Claudio Freire

Paralelismo en Python

Objetivos

- Aprovechar al 100% los multicore modernos
  - con Python
- Abrirse a la posibilidad de escalar a multi-nodo (clusters)
- Mantener la facilidad de desarrollo característica de Python

Paralelismo
¿parale-qué?

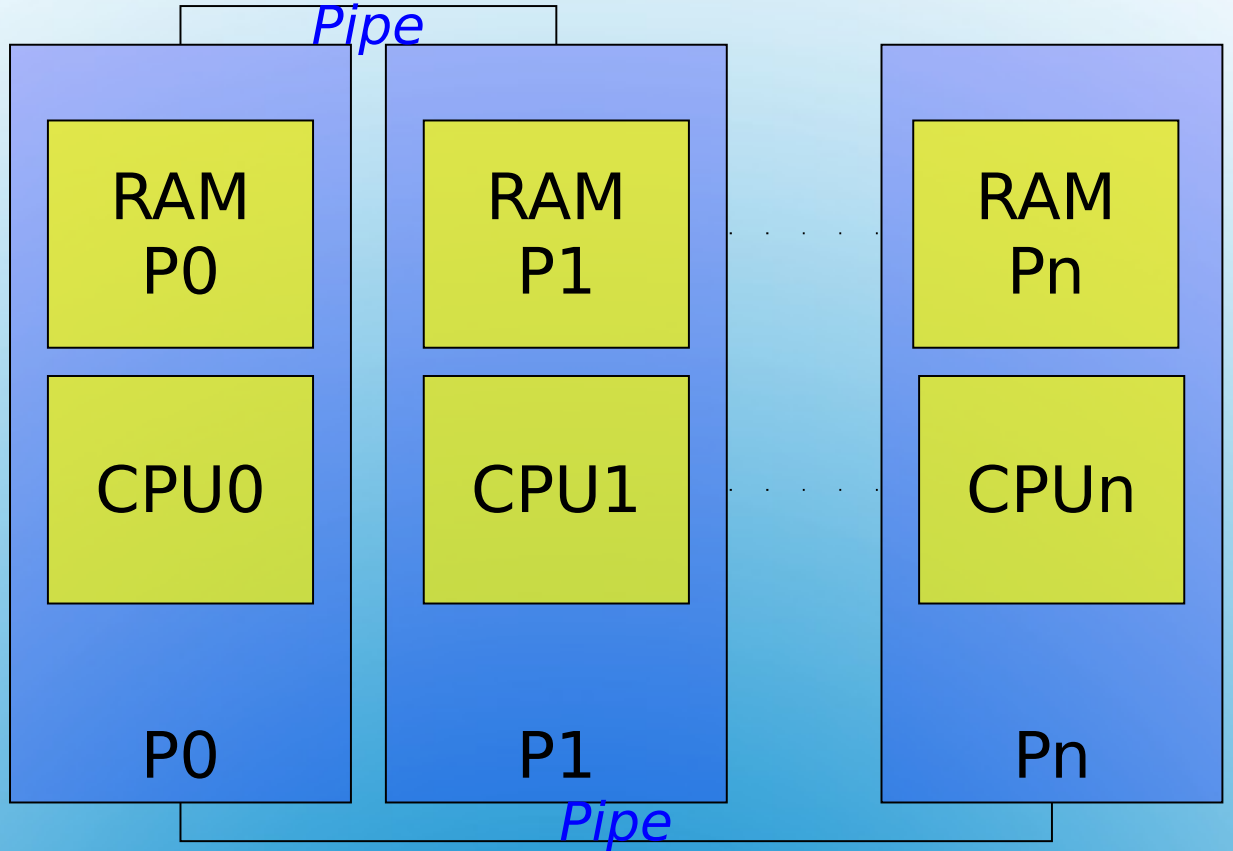
- El arte de dividir una tarea en varias subtareas paralelizables
  - Sumar dos arrays de números
  - Calcular promedios
  - Multitud de otras tareas intensivas en cómputo
- También es realizar varias instancias independientes de una tarea en paralelo.
  - Más transacciones por segundo, cuando son CPU-bound

- Correr varias aplicaciones diferentes en un multicore
- Paralelismo de I/O
  - Más transacciones por segundo, cuando son IO-bound

Paralelismo
...del que no nos interesa

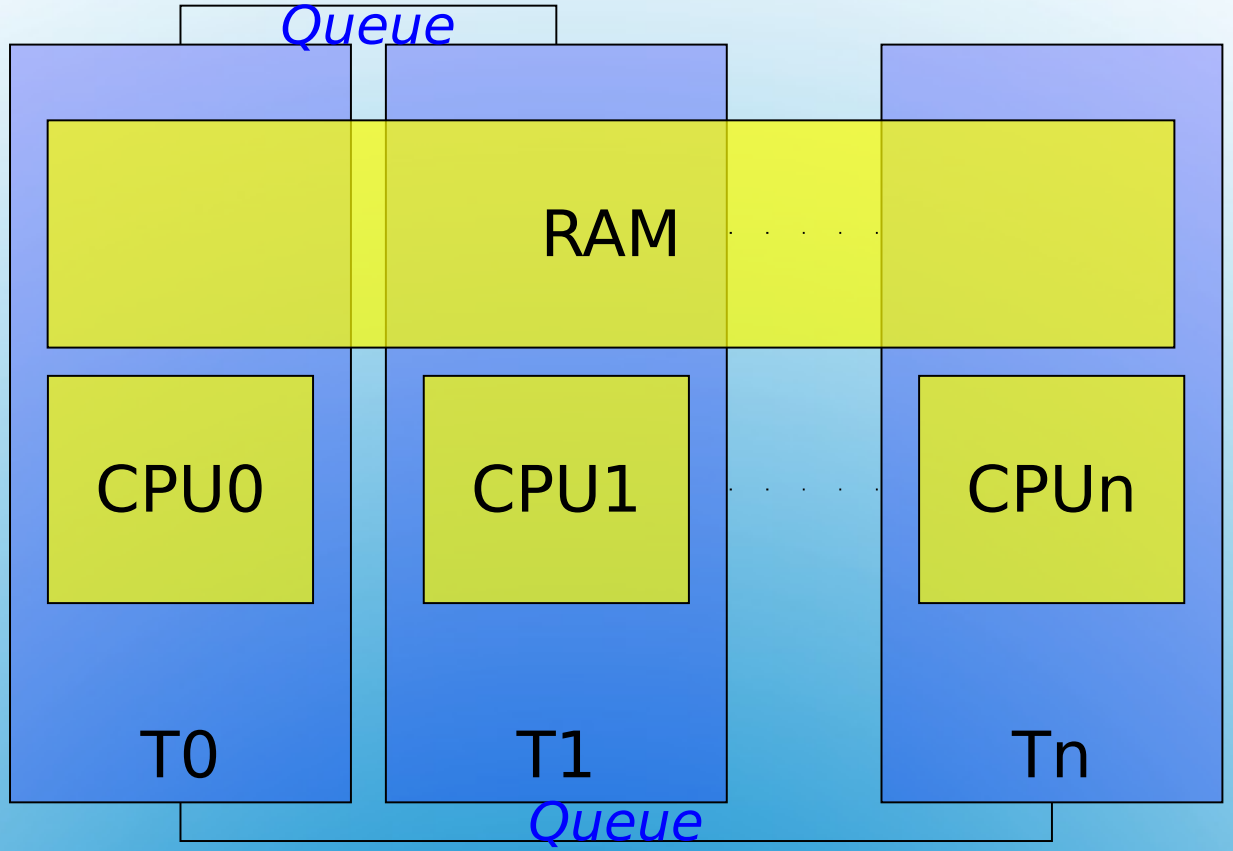
Formas de multiprocesamiento

- Multiproceso
  - Correr varias instancias separadas del proceso, cada una con una parte del “workload”4
- RPC
  - Multiproceso, con un protocolo para pedir la ejecución de procedimientos en otro proceso.
- Multithreading
  - Esencialmente como multiproceso, pero:
    - Corren en el mismo espacio de memoria
    - Se pueden crear más threads y más rápido que procesos



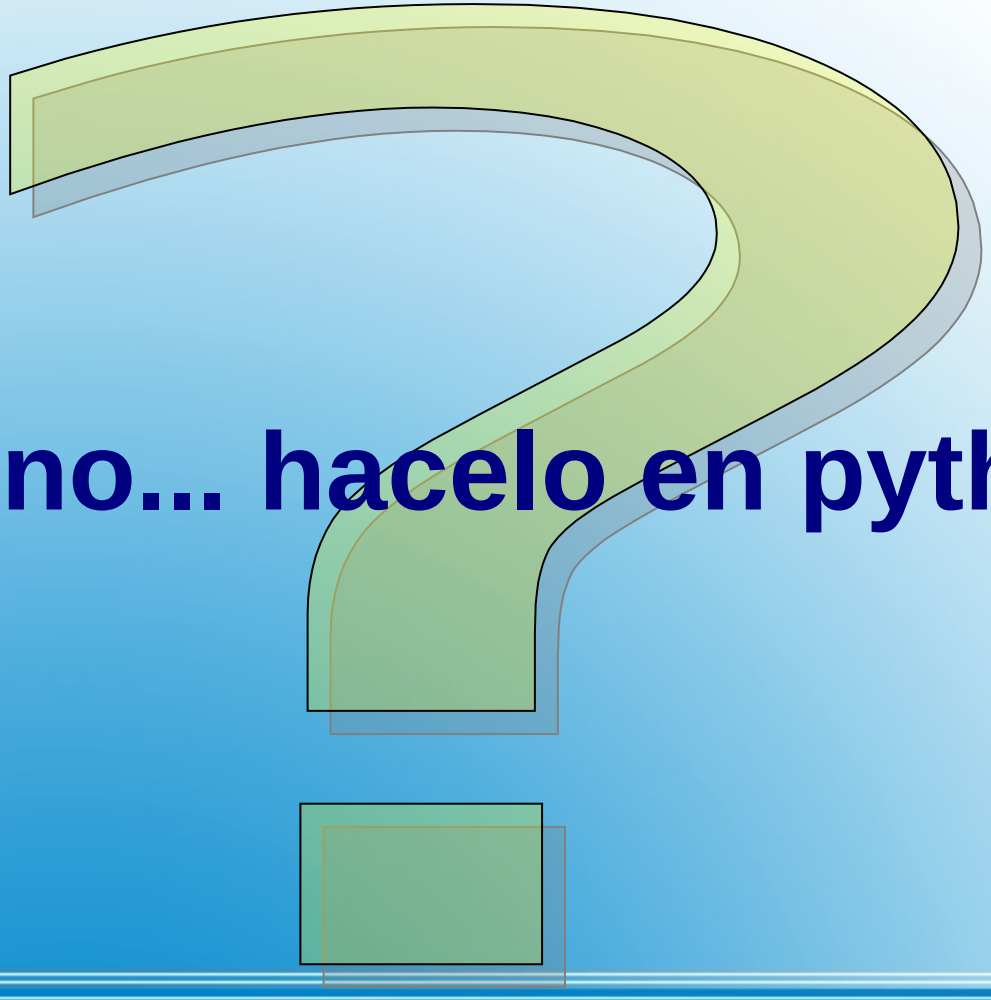
- No comparten nada
  - Excepto CPU si hay < n
- Sólo se comunican por IPC
  - Pipes
  - Sockets
  - Archivos
  - SHM

