PEP-3156
Async I/O en Python

Saúl Ibarra Corretgé
@saghul

PyConES 2013

repr(self)

```
>>> from Amsterdam import saghul
>>> saghul.work()
VoIP, SIP, XMPP, chat, Real Time Communications
>>> saghul.other()
networking, event loops, sockets, MOAR PYTHON
>>> saghul.languages()
Python, C
>>> saghul.message()
Estoy encantado de participar en PyConES!
>>>
```

`import open_source`

- Core Team de libuv
- Tulip / asyncio
- Muchos experimentos con event loops y más cosas
  - github.com/saghul

import socket

```
import socket

server = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
server.bind(('127.0.0.1', 1234))
server.listen(128)
print("Server listening on: {}".format(server.getsockname()))

client, addr = server.accept()
print("Client connected: {}".format(addr))

while True:
    data = client.recv(4096)
    if not data:
        print("Client has disconnected")
        break
    client.send(data)

server.close()
```

except Exception

- Solo podemos gestionar una conexión
- Soluciones para soportar múltiples clientes
  - Threads
  - Procesos
  - Multiplexores de I/O

import thread

```
import socket
import thread

def handle_client(client, addr):
    print("Client connected: {}".format(addr))
    while True:
        data = client.recv(4096)
        if not data:
            print("Client has disconnected")
            break
        client.send(data)

server = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
server.bind(('127.0.0.1', 1234))
server.listen(128)
print("Server listening on: {}".format(server.getsockname()))

while True:
    client, addr = server.accept()
    thread.start_new_thread(handle_client, (client, addr))
```

except Exception

- Los threads añaden *overhead*
  - Tamaño del *stack*
  - Cambios de contexto
- Sincronización entre threads
- El GIL - NO es un problema

Multiplexores de I/O

- Examinar y bloquearse a la espera de que un *fd* esté listo
- ¡Aun así la operación puede bloquear!
- El *fd* tiene que ser puesto en modo no bloqueante
- ¿Y Windows?


KEEP
CALM
I'LL
EXPLAIN
LATER

Multiplexores de I/O

1. Poner un *fd* en modo no bloqueante
2. Añadir el *fd* al multiplexor
3. Esperar un rato
4. Realizar la operación bloqueante sobre el *fd* si está listo

def set_nonblocking

```
import fcntl

def set_nonblocking(fdobj):
    # unix only
    try:
        setblocking = fdobj.setblocking
    except AttributeError:
        try:
            fd = fdobj.fileno()
        except AttributeError:
            fd = fdobj
        flags = fcntl.fcntl(fd, fcntl.F_GETFL)
        fcntl.fcntl(fd, fcntl.F_SETFL, flags | os.O_NONBLOCK)
    else:
        setblocking(False)
```

`import select`

- El módulo “select” implementa los distintos multiplexores de i/o
  - select
  - poll
  - **kqueue**
  - **epoll**
- En Python 3.4, módulo “selectors”

`sys.platform == 'win32'`

- Soporta select()!
  - 64 *fds* por thread
- >= Vista soporta WSAPoll!
  - Está roto: bit.ly/Rg06jX
- IOCP es lo que mola

`import IOCP`

- Empezar la operación de i/o en modo *overlapped*
- Recibe un callback cuando ha terminado
- *Completion Ports*
- Abstracción totalmente distinta a Unix

windows.rules = True


I/O IS HARD
LET'S GO SHOPPING
memegenerator.net

Frameworks

- Los frameworks nos abstraen de las diferentes plataformas
- Protocolos
- Integracion con otros event loops: Qt, GLib, ...
- Distintas APIs

import twisted

- Usa select, poll, kqueue, epoll del módulo select
- IOCP en Windows
- Integración con otros loops como Qt
- Protocolos, transportes, ...
- Deferred

`import tornado`

- Usa select, poll, kqueue, epoll del modulo select
- select() en Windows :-(
- Principalmente orientado a desarrollo web
- API síncrona con coroutines

