gRPC Python, C Extensions, and AsyncIO

Discord channel: #talk-grpc-and-asyncio

About us

- Lidi Zheng
  - Software Engineer at Google
  - Maintainer of gRPC Python
- Pau Freixes
  - Former Senior Software Engineer at Skyscanner
  - Currently at Onna.com (we are hiring!)
  - Python enthusiast, but definitely what likes most is solve problems.
  - Open source contributor: Aiohttp, emcache, etc

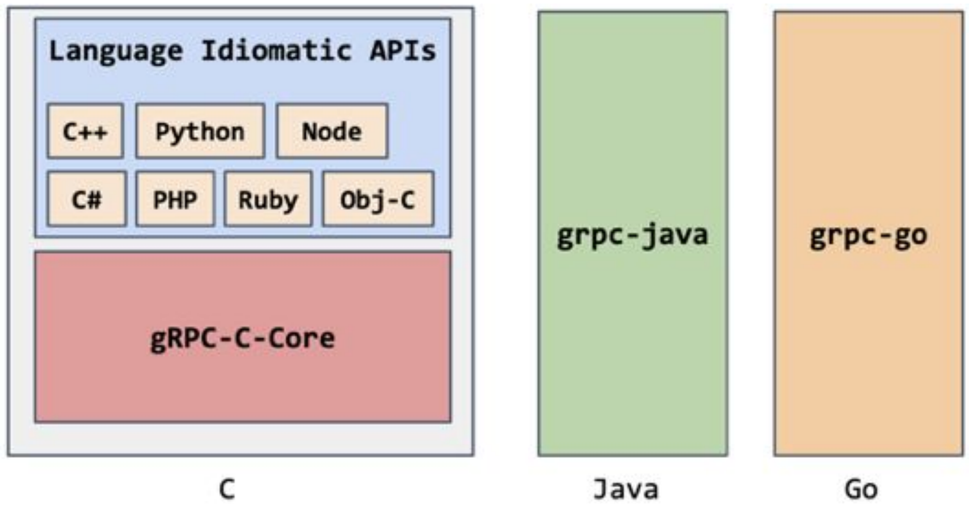
What is gRPC?

- RPC framework upon HTTP/2
- Fast, light-weight and feature rich:
  - Bi-directional streaming RPC
  - Client-side/Look-aside load balancing
  - Interceptors
  - ProtoBuf
  - ...
- ~400k downloads / day (grpcio)

GitHub ⭐: 26.8k
Contributors: 572

Core and Python

- Python is a wrapper over Core
- 14 supported languages
- Benefits:
  - Better performance
  - Lower maintenance burden
- Frictions:
  - Segfaults
  - Memory leaks
  - Compilation



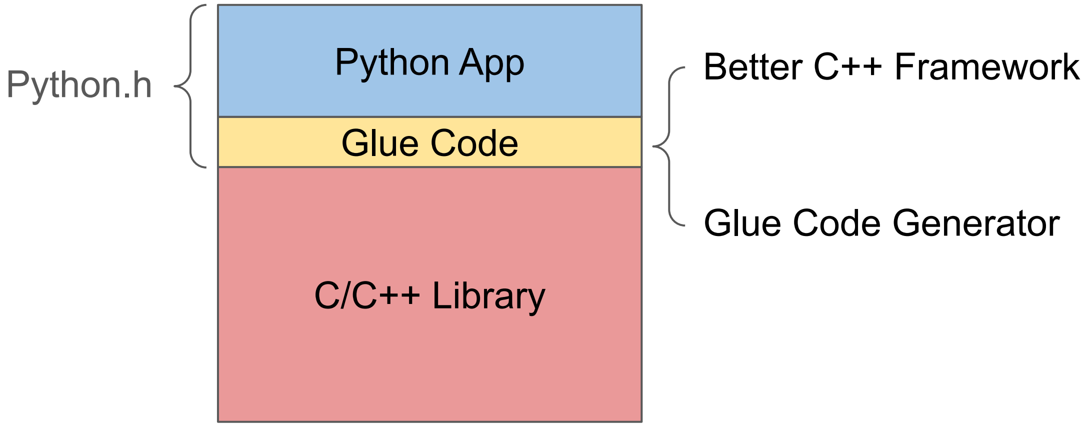
What’s Python C Extension?

- Module written in C/C++
- Python.h
- Complex to write:
  - Version compatibility
  - Lot’s of boilerplate
- Why?
  - Integration
  - Performance

```c
#include <Python.h>

static PyObject* hello_world(PyObject* self, PyObject* args) {
    printf("Hello World\n");
    return Py_None;
}

static PyMethodDef methods[] = {
    { "hello_world", hello_world, METH_NOARGS, "Prints hello world."},
    { NULL, NULL, 0, NULL }
};

static struct PyModuleDef hello_world_module = {
    PyModuleDef_HEAD_INIT,
    "hello_world_module",
    "Test Module",
    -1,
    methods
};

PyMODINIT_FUNC PyInit_hello_world_module(void) {
    return PyModule_Create(&hello_world_module);
}
```

What’s Python C Extension?