Multiproceso



- Comparten memoria
  - Y CPU si hay < n
- Se comunican rápidamente mediante estructuras compartidas
  - Colas
  - Buffers
- Requieren sincronización
  - Mutexes
  - Semáforos
  - Transacciones

Multithreading



Bueno... hacelo en python

- threading
- worker pools

Ejemplo

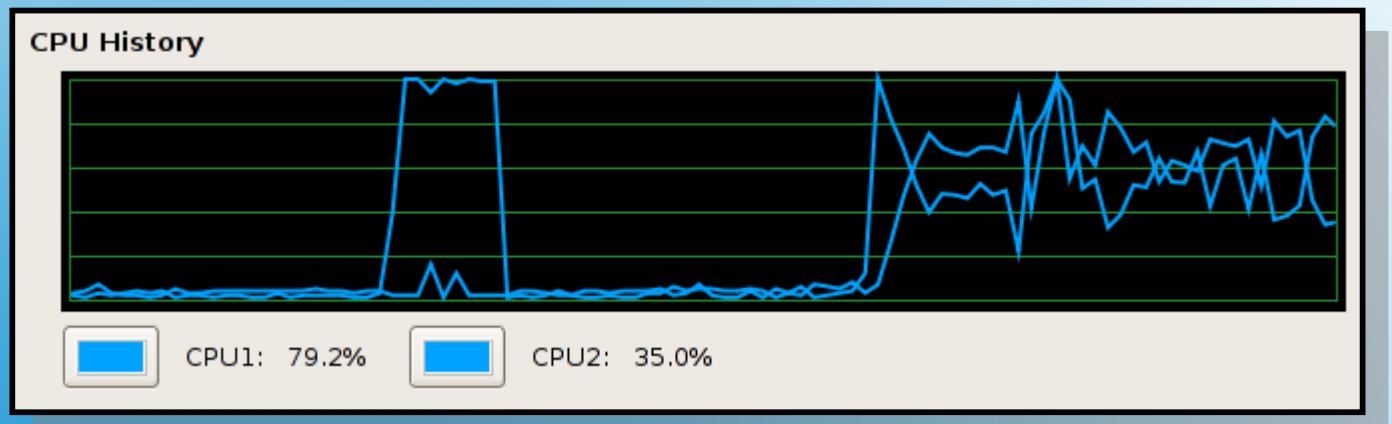
```
>>> def stupid():
...     x = 0
...     while True:
...         x = x + 1
...
>>> from threading import Thread
>>> t1 = Thread(target=stupid)
>>> t2 = Thread(target=stupid)
>>> t1.start()
>>> t2.start()
```

Ejemplo

```
>>> def geturl(url):
...     from urllib import urlopen
...     return urlopen(url).read()
...
>>> u1 = “http://www.google.com.ar”
>>> u2 = “http://ar.livra.com”
>>> from threading import Thread
>>> t1 = Thread(target=geturl, args=(u1,))
>>> t2 = Thread(target=geturl, args=(u2,))
>>> t1.start()
>>> t2.start()
```

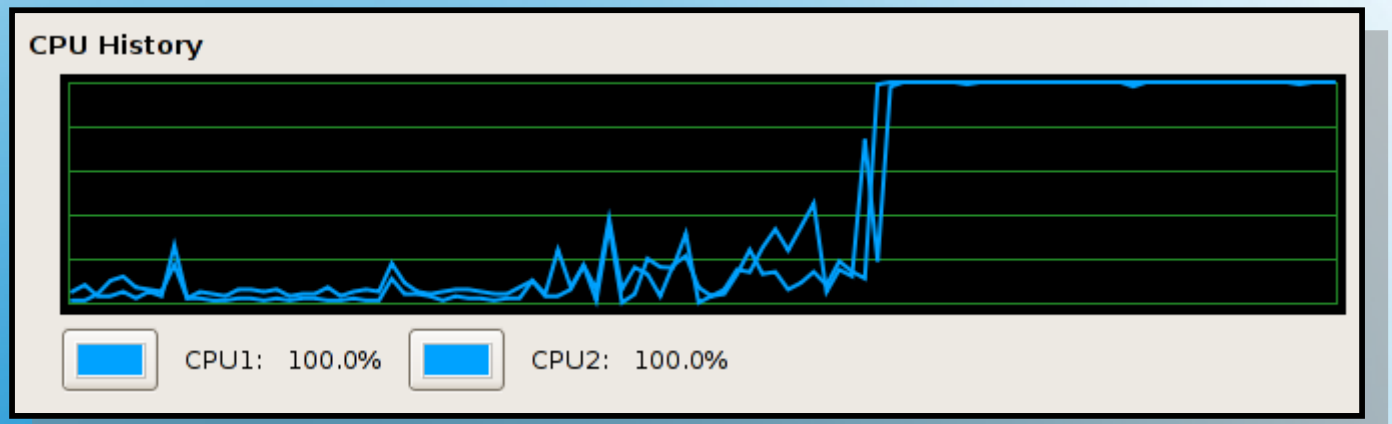
Multithreading en python

```
>>> def stupid():
...     x = 0
...     while True:
...         x = x + 1
...
>>> from threading import Thread
>>> t1 = Thread(target=stupid)
>>> t2 = Thread(target=stupid)
>>> t1.start()
>>> t2.start()
```



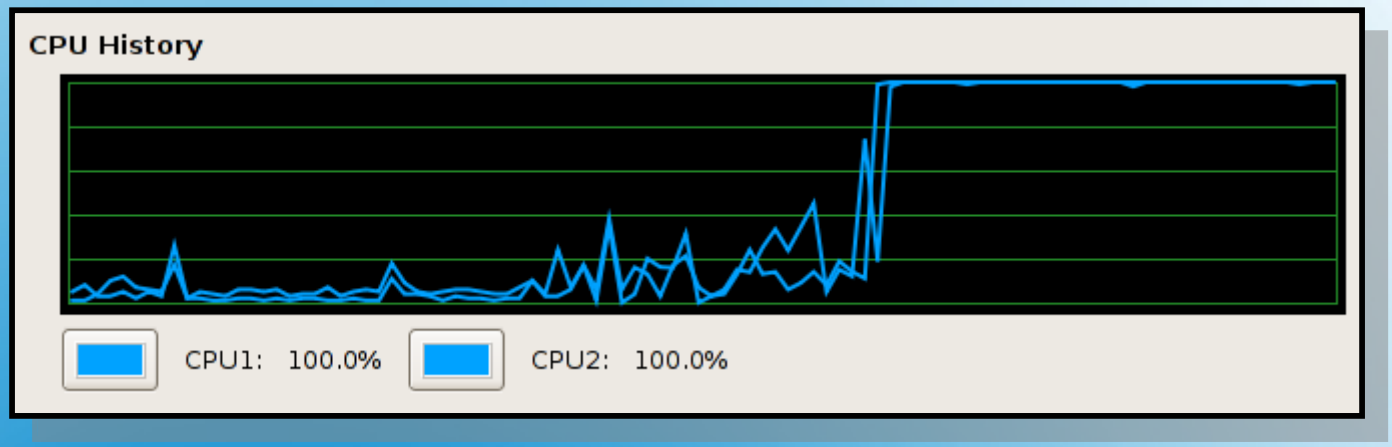
¿WTF?

```
>>> def stupid():
...     x = 0
...     while True:
...         x = x + 1
...
>>> from multiprocessing import Process
>>> t1 = Process(target=stupid)
>>> t2 = Process(target=stupid)
>>> t1.start()
>>> t2.start()
```



¿WTF?

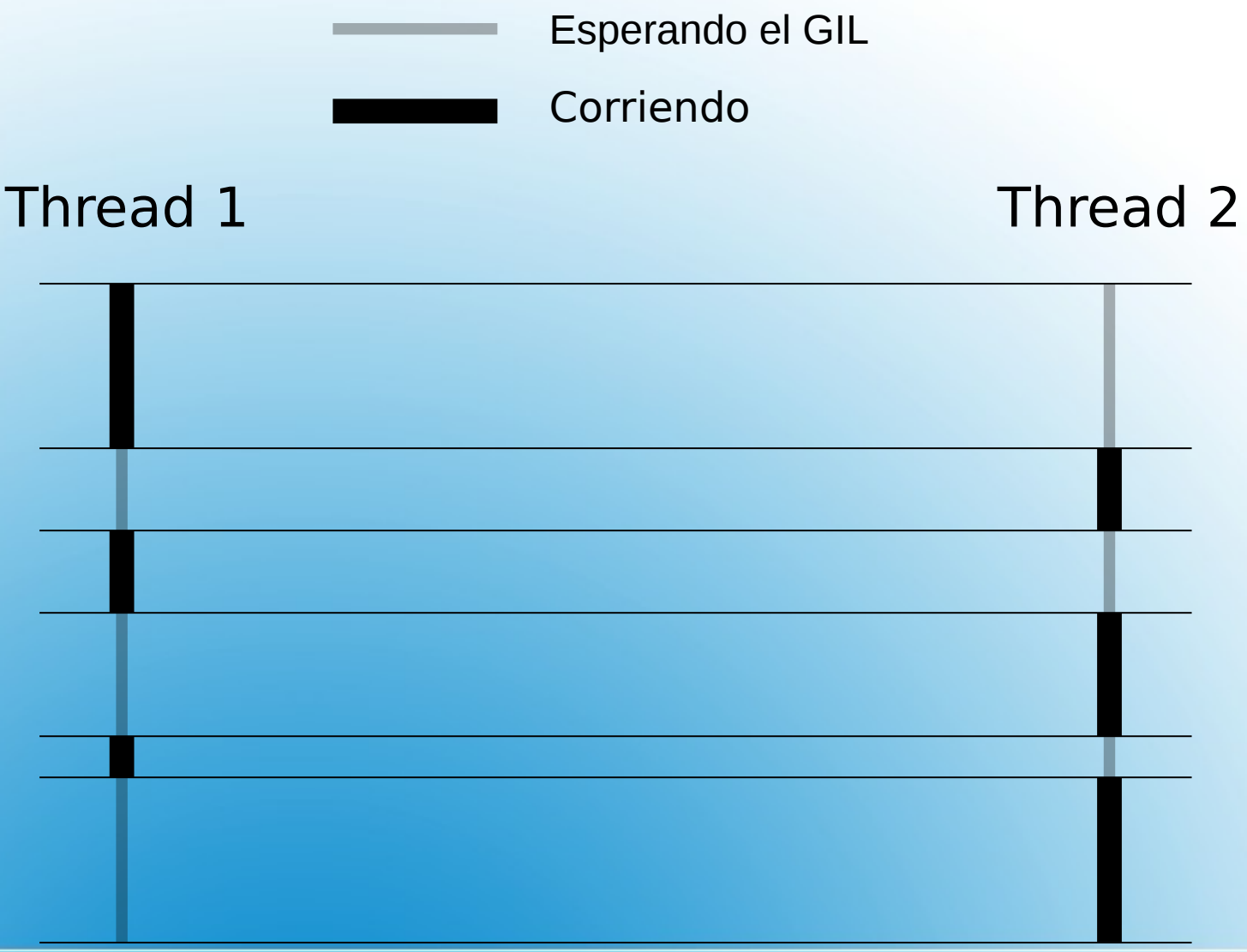
```
>>> def stupid():
...     x = 0
...     while True:
...         x = x + 1
...
>>> from multiprocessing import Process
>>> t1 = Process(target=stupid)
>>> t2 = Process(target=stupid)
>>> t1.start()
>>> t2.start()
```



¿WTF?