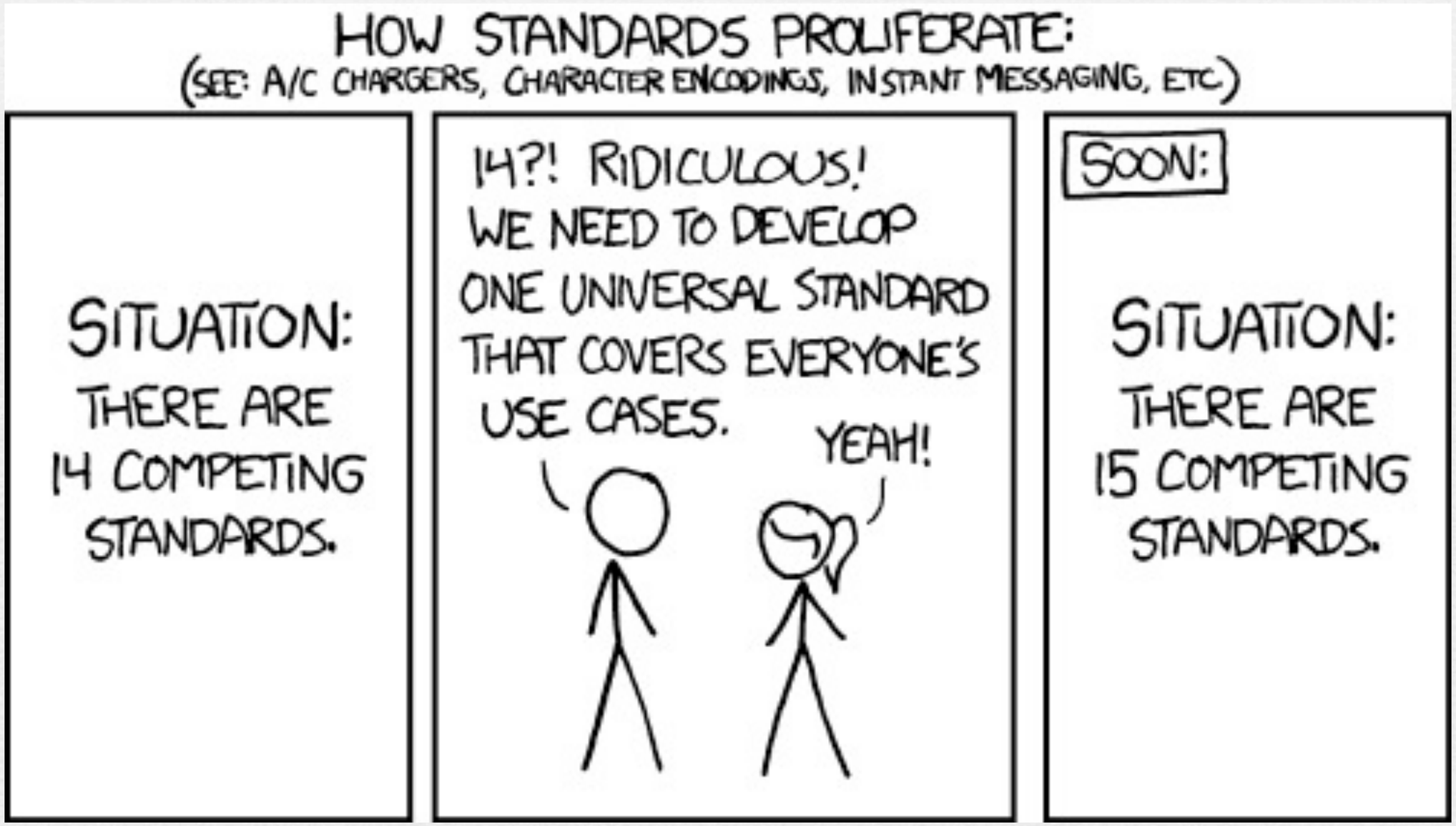
`import gevent`

- Usa libevent en la 0.x y libev en la 1.x
- select() en Windows :-(
- API síncrona con greenlet

import asyncore

- raise RuntimeError("NOT GOOD ENOUGH")
- "asyncore: included batteries don't fit" bit.ly/182HcHT

