# Todo lo que -*no*- querés saber

## de **async**

Claudio Daniel Freire
*klaussfreire@gmail.com*

PyCon AR 2018

jampp

BOOSTING MOBILE SALES

# Async te resuelve todo

- Manejar millones de requests a la vez
- Subprocess pero más rápido
- Multiprocessing **y** multithreading pero más rápido
  - Simple y fácil de entender
    - Sin condiciones de carrera
    - Sin locking
- Quita el GIL

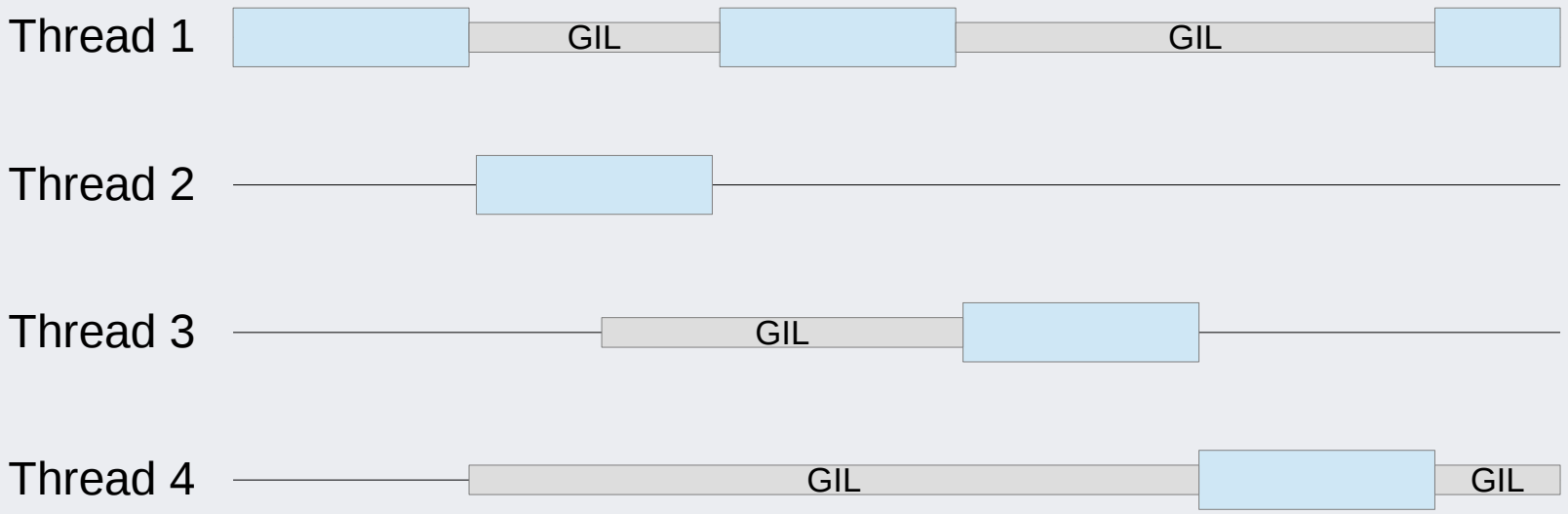
# Async te resuelve todo

Async es **System D**!

# ¿Por qué async?

- Multithreading es **difícil**
  - Condiciones de carrera difíciles de predecir
  - Contención de recursos
  - Locking
- Multithreading es **lento** (en Python)
  - GIL
  - GIL
  - GIL

# ¿Por qué async?

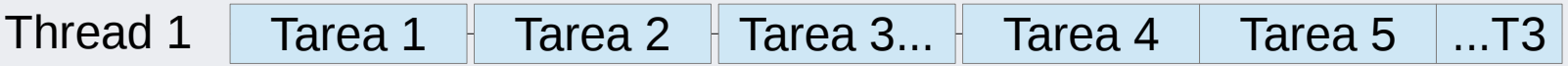



*¿Llamás a eso paralelismo?*

# ¿Por qué async?


*¿Por qué no convertir eso...*

# ¿Por qué async?




*...en esto?*

# ¿Por qué async?

- Y así nace async (versión mundo ideal)
  - Schedulear tareas *asincrónicamente* en uno (o varios) hilos
  - Sin GIL
  - “Sin” context switches (ponele)
  - Sin locks (hay un único thread… ¿no?)