- ¿Por qué threading falla?
  - Compartir memoria es peligroso
    - Los threads pueden pisarse entre sí y producir comportamiento no determinista
  - CPython no es thread-safe
    - Si más de una instancia de CPython corriera al mismo tiempo, probablemente funcionaría mal
  - Luego: GIL
    - Global Interpreter Lock
    - En python, no pueden correr dos threads a la vez

¿WTF?



GIL

- ¿Para qué sirven los threads entonces?
  - Paralelizar I/O
    - Python libera el GIL cuando está esperando datos de la red o el disco
  - Simplificar conceptos
    - Hay aplicaciones que son más fáciles de expresar como múltiples hilos de ejecución que como un programa lineal

¿WTF?

- Otras implementaciones
  - PyPy-stm
  - Jython
- Multiproceso
  - fork
  - multiprocessing
  - XML-RPC o ZMQ
  - MPI
- Multithreading
  - Pyrex + with nogil

¿Opciones?

- Otras implementaciones
  - PyPy
  - Jython
- Multiproceso
  - fork
  - multiprocessing
  - XML-RPC o ZMQ
  - MPI
- Multithreading
  - Pyrex + with nogil

¿Opciones?

- **Pros**:
  - ¡Optimiza!
    - Pyrex (simil-Python) genera código **C** y luego código nativo, optimizado por el compilador de C
    - Permite crear estructuras por fuera de python, sin sus ataduras (*GIL*), e incluso con algo de esfuerzo utilizar las grandes librerías de **C++**
  - Por fuera parece python, por dentro es **C**
  - Cuando una sección no necesita tocar estructuras de python, puede liberar el GIL (*with nogil*), permitiendo paralelismo y verdadero multithreading.

Pyrex + with nogil

- **Cons**:
  - Necesita compilación
  - Hay que tener mucho cuidado con el uso de “*with nogil*”
  - Como en **C**, errores de programación pueden resultar en “segmentation fault”
    - Algo que en Python rara vez se ve
  - Si no se libera el GIL, puede resultar en problemas de inversión de prioridad
  - No es python puro

Pyrex + with nogil

- **Pros**:
  - Conceptualmente equivalente a threading
    - Ya se vio un ejemplo de uso, prácticamente idéntico a threading
  - Altamente portable
- **Cons**:
  - Alto overhead de comunicación interproceso
  - Limitado a un único nodo
  - Para tareas complejas, tiene problemas de seguridad, escalabilidad y estabilidad

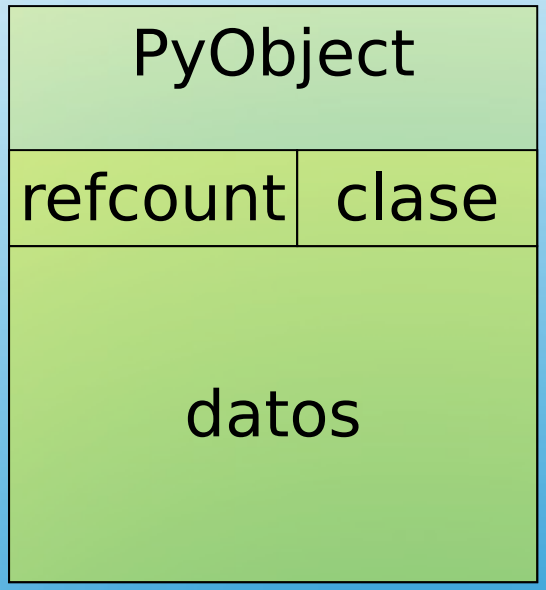
Multiprocessing

Forking es caro

- Multiplica el uso de memoria
  - Todos los datos del proceso padre se duplican en el hijo
  - Salvo *parches mágicos*
- Serialización
  - La única forma de devolver resultados estructurados al padre