Solución


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION:
THERE ARE
14 COMPETING
STANDARDS.
14?! RIDICULOUS!
WE NEED TO DEVELOP
ONE UNIVERSAL STANDARD
THAT COVERS EVERYONE'S
USE CASES.
YEAH!
SOON:
SITUATION:
THERE ARE
15 COMPETING
STANDARDS.

I'm not trying to reinvent the wheel. I'm trying to build a good one.

Guido van Rossum

asyncio

import tulip

`import asyncio`

- Implementación de referencia del PEP-3156
- Componentes base para i/o asíncrona
- Funciona en Python >= 3.3

Objetivos

- Implementación moderna de i/o asíncrona para Python
- Utilizar yield from (PEP-380)
  - Pero no obligar a nadie
- No utilizar nada que requiera Python > 3.3
- Interoprabilidad con otros frameworks

Objetivos

- Soporte para Unix y Windows
- IPv4 e IPv6
- TCP, UDP y pipes
- SSL básico (seguro por defecto)
- Subprocesos

No Objetivos

- Perfección
- Reemplazar los frameworks actuales
- Implementaciones de protocolos concretos
- Reemplazar httplib, smtplib, ...
- Funcionar en Python < 3.3

¿Interoperabilidad?

twisted

tornado

gevent

...

asyncio

selectors

iocp

¿Interoperabilidad?

twisted

tornado

gevent

...

asyncio

rose / pyuv

Arquitectura

Componentes

Event loop, policy

Coroutines, Futures, Tasks

Transports, Protocols

Event loop

Event loop y policy

- Selecciona el mejor selector para cada plataforma
- APIs para crear servidores y conexiones (TCP, UDP, ...)

Callbacks

- loop.call_soon(func, *args)
- loop.call_later(delay, func, *args)
- loop.call_at(when, func, *args)
- loop.time()

Callbacks para I/O

- loop.add_reader(fd, func, *args)
- loop.add_writer(fd, func, *args)
- loop.remove_reader(fd)
- loop.remove_writer(fd)

Señales Unix

- loop.add_signal_handler(sig, func, *args)
- loop.remove_signal_handler(sig)

Interfaz con Threads

- loop.call_soon_threadsafe(func, *args)
- loop.run_in_executor(exc, func, *args)
- loop.set_default_executor(exc)

Inicio y parada

- loop.run_forever()
- loop.stop()
- loop.run_until_complete(f)

Acceso a la instancia

- get_event_loop()
- set_event_loop(loop)
- new_event_loop()

Policy (default)

- Un event loop por thread
- Solo se crea un loop automágicamente para el main thread

Policy

- Configura qué hacen get/set/new _event_loop
- Es el único objeto global
- ¡Se puede cambiar! (ejemplo: rose)

Coroutines, Futures y Tasks

Coroutines, Futures y Tasks

- Coroutines
  - una función generator, puede recibir valores
  - decorada con @coroutine
- Future
  - promesa de resultado o error
- Task
  - Future que ejecuta una coroutine

Coroutines y yield from

```
import asyncio
import socket

loop = asyncio.get_event_loop()

@asyncio.coroutine
def handle_client(client, addr):
    print("Client connected: {}".format(addr))
    while True:
        data = yield from loop.sock_recv(client, 4096)
        if not data:
            print("Client has disconnected")
            break
        client.send(data)

@asyncio.coroutine
def accept_connections(server_socket):
    while True:
        client, addr = yield from loop.sock_accept(server_socket)
        asyncio.async(handle_client(client, addr))

server = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
server.bind(('127.0.0.1', 1234))
server.listen(128)
server.setblocking(False)
print("Server listening on: {}".format(server.getsockname()))

loop.run_until_complete(accept_connections(server))
```

