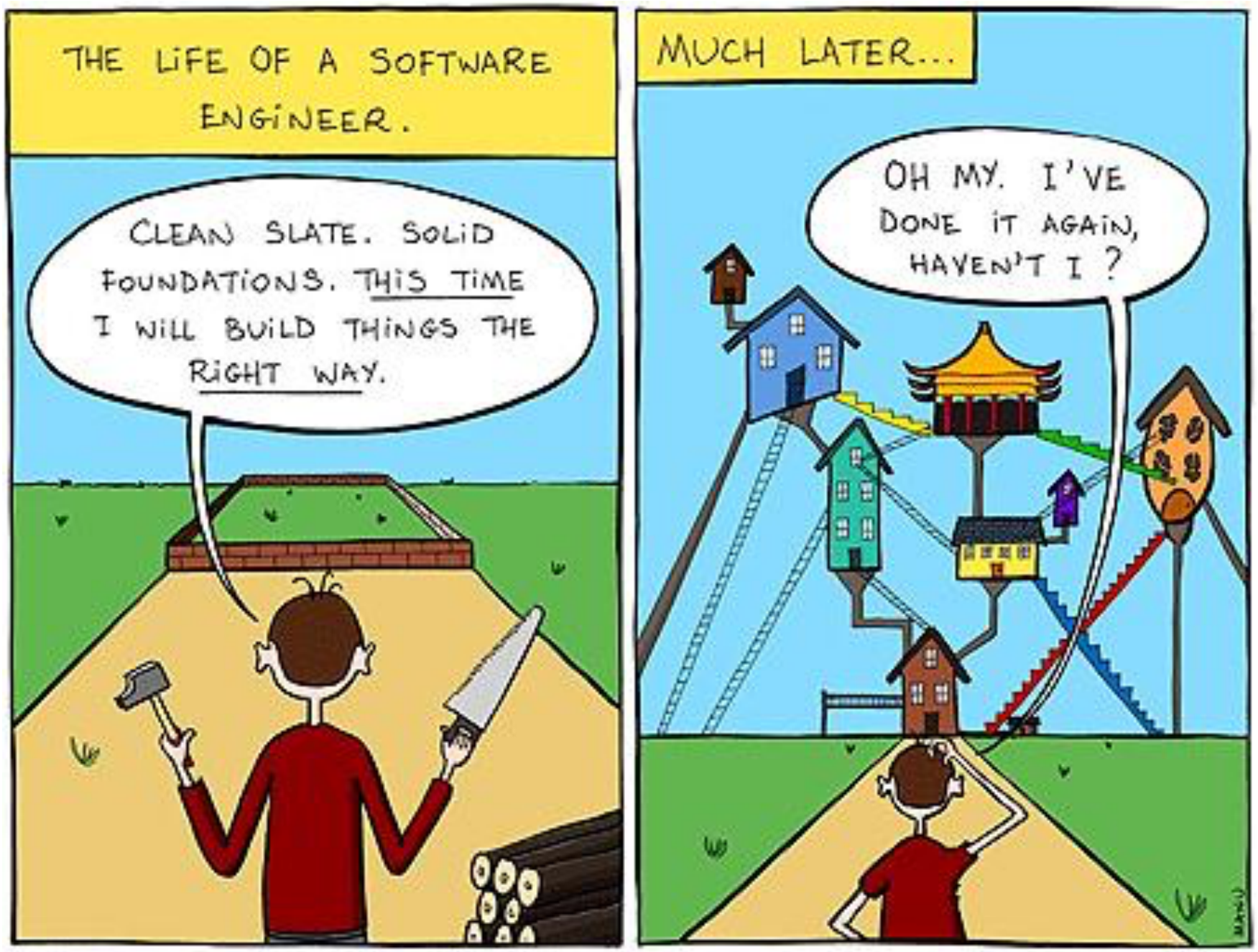
ECOSISTEMA ASYNCIO

- https://github.com/aio-libs
- https://github.com/python/asyncio/wiki/ThirdParty