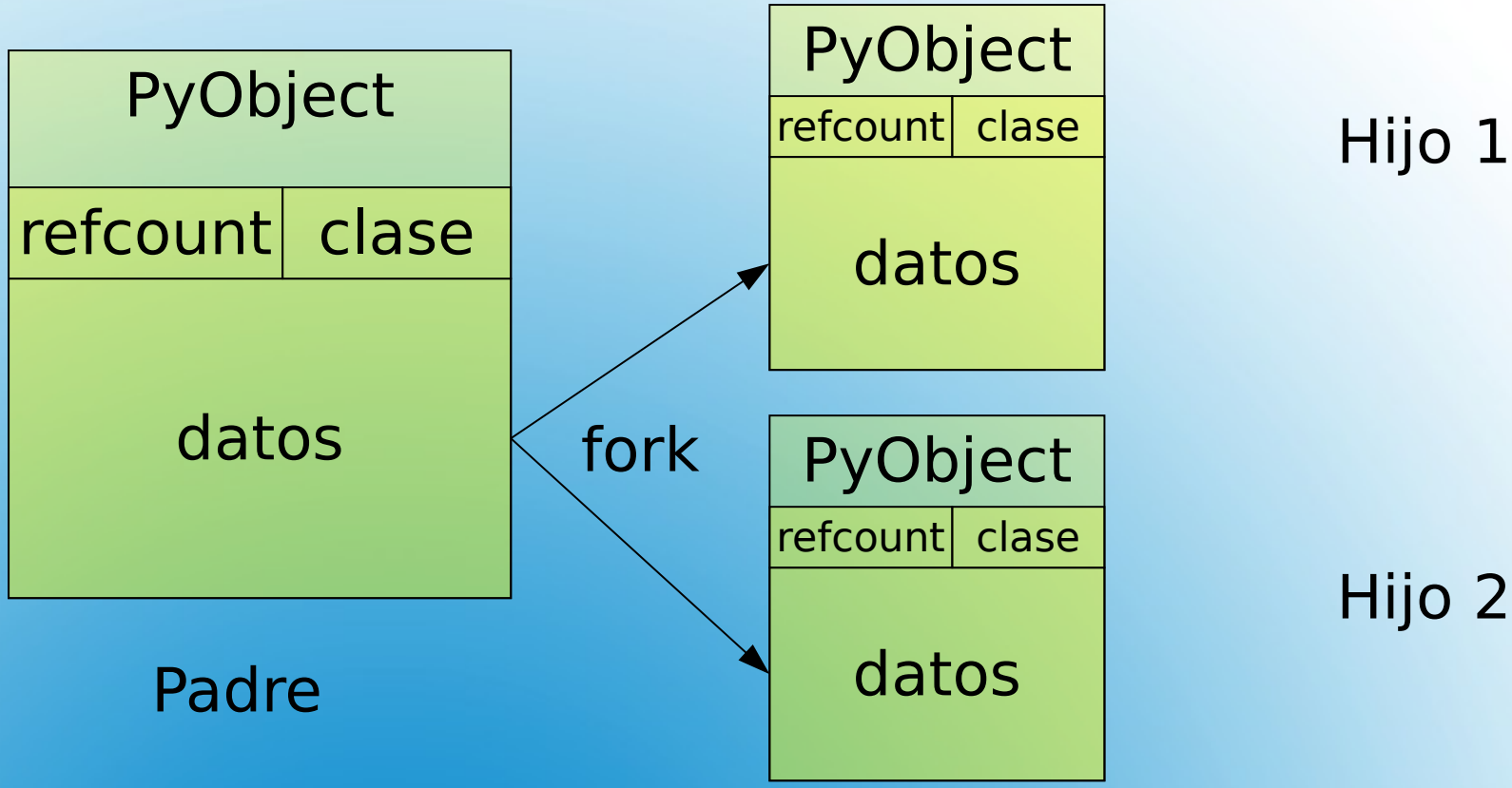
Forking es caro

Multiplica el uso de memoria



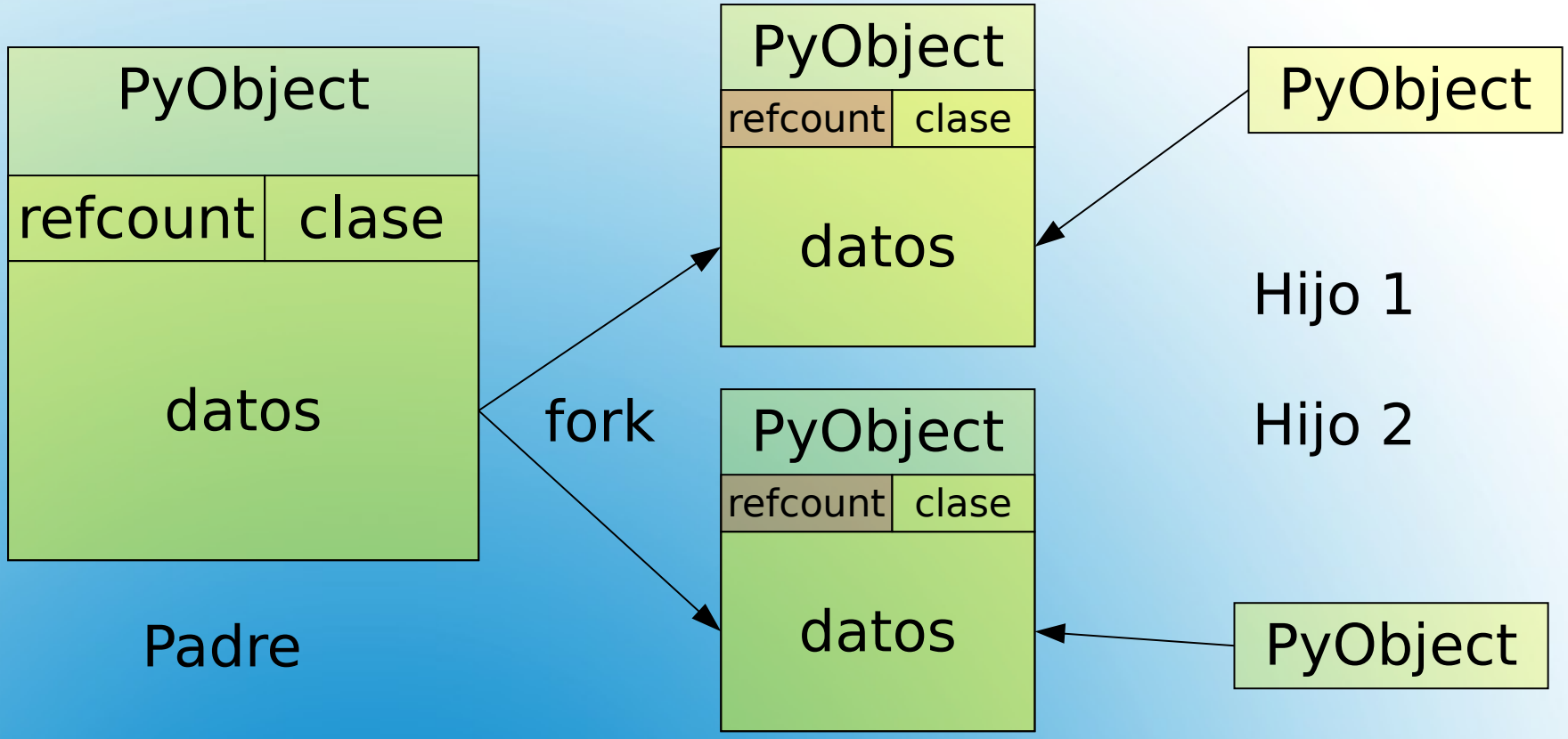
Forking es caro

Multiplica el uso de memoria



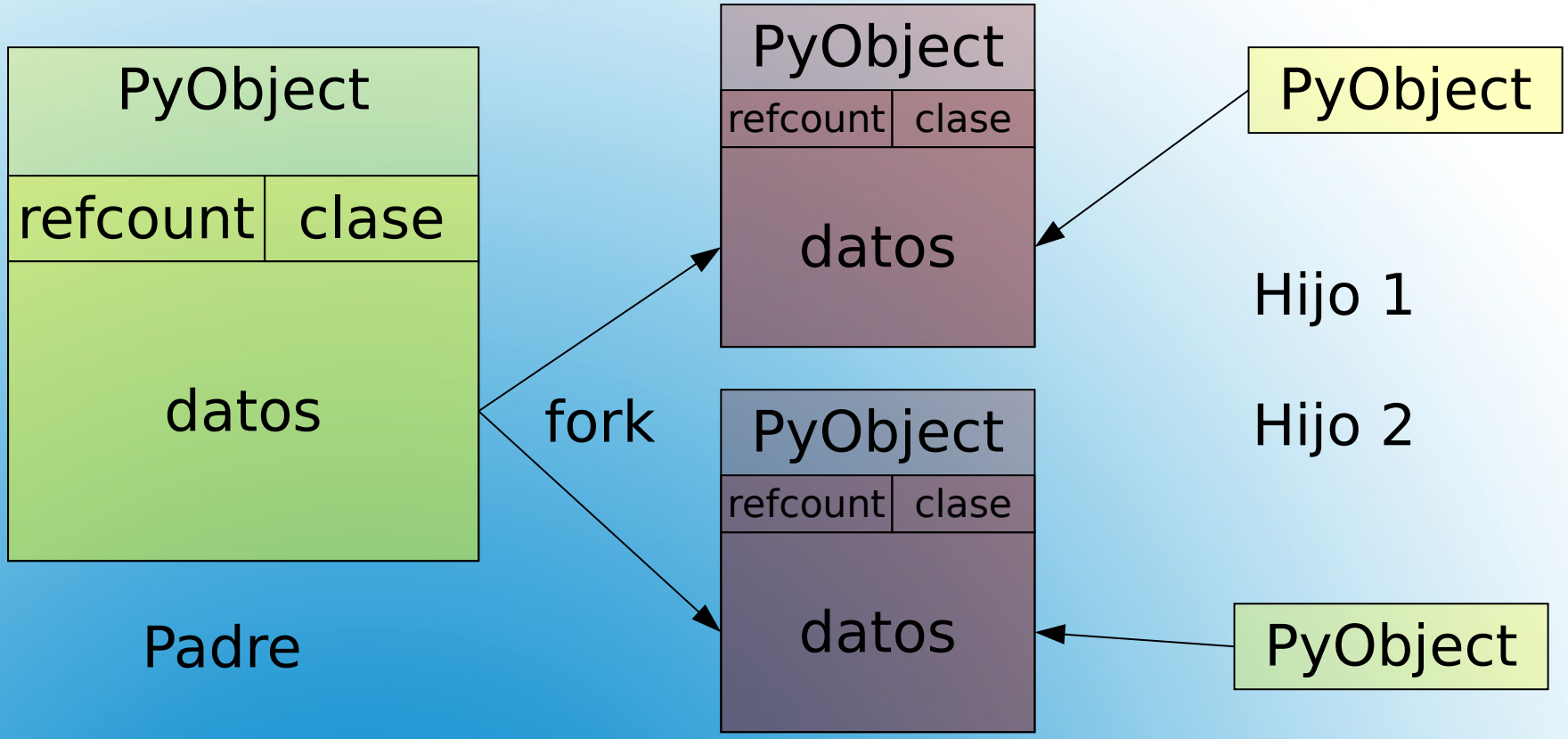
Forking es caro

Multiplica el uso de memoria



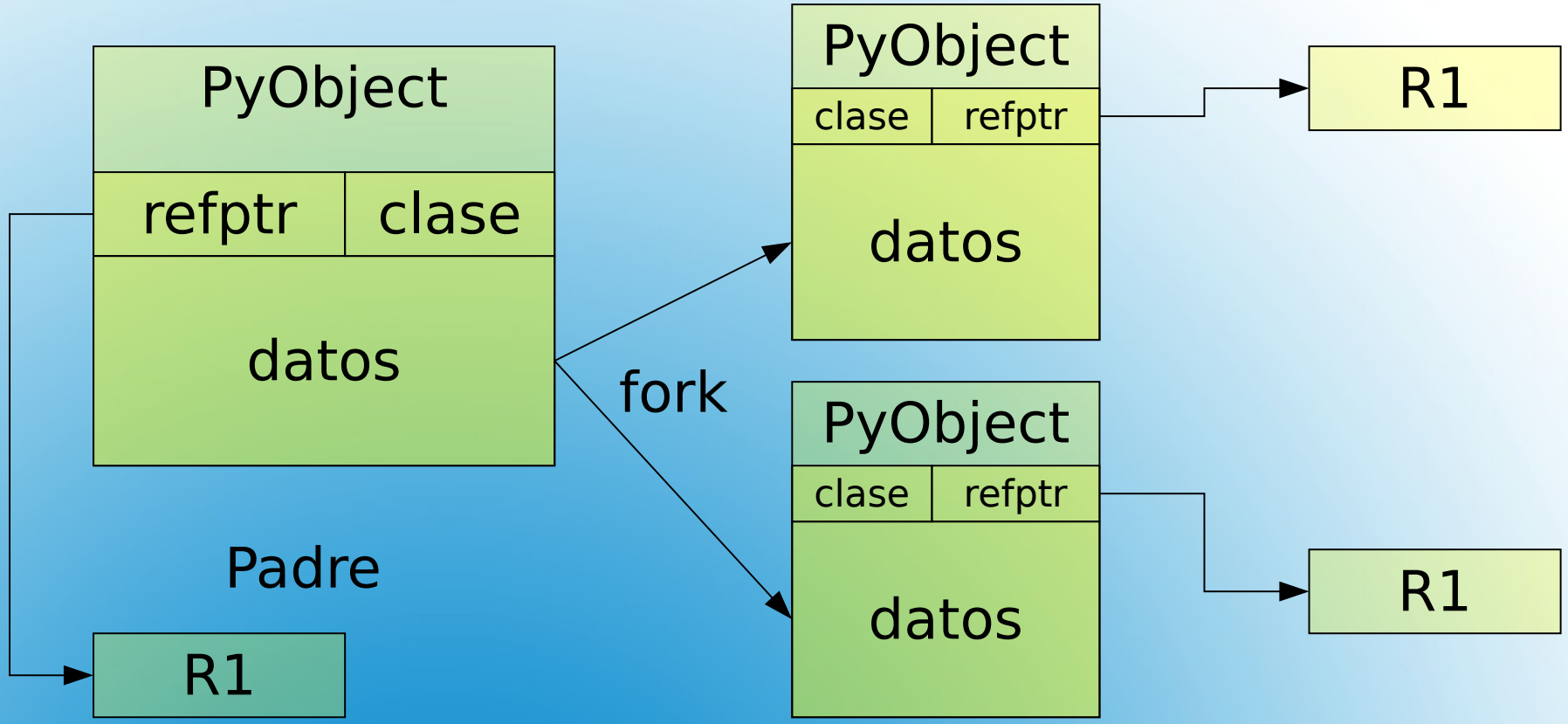
Forking es caro

Multiplica el uso de memoria



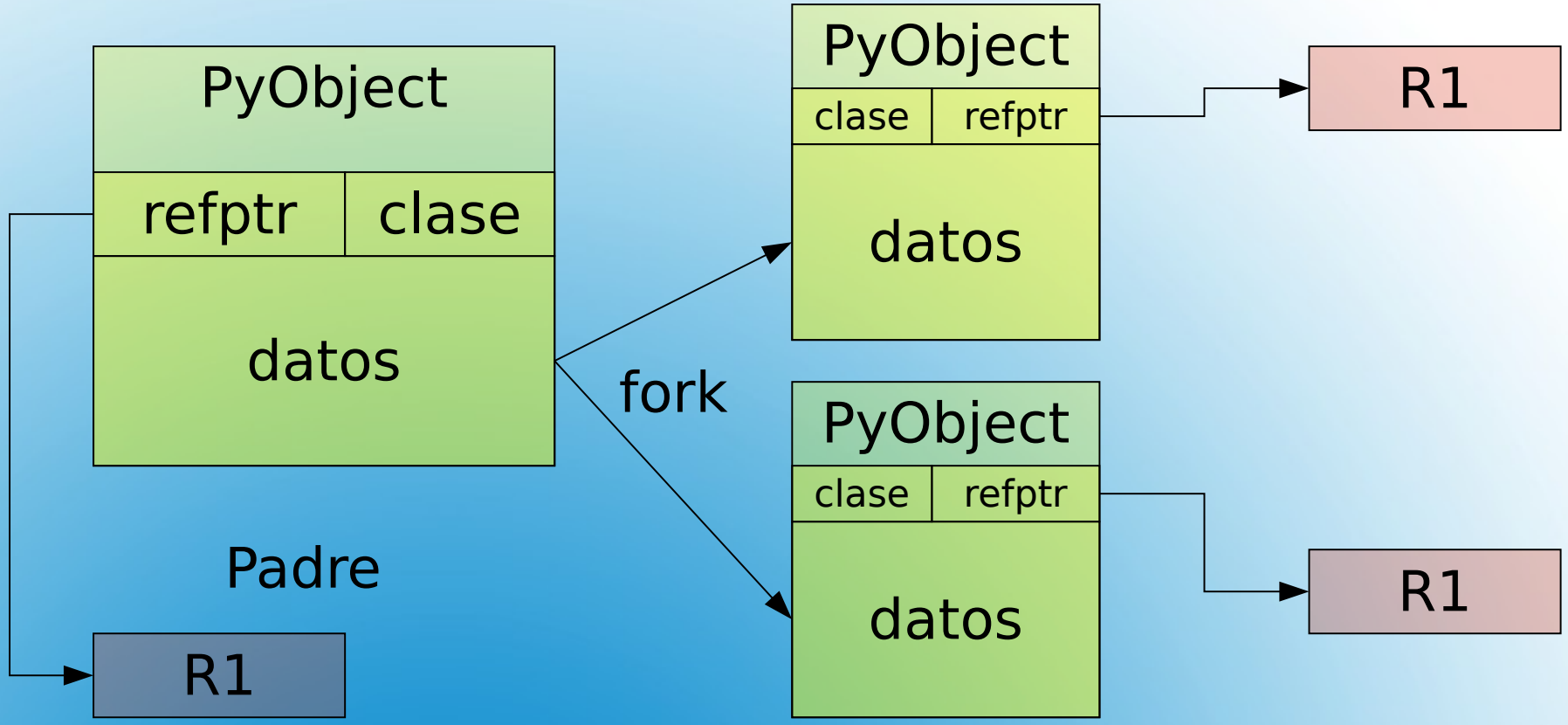
Forking es caro

Emparchando el uso de memoria



Forking es caro

Emparchando el uso de memoria



Moraleja

Para los que no quieren emparchar python

- Un buffer de datos compartido, si no se modifica, no se duplica
- Muchos objetos pequeños compartidos sí
- **Luego**: compactar estructuras
  - Uno o varios numpy.array grandes
  - Uno o varios strings grandes
  - Uno o varios **mmap** grandes

Memoria compartida entre procesos

- mmap permite compartir memoria entre procesos, no solo threads
  - Pero son sólo buffers, no pueden contener objetos
  - Lo que es bueno, porque el GIL existe para proteger operaciones thread-unsafe en objetos

Memoria compartida entre procesos

- mmap permite compartir memoria entre procesos, no solo threads
  - Pero son sólo buffers, no pueden contener objetos
  - Lo que es bueno, porque el GIL existe para proteger operaciones thread-unsafe en objetos
  - Resistir la tentación de usar *pickle*, *struct*, y *marshal*
    - Eliminaría la ventaja de mmap
    - Usar pyrex en cambio, y su habilidad de castear punteros y trabajar directamente en el buffer

Memoria compartida entre procesos

- mmap permite
  - Lockless message queues