- Module written in C/C++
- Python.h
- Complex to write:
  - Version compatibility
  - Lot’s of boilerplate
- Why?
  - Integration
  - Performance



Popular Gluing Approaches

Approach	**Pros**	**Cons**
Pyclif	Straightforward template syntax	Needs to learn the templating language; more glue logic in C++
Pybind11	Portable, lightweight, header-only.	Requires to code in C++ (might be a plus for C++ fans)
Cython	Ease to develop (adopted by NumPy and SciPy).	Language itself is a **“superset” of Python**

Other options: Ctypes, CFFI, SWIG, Boost.python

Cython in a Nutshell

```python
import math

def am_i_prime(x: int) -> bool:
    for i in range(2, math.floor(math.sqrt(x))):
        if x % i == 0:
            return False
    return True
```

prime_checker.py

import →

```python
import prime_checker

print(prime_checker.am_i_prime(2**31-1))
```

```cython
from libc.math cimport sqrt, floor

cdef am_i_prime(int x):
    cdef double root = sqrt(<double>x)
    for i in range(2, <int>floor(root)):
        if x % i == 0:
            return False
    return True
```

prime_checker.pyx

compile → 4000+ lines C file → compile → prime_checker.so → import

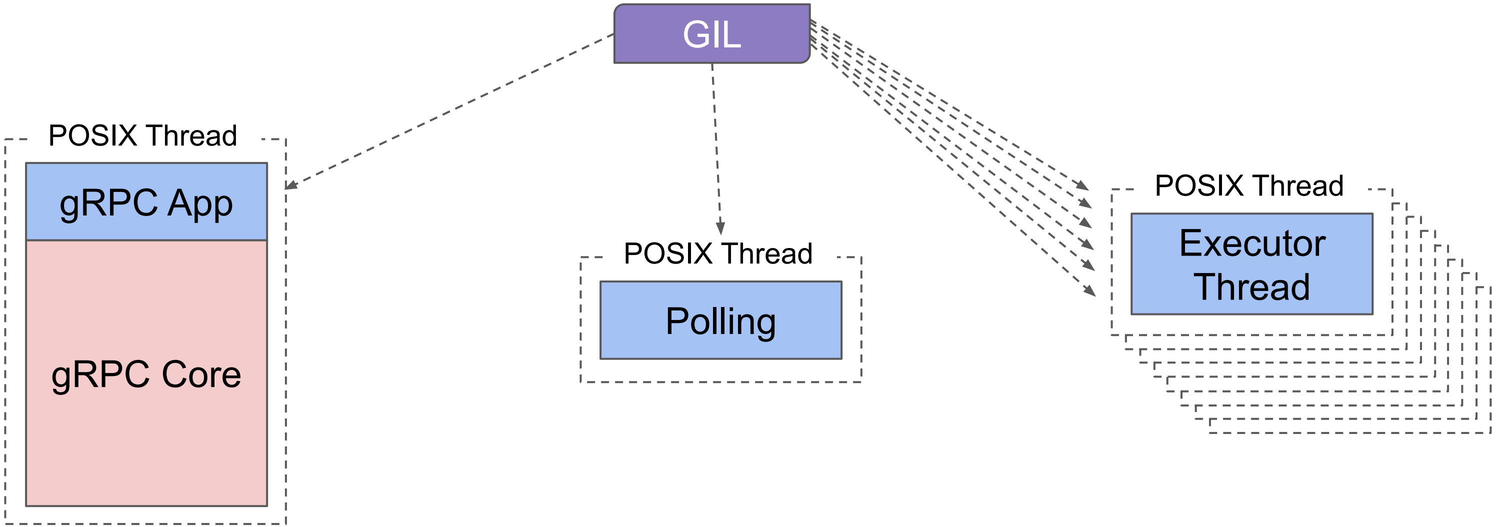
[Read More] https://cython.readthedocs.io/en/latest/src/tutorial/cython_tutorial.html

Python & gdb

```
lidi@dev:grpc$ gdb python3.7
(gdb) source /users/lidi/src/Python-3.7.0/python-gdb.py
(gdb) run _channel_ready_future_test.py
...
^C
Thread 1 "python" received signal SIGINT, Interrupt.
(gdb) py-bt
Traceback (most recent call first):
  File "/usr/local/lib/python3.7/threading.py", line 300, in wait
    gotit = waiter.acquire(True, timeout)
...
  File "src/python/grpcio_tests/tests/unit/_channel_ready_future_test.py", line 97, in <module>
    unittest.main(verbosity=2)
(gdb) py-list
 299    if timeout > 0:
>300        gotit = waiter.acquire(True, timeout)
 301    else:
(gdb) print __pyx_v_self
$1 = <grpc._cython.cygrpc.CompletionQueue at remote 0x7ffff360fd50>
(gdb) bt
#22 0x000055555568416a in PyEval_EvalFrameEx (throwflag=0,
    f=Frame 0x555555fa5888, for file /usr/local/lib/python3.7/unittest/case.py, line 615...
```