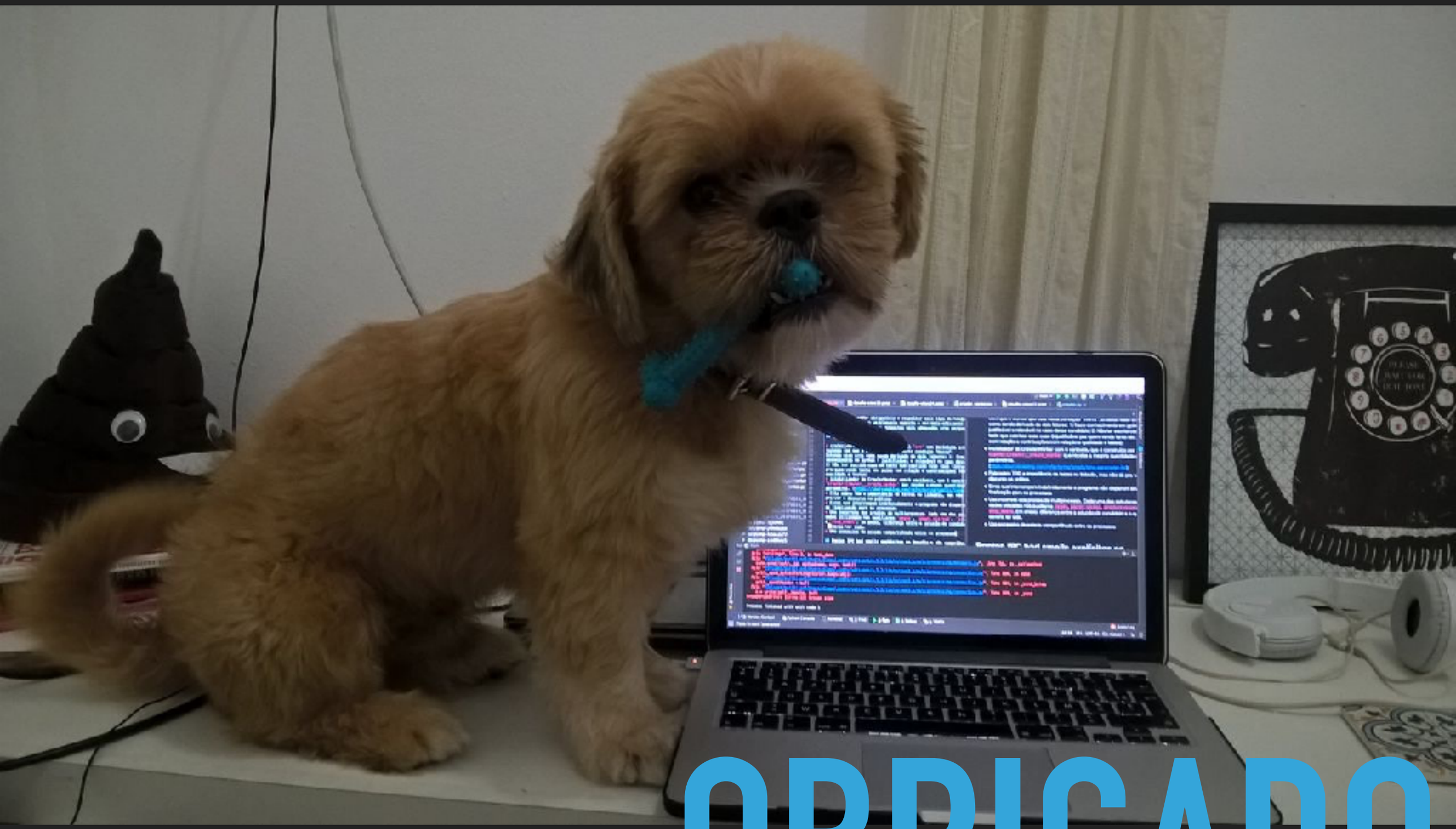
ECOSISTEMA ASYNCIO

	asyncio
pymysql	aiomysql
pyscopg2	aiopg
pymongo	motor
requests	aiohttp
redis	aioredis
unittest	asynctest|pytest-asyncio
pika	aioamqp | easyqueue
paramiko	asyncssh

NÃO USE NADA DISSO



PERGUNTAS ?



OBRIGADO