  - Buffers compartidos
  - Lookup tables compartidos
  - Proxy Pool
  - Cualquier otra estructura que se pueda trabajar directamente sobre el buffer

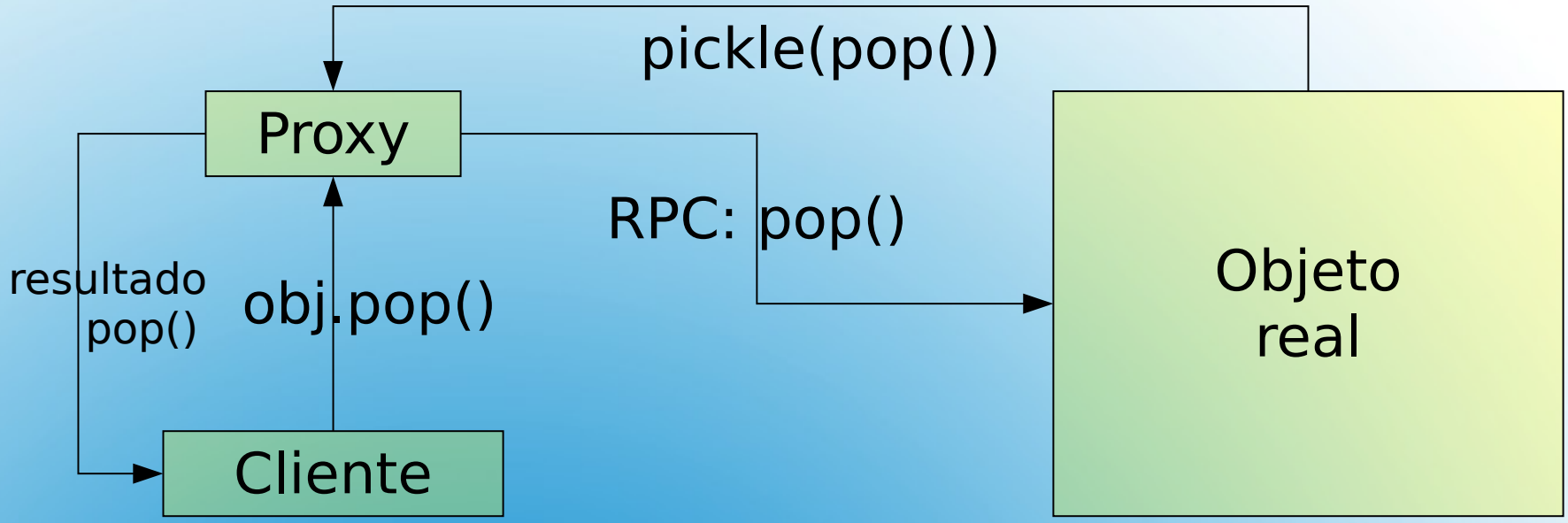
Managers

- Procesos que "administran" objetos remotos
  - Colas compartidas
  - Diccionarios compartidos
  - En fin, objetos compartidos

Managers

- En sí, es un proceso con RPC estandarizado
  - Usa pickle para transferir valores
  - Proxies para representar objetos del lado del cliente
  - Un protocolo interno para representar manipulaciones

Managers


pickle(pop())
Proxy
RPC: pop()
resultado
pop()
obj.pop()
Objeto
real
Cliente

Managers

```
>>> from multiprocessing import Manager, Process
>>> manager = Manager()
>>> d = manager.dict()
>>> d
<DictProxy object, typeid 'dict' at 0x90c410>
>>> print d
{}
>>> p1 = Process(target = lambda : d.setdefault(3,4))
>>> p1.start()
>>> print d
{3: 4}
>>>
```

Managers

- **Seguridad**
  - **Pickle es una máquina virtual, y los sockets del manager son vulnerables a inyección y MITM**
- **Estabilidad**
  - **Incluso sin malicia, son vulnerables a conexiones truncadas, que causan segfault / exceptiones fatales**
- **Escalabilidad**
  - **No hay provisión para customización del protocolo**

Una aplicación compleja
necesita una
Arquitectura compleja

Multiprocessing

Message Passing Interface

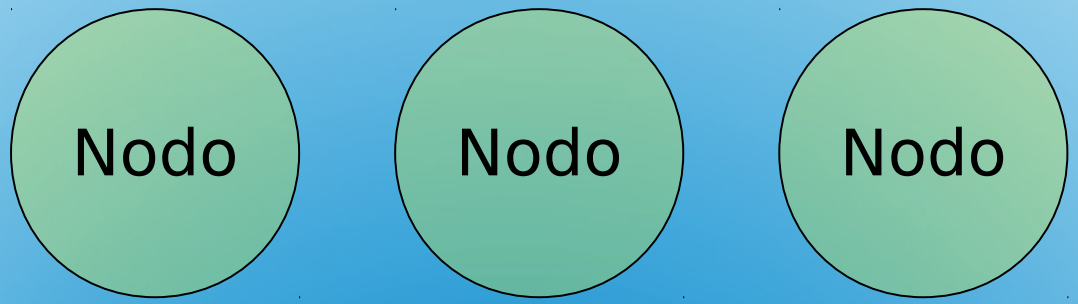
MPI

Message Passing Interface

- No es sólo OpenMPI
- Es un paradigma
  - RPC, OpenMPI, Hadoop, son encarnaciones del paradigma de pasaje de mensajes
  - La diferencia radica en los mensajes que se pasan