[Read More] https://wiki.python.org/moin/DebuggingWithGdb

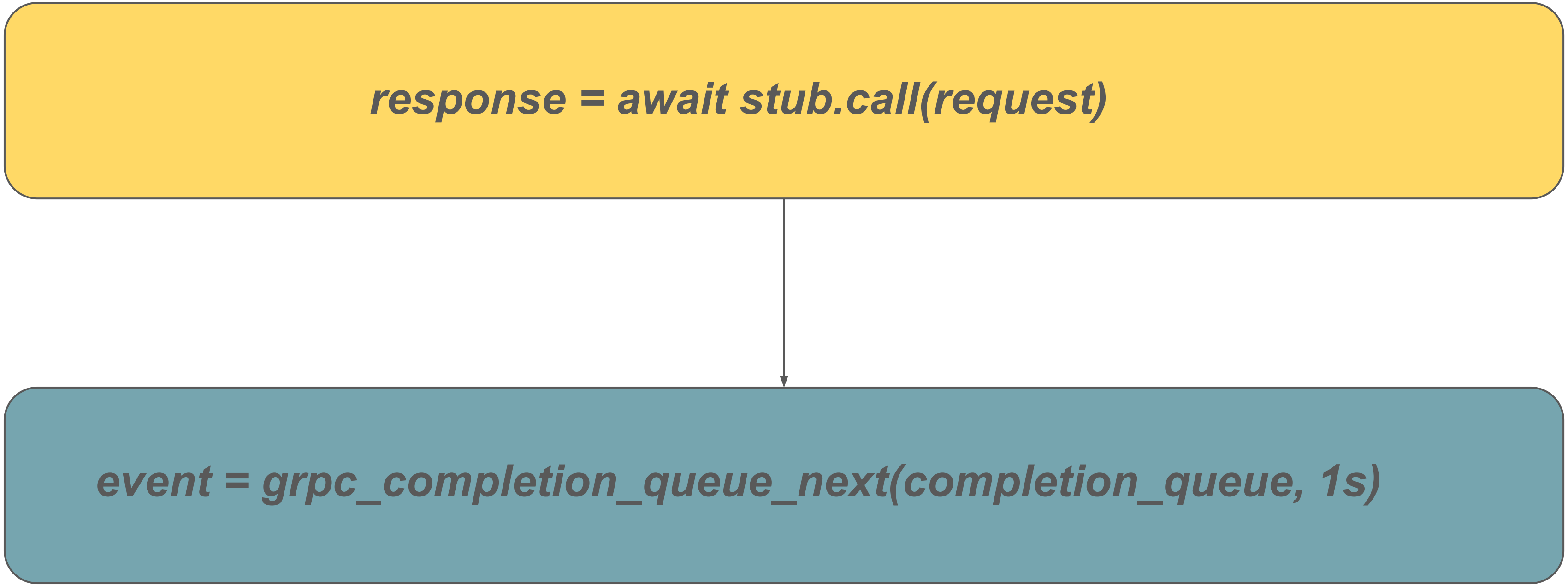
Non-AsyncIO Threading Model



[Read More] https://wiki.python.org/moin/GlobalInterpreterLock

gRPC and Asyncio

Not blocking the loop, what a headache



Not blocking the loop, what a headache

- gRPC C++ interface provided a way of installing custom IO managers
  - read, write, etc ...
- But the interface for **polling gRPC events was still blocking**
  - For Asyncio this was a no go.
- Other frameworks had a similar problem but managed to solve the issue
  - Gevent, by just providing its custom IO manager
  - Node.js, implicit cooperation by using same libuv loop instance behind the scenes

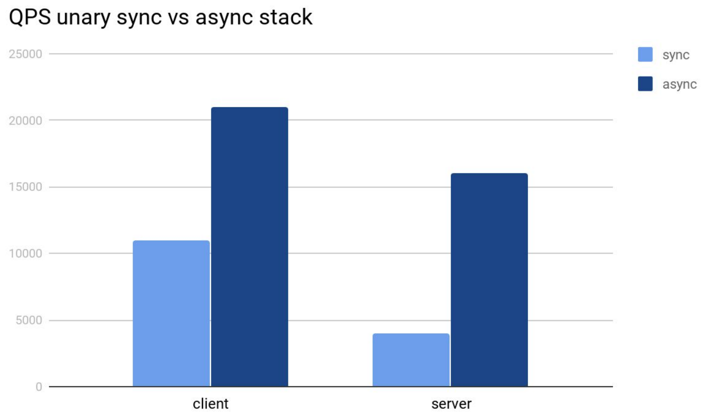
Not blocking the loop, what a headache

- … and gRPC C++ introduced a new completion queue based on callbacks
  - Was orginally developed for having fully asynchronous C++ implementations
- Instead of making blocking calls a callback