# ¿Por qué async?

**Async:**

```
async def download(sock, f):
    buf = True
    while buf:
        buf = await sock_recv(sock, 1024)
        if buf:
            f.write(buf)

async def download_all(socks_and_files):
    tasks = [
        download(sock, f)
        for sock, f in socks_and_files
    ]
    await gather(*tasks)

get_event_loop().run_until_complete(
    download_all(...))
```

**Sync:**

```
def download_all(socks_and_files):
    sockmap = { sock.fileno() : (sock, f)
        for sock, f in socks_and_files }
    while sockmap:
        readable, _, _ = select(
            sockmap.keys(), [], [])
        for fd in readable:
            sock, f = sockmap[fd]
            buf = sock.read(1024)
            if buf:
                f.write(buf)
            else:
                sockmap.pop(fd)
```

# ¿Por qué async?

**Motivación**

- Cada thread tiene demasiado estado (un stack, mínimo)
  - Hace difícil y costoso tener miles de hilos a la vez
- Levantar un thread es costoso
  - Lo que lleva a threadpools
  - Lo que lleva a cada vez más complejidad
- En muchos casos reales, la mayoría de los hilos:
  - No trabajan todo el tiempo
  - No necesitan mucho estado

# ¿Cómo es async?

- En el mundo real
  - Tiene GIL, de hecho, async es single-thread
  - Tiene context switches. Pero explícitos.
  - Tiene locks
  - Tiene muchos de los problemas de multithreading
    - Más algunos propios

# Veamos antipatrones

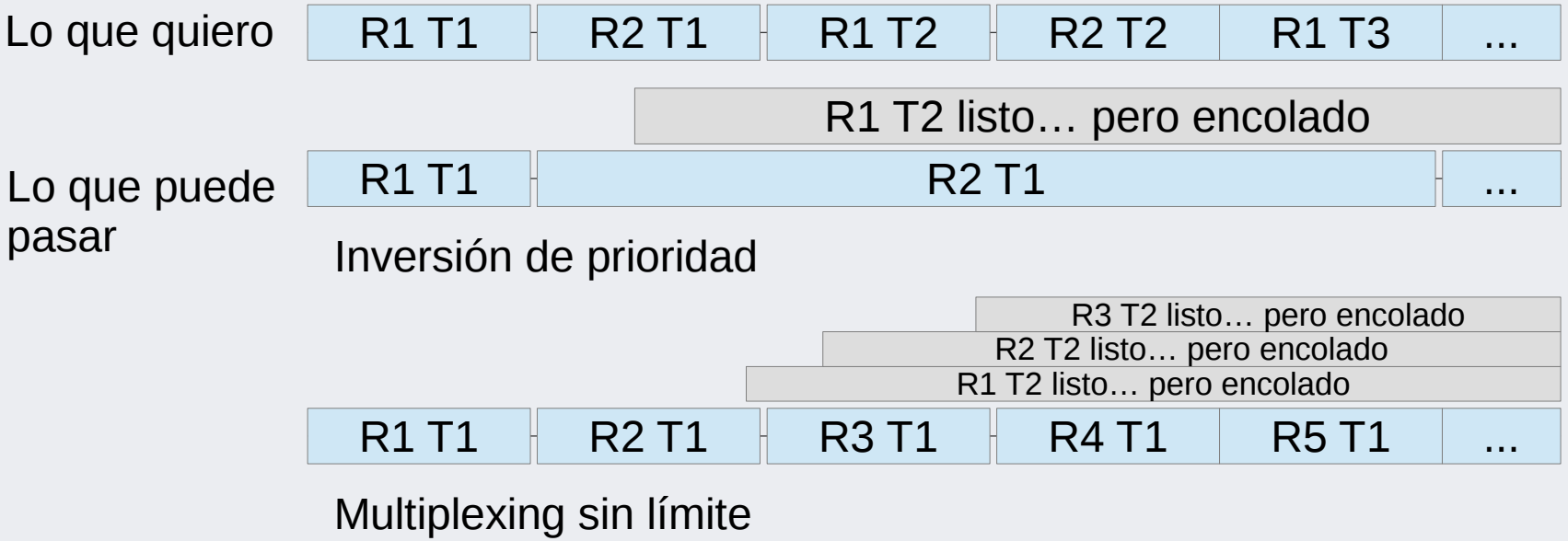
## Usar async para tareas sincrónicas

```
async def process_all(items):
    tasks = [
        process(item)
        for item in items
    ]
    await gather(*tasks)
```

```
async def process(data):
    for whatever in data:
        do_a_lot_of_math(data)

get_event_loop().run_until_complete(
    process_all(...))
```