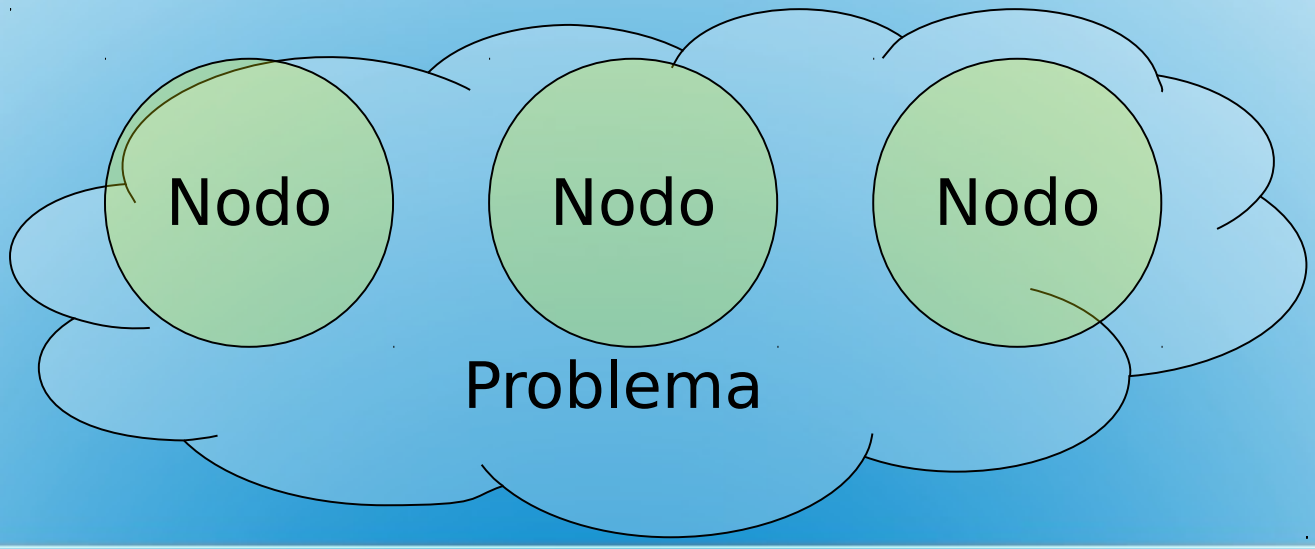
Message Passing Interface

- El paradigma
  - Hay varios nodos



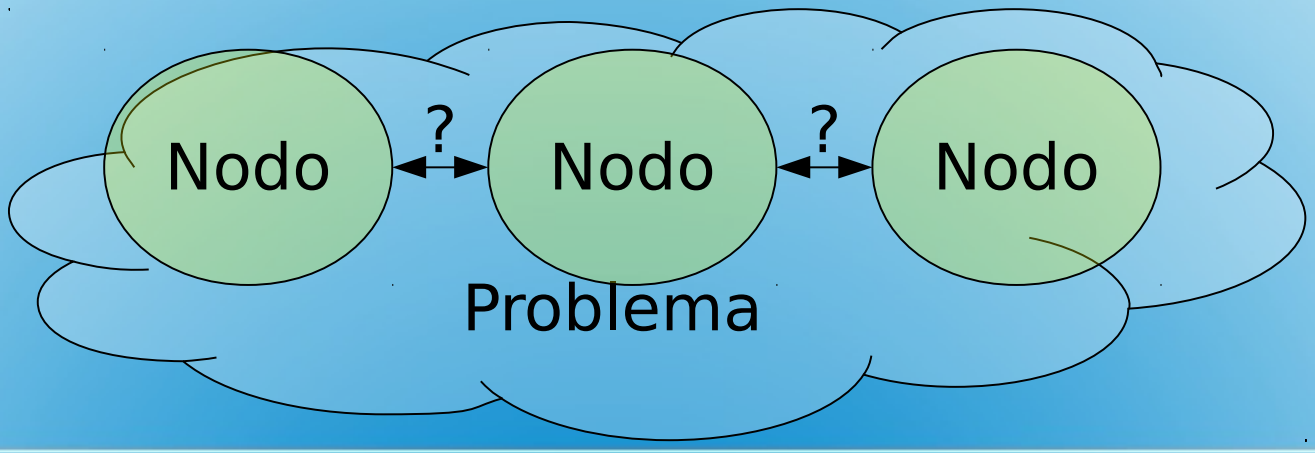
Message Passing Interface

- El paradigma
  - Hay varios nodos
  - Que computan partes de un problema



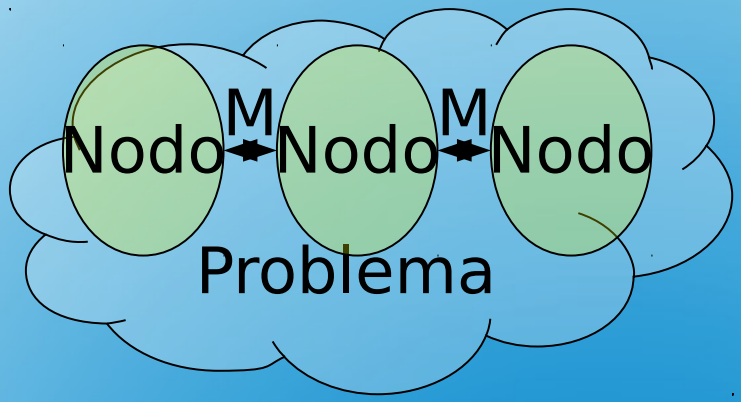
Message Passing Interface

- El paradigma
  - Hay varios nodos
  - Que computan partes de un problema
  - Y coordinan el ensamblaje del resultado, usando...



Message Passing Interface

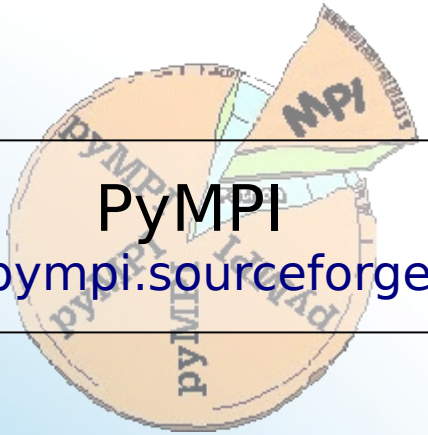
- El paradigma
  - Hay varios nodos
  - Que computan partes de un problema
  - Y coordinan el ensamblaje del resultado, usando...



Mensajes

MPI

- Pros
  - Altamente escalable
  - Estándar
  - Implementaciones heterogéneas
- Cons
  - Orientado a supercomputadoras
    - Esquema de programación complejo
    - Difícil de aprender y programar



PyMPI
http://pympi.sourceforge.net/

OpenMPI

- ejemplo (tomado de la página)

Ejemplo
barrier.py

```
import mpi
data = open('foo%02d.data'%mpi.rank).read()
print 'Work on',mpi.rank
DATA = data.upper()
open('foo%02d.DATA'%mpi.rank,'w').write(DATA)
mpi.barrier()
if mpi.rank == 0:
print 'DONE'

% mpirun -np 3 pyMPI barrier.py
Work on 0
Work on 2
Work on 1
DONE
```

- ejemplo (tomado de la página)

Ejemplo
barrier.py

```
import mpi
data = open('foo%02d.data'%mpi.rank).read()
print 'Work on',mpi.rank
DATA = data.upper()
open('foo%02d.DATA'%mpi.rank,'w').write(DATA)
mpi.barrier()
if mpi.rank == 0:
print 'DONE'

% mpirun -np 3 pyMPI barrier.py
Work on 0
Work on 2
Work on 1
DONE
```

mpi.rank nos dice qué proceso somos

Así podemos hacer que cada proceso haga su parte correspondiente

- ejemplo (tomado de la página)

Ejemplo
barrier.py

```
import mpi
data = open('foo%02d.data'%mpi.rank).read()
print 'Work on',mpi.rank
DATA = data.upper()
open('foo%02d.DATA'%mpi.rank,'w').write(DATA)
mpi.barrier()
if mpi.rank == 0:
print 'DONE'

% mpirun -np 3 pyMPI barrier.py
Work on 0
Work on 2
Work on 1
DONE
```

mpi.barrier hace que todos los procesos sincronicen en ese punto

Similar a fork/join en openMP

- ejemplo (tomado de la página)

Ejemplo
barrier.py

```
import mpi
data = open('foo%02d.data'%mpi.rank).read()
print 'Work on',mpi.rank
DATA = data.upper()
open('foo%02d.DATA'%mpi.rank,'w').write(DATA)
mpi.barrier()
if mpi.rank == 0:
print 'DONE'

% mpirun -np 3 pyMPI barrier.py
Work on 0
Work on 2
Work on 1
DONE
```

MPI necesita un ambiente especial – pyMPI en vez de CPython, a través de mpirun.
Los clusters ya tienen esto, pero es difícil de configurar "de entrecasa"

- Barreras y mutex a través de la red son necesariamente abstracciones de algoritmos complejos
  - Central counter (barrera)
  - Tournament (barrera)
  - Bakery (mutex)
- Lento se queda corto

- Distribución de trabajo por rango
  - mpi.rank define una partición del input
- Unificación de los resultados mediante reducción
  - mpi.reduce, aplica una función a los resultados de cada nodo
- Básicamente un precursor de map-reduce
  - Trivialmente implementable con MPI

- MPI es muy bajo nivel
  (para la mayoría de las aplicaciones)
- Pero rescatemos:
  - Sincronización distribuida
  - Compartir mucho y transferir poco – mensajes
  - Divide & Conquer (o sea: map-reduce)

- **Pros**:
  - Estándar
  - Implementaciones heterogéneas
  - Escalable a múltiples nodos
  - Permite recrear conceptos básicos de MPI
- **Cons**:
  - Difícil comunicación interproceso
    - Alto overhead (más que multiprocessing)
    - Esquema rígido
    - No es transparente
  - Afecta al diseño de la aplicación
    - No es un mero detalle de la implementación
  - No es batteries-included

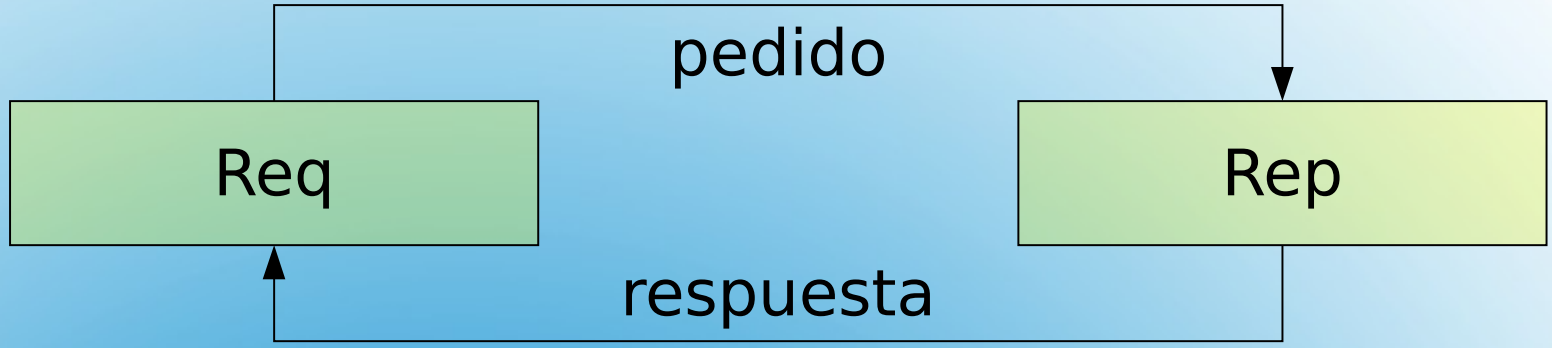
ZMQ

- Mucho mejor para colas de tareas
  - Eficiente en un solo nodo
    - Usando el modo IPC
  - Escalable a múltiples nodos
    - Usando TCP
  - Sincronización mediante PUB/SUB