Coroutines y yield from

- Imagina que el yield from no existe
- Imagina que el código es secuencial
- No te bases en la definición formal de yield from (PEP-380)

Futures

- Parecidos a los Futures de PEP-3148
  - concurrent.futures.Future
- API (casi) idéntico:
  - f.set_result(); r = f.result()
  - f.set_exception(e); e = f.exception()
  - f.done()
  - f.add_done_callback(x); f.remove_done_callback(x)
  - f.cancel(); f.cancelled()

Futures + Coroutines

- yield from funciona con los Futures!
  - f = Future()
    - Alguien pondrá el resultado o la excepción luego
  - r = yield from f
    - Espera a que esté listo y devuelve f.result()
- Normalmente devueltos por funciones

Deshaciendo callbacks

```
@asyncio.coroutine
def sync_looking_function(*args):
    fut = asyncio.Future()
    def cb(result, error):
        if error is not None:
            fut.set_result(result)
        else:
            fut.set_exception(Exception(error))
    async_function(cb, *args)
    return (yield from fut)
```

Tasks

- Unicornios con polvo de hada
- Es una coroutine metida en un Future
- WAT
- Hereda de Future
- Funciona con yield from
  - r = yield from Task(coro(...))

Tasks vs coroutines

- Una coroutine no “avanza” sin un mechanismo de scheduling
- Los Tasks pueden avanzar solos, sin necesidad de esperar
- El event loop es el scheduler!
  - Magia!

Otro ejemplo

```
import asyncio

loop = asyncio.get_event_loop()
clients = {} # task -> (reader, writer)

def _accept_client(client_reader, client_writer):
    task = asyncio.Task(_handle_client(client_reader, client_writer))
    clients[task] = (client_reader, client_writer)

    def client_done(task):
        del clients[task]

    task.add_done_callback(client_done)

@asyncio.coroutine
def _handle_client(client_reader, client_writer):
    while True:
        data = (yield from client_reader.readline())
        client_writer.write(data)


f = asyncio.start_server(_accept_client, '127.0.0.1', 12345)
server = loop.run_until_complete(f)
loop.run_forever()
```

Transports y Protocols

Transports y Protocols

- Transport: representa una conexión (socket, pipe, ...)
- Protocol: representa una aplicación (servidor HTTP, cliente IRC, ...)
- Siempre van juntos
- API basada en llamadas a función y callbacks

Clientes y servidores

- loop.create_connection(...)
  - crea un Transport y un Protocol
- loop.create_server(...)
  - crea un Transport y un Protocol por cada conexión aceptada
  - devuelve un objeto Server

Clientes y servidores

- loop.open_connection(...)
  - wrapper sobre create_connection, devuelve (stream_reader, stream_writer)
- loop.start_server(...)
  - wrapper sobre create_server, ejecuta un callback con (stream_reader, stream_writer) por cada conexión aceptada

Interfaz Transport -> Protocol

- connection_made(transport)
- data_received(data)
- eof_received()
- connection_lost(exc)

Interfaz Protocol -> Transport

- write(data)
- writelines(seq)
- write_eof()
- close()
- abort()

MOAR PROTOCOL!

- ¡Esta tarde a las 17:30!
- “**Tulip or not Tulip**” - Iñaki Galarza

¿Y ahora qué?

- Ven a la charla de Iñaki
- Lee el PEP-3156
- Implementa algún protocolo sencillo (cliente IRC por ejemplo)
- ¡Usa asyncio en tu próximo proyecto!

¿Preguntas?

@saghul

bettercallsaghul.com

Referencias

- code.google.com/p/tulip/
- groups.google.com/forum/#!forum/python-tulip
- PEP-3156
- http://www.youtube.com/watch?v=1coLC-MUCJc
- https://www.dropbox.com/s/essjj4qmmtrhys4/SFMeetup2013.pdf