# ¿Por qué async?



# Veamos antipatrones

## Usar async sin saberlo

- eventlet
  - Es llamar al desastre:
    Agregar concurrencia **nunca** es transparente.
- Usar un I/O loop para hacer operaciones bloqueantes
  - Bloquea el I/O loop y tenés lo **peor** de ambos mundos
- Usar una lib/framework **stateful** dentro de una corutina
  - La lib no conoce el contexto de la corutina
  - Leak de estado desde una tarea a otra = caos

# Veamos antipatrones

Usar async como si fueran threads

- Locks por todos lados
  - Es contención por todos lados
  - Capaz hasta deadlocks
- Usar async sólo para levantar procesos
  - Proceso = mucho estado, para eso poné threads
- Tratar de correr cosas “en background”

# Pero...

- Async sirve para tareas que:
  - Tienen poco estado (menos de unos MB)
  - Pasan gran parte del tiempo esperando
    - I/O, network, recursos
  - Tienen soporte del framework

Pero te lo venden para todo

# Demistificando Async

- Async no te salva de pensar:
  - Si usás concurrencia, incluso con Async, tenés que:
    - Coordinar acceso a los recursos compartidos
    - Limitar la concurrencia si te importa la latencia
    - Elegir una política de scheduling
      - ASAP? FIFO? Fair scheduling?
    - **Pensar** acerca de todo lo de arriba
- Async es más fácil de leer
  - Pero no tanto de razonar ni de depurar
  - Hay condiciones de carrera y todo lo que odiás de threading
    - Sólo que dónde puede haber context-switching lo controlás vos

# Usemos Async “bien”

# Usemos Async “bien”

- No usarlo
  - Si no te reporta un beneficio
  - Si no coopera con tu framework
  - Si no lo entendés

# Usemos Async “bien”

- Realmente “async”
  - Con soporte del framework
    - Tornado
  - Delegar en un threadpool interacción bloqueante. Por ej:
    - Acceso a base de datos a través de un ORM
    - Cómputo intensivo
  - Para tareas que realmente “liberan” el I/O loop:
    - O pasan mucho tiempo esperando eventos o datos
    - O esperan “su turno” según una política de scheduling

# Usemos Async “bien”

- Conociendo tus prioridades y tradeoffs
- Elegí 2:
  - Baja latencia
  - Alto thoughput
  - Legibilidad
- Conociendo el I/O loop

# Usemos Async “bien”

Caso real: tornado 3.1 → 5.0

- Para aumentar thoughput:
  - Acepta múltiples conexiones a la vez
  - Trata de no ceder el control si no necesita esperar I/O
- Resultado:
  - Si tengo 80 procesos y 500 requests y 100ms por request:
    - Tornado 3.1: Cada proceso agarra ~6 requests, termina en 600ms
    - Tornado 5.0: Un proceso agarra 500 requests, termina en 50s

# Usemos Async “bien”

Caso real: tornado 3.1 → 5.0

- Para priorizar latencia:
  - Aceptar una conexión a la vez
- Resultado:
  - De vuelta a los 600ms
  - Un poco de uso de CPU extra como tradeoff

# ¿Preguntas?

jampp

BOOSTING MOBILE SALES