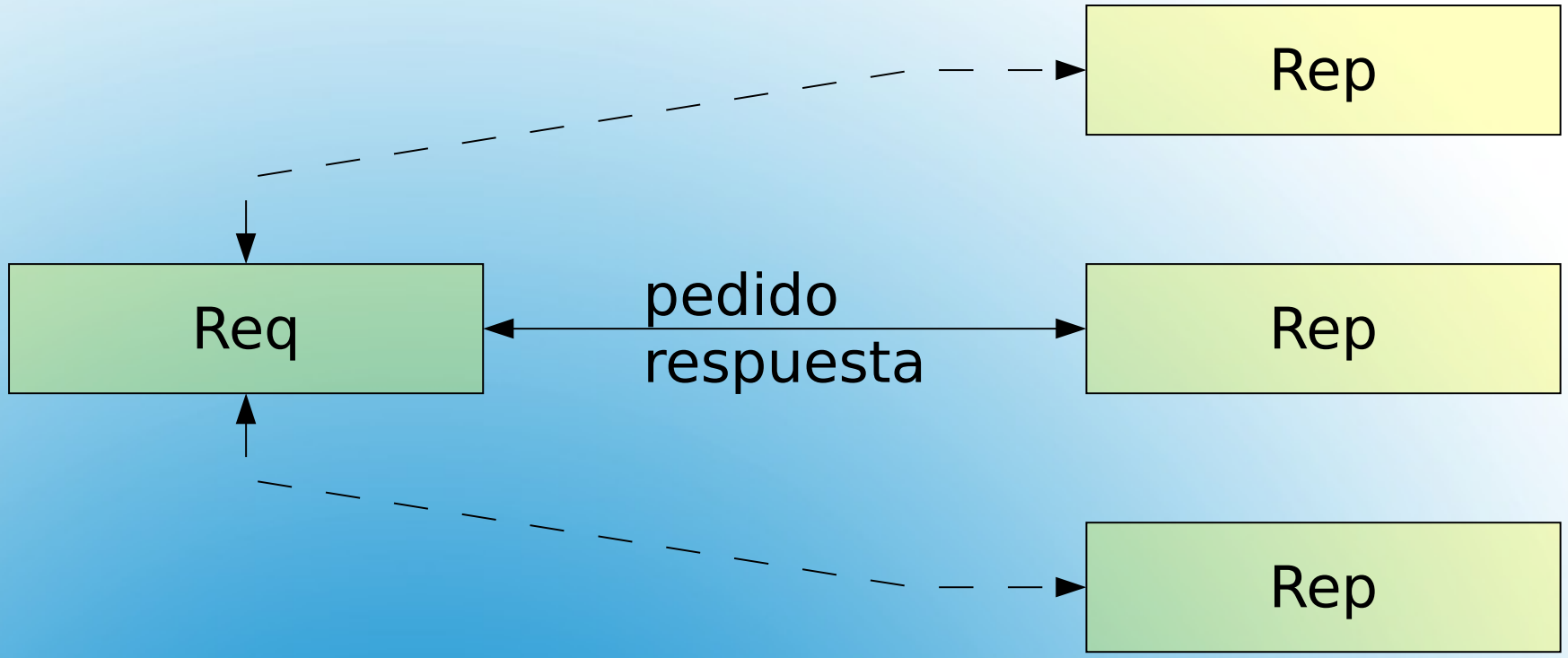
- Req / Rep
- Push / Pull
- Pub / Sub

ZMQ
patrones

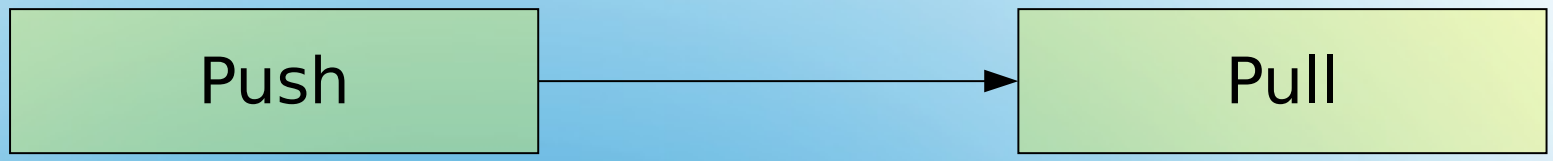
- Req / Rep



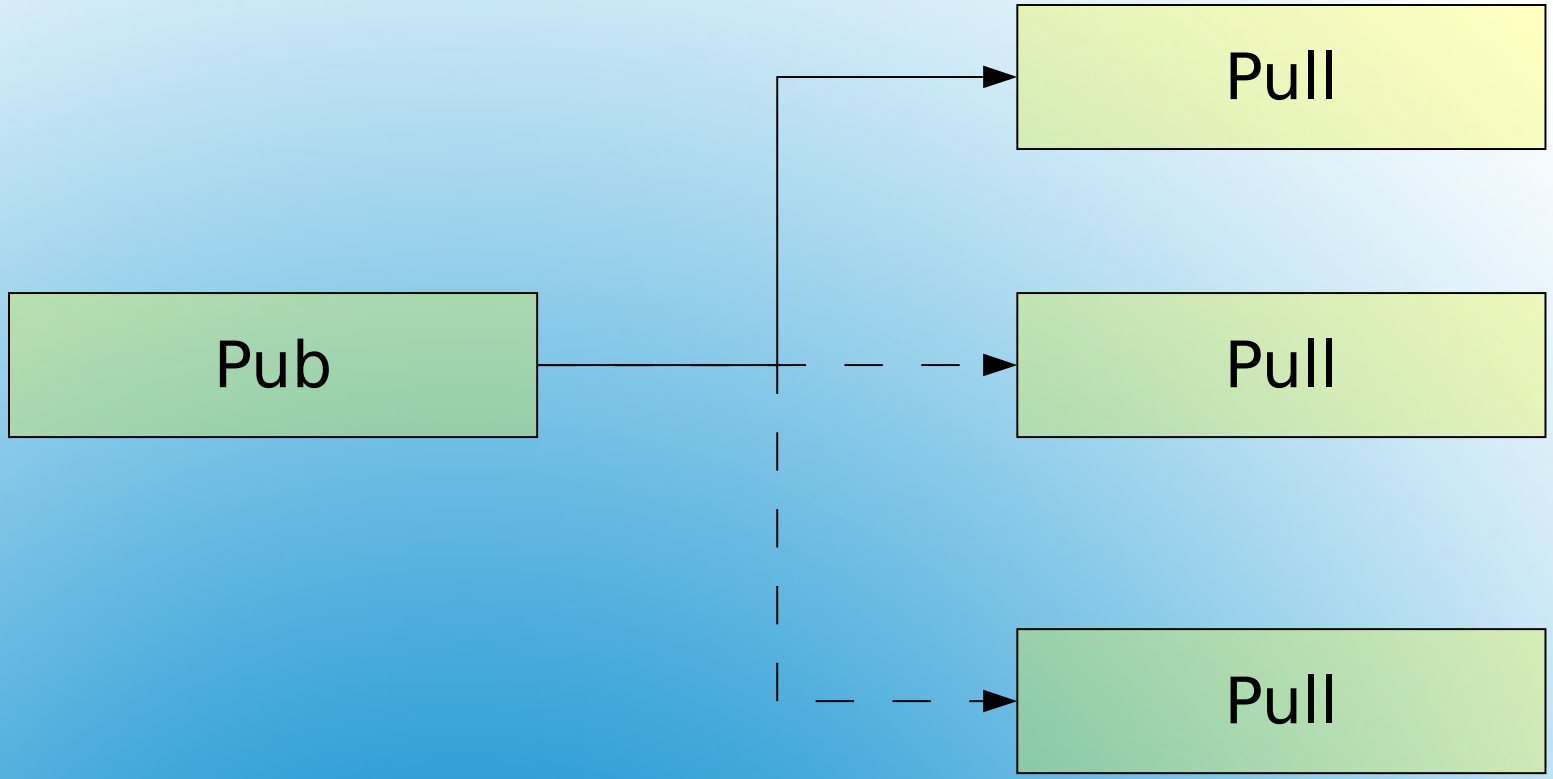
- Req / Rep



- Push / Pull

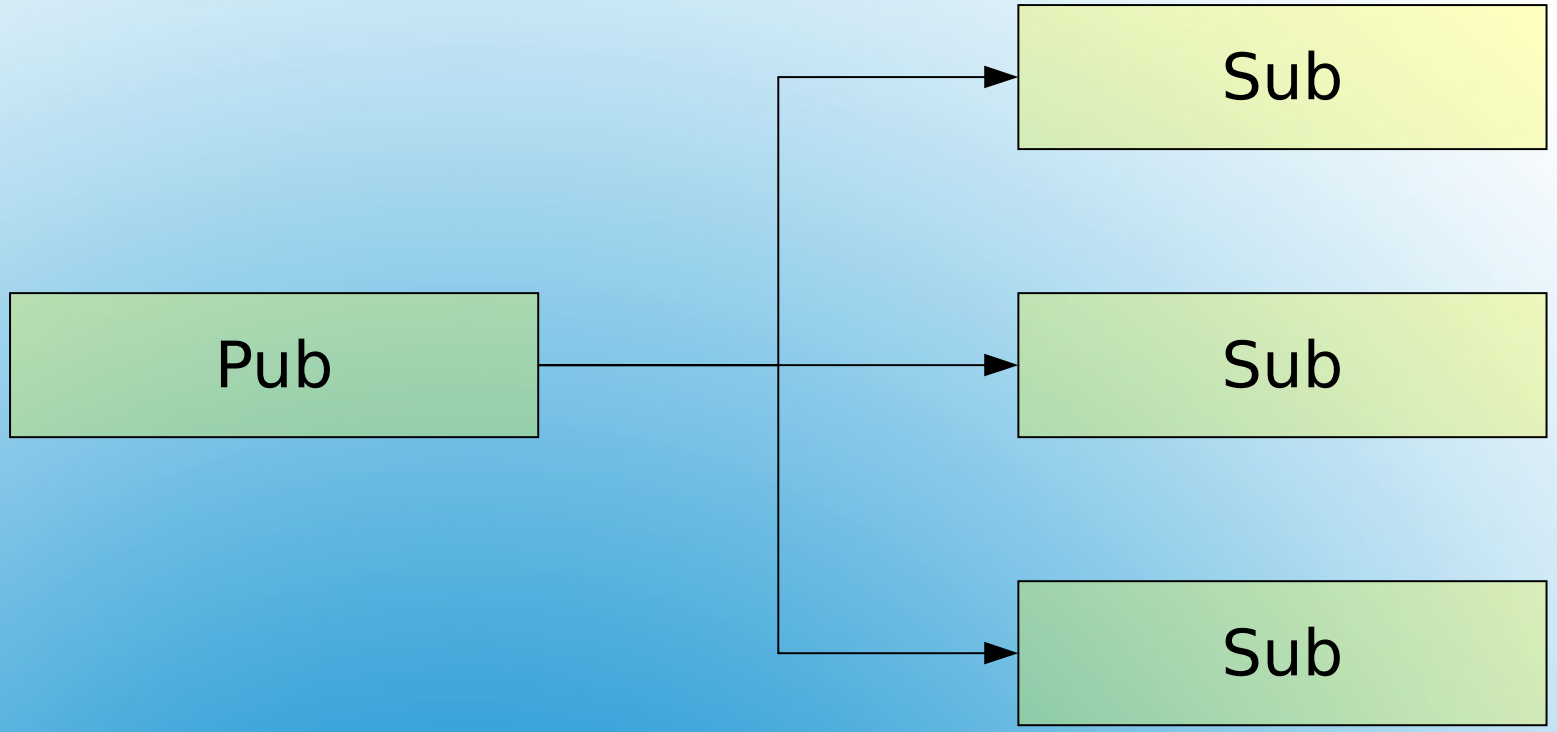


- Push / Pull



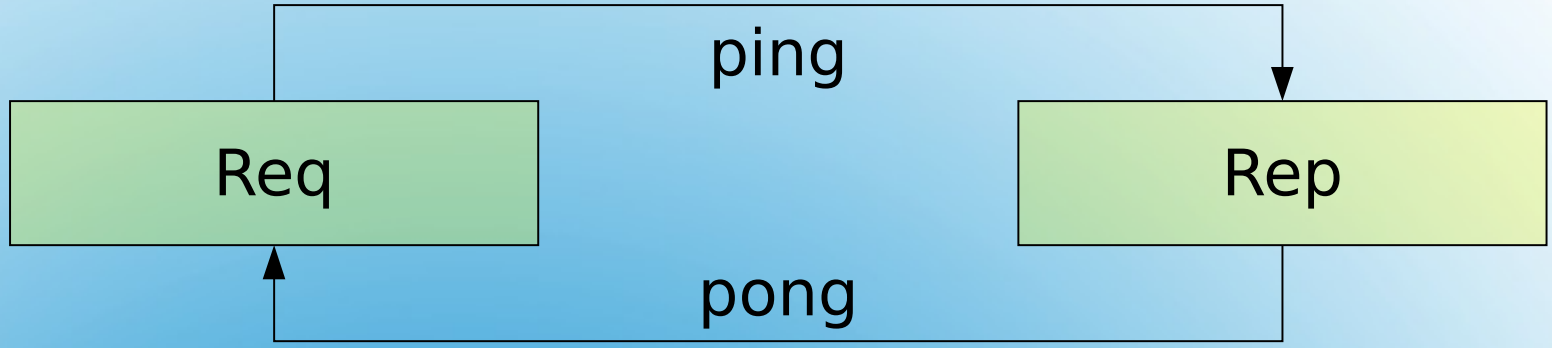
ZMQ
patrones

- Pub / Sub

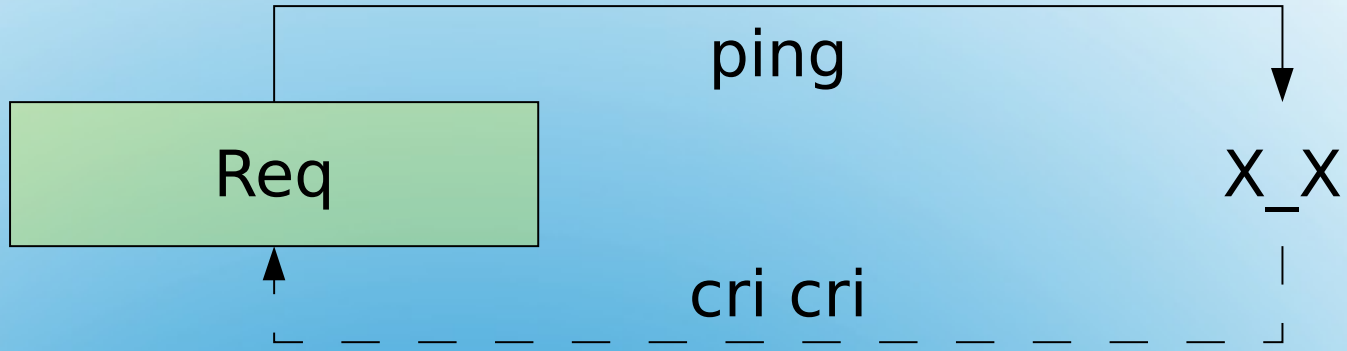


ZMQ
patrones

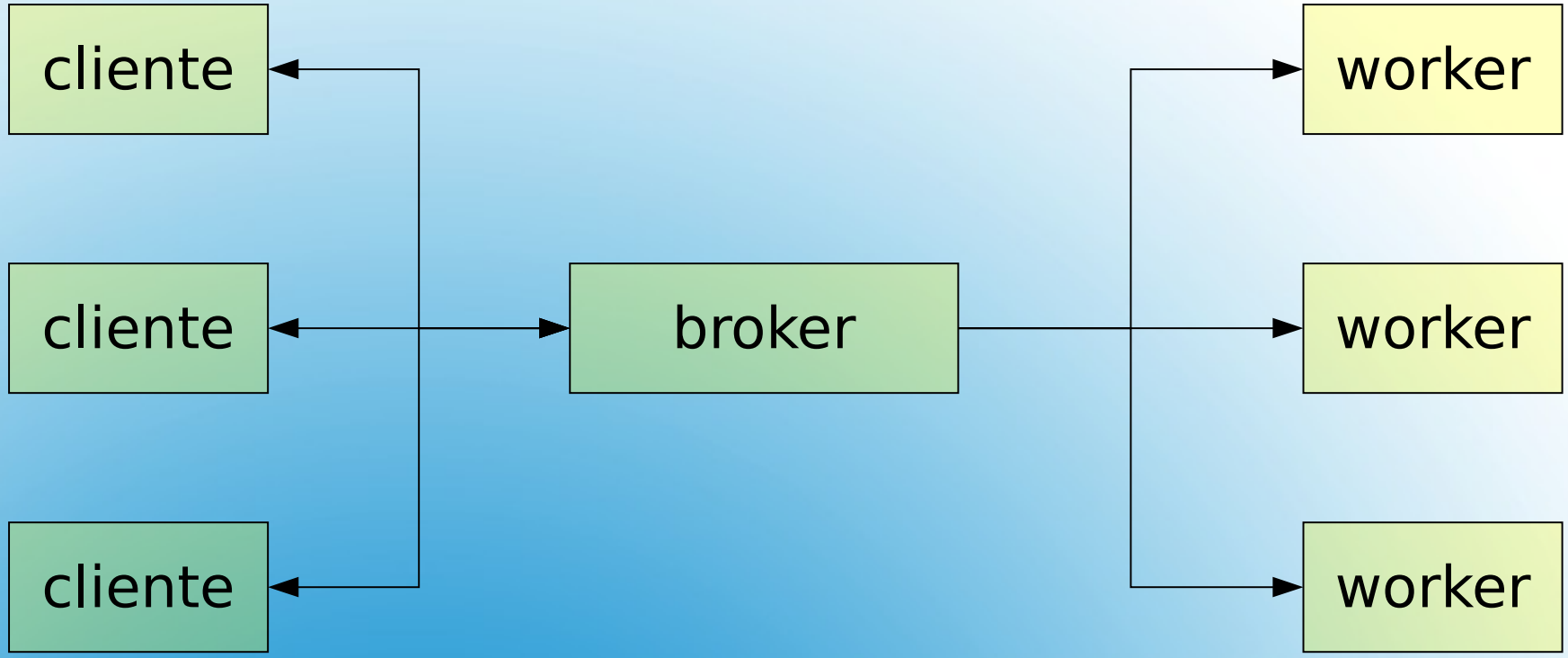
- Heartbeating



- Heartbeating



- Majordomo



- Los patrones funcionan...
  - ...con threads
    - usando el transporte **inproc** no implica overhead alguno... más allá del de serialización
  - ...con multiprocessing
    - usando el transporte **ipc** no implica latencia ni tráfico de red, va todo por pipes
  - ...en clusters
    - usando **tcp**
    - o incluso **multicast** (pgm)

- Zero-copy
  - Si se trabaja con gran volumen de datos, los brokers van a tener que barajar MB/s
    - Se hace **necesario** evitar copias innecesarias
    - ZMQ soporta zero-copy, usarlo
  - Si uso ZMQ para colas de trabajo a través de IPC
    - Gran parte del costo de serialización está en las copias, zero-copy lo reduce considerablemente
  - Si se escriben librerías genéricas
    - Zero-copy para evitar overheads imprevistos cuando se envían mensajes enormes

- Referencias a almacenamiento compartido
  - Si los mensajes son grandes
  - Si hay muchos intermediarios
  - Si el almacenamiento compartido es distribuido
    - O si es memoria compartida
- Reduce o distribuye mejor el tráfico de red
- Reduce el costo de serialización

ZMQ
mensajes indirectos

Ejemplo

```
import zmq, numpy.core.memmap, multiprocessing, threading, cPickle
data = open("/dev/shm/bigarray", "w+b") ; data.seek(1<<24) ; data.truncate()
mdata = numpy.memmap(data, numpy.float32)
mdata[:] = numpy.arange(1<<22)
ctx = zmq.Context()
push = ctx.socket(zmq.PUSH) ; res = ctx.socket(zmq.PULL)
push.bind("ipc://tmpqueue") ; res.bind("ipc://tmpresult")
def sumslice():
    ctx = zmq.Context()
    pull = ctx.socket(zmq.PULL) ; push = ctx.socket(zmq.PUSH)
    pull.connect("ipc://tmpqueue") ; push.connect("ipc://tmpresult")
    slice = 1
    while slice:
        s = cPickle.loads(pull.recv())
        push.send(cPickle.dumps(mdata[s].sum()))
def collect():
    print sum(cPickle.loads(res.recv()) for i in xrange(0, len(mdata), 1<<10))
processes = [ multiprocessing.Process(target=sumslice)
              for _ in xrange(multiprocessing.cpu_count()) ]
for p in processes: p.start()
collector = threading.Thread(target=collect)
collector.start()
for i in xrange(0, len(mdata), 1<<10): push.send(cPickle.dumps(slice(i,i+1<<10)))
collector.join()
```

ZMQ
mensajes indirectos

- Mreq / Mrep – pipelining
  - Juntar varios requests en un megamensaje
  - Permite explotar la naturaleza async de ZMQ
    - Mientras el worker trabaja, acumulo requests
    - Cuando el worker está listo, los mando todos juntos
      - Y recibo todas las respuestas juntas
  - Amortiza los costos de latencia
    - Pero incrementa la latencia individual
    - Sólo adecuado si quiero **thoughput**

- Multiprocessing permite total paralelismo
  - Pero dificulta la comunicación
- Multithreading permite total comunicación
  - Pero GIL bloquea el paralelismo
- **Combinando paradigmas, lo mejor de ambos**

tarea para el hogar

Híbridos

Conclusión

- Ningún método es la panacea universal
- Cada problema tiene su truco
- Cada truco su punto fuerte
  - Y su punto débil
- *“embrace extensions”*
  - *Cython*
  - *ZMQ*
- El GIL no es el malo de la película
  - Recordar que existe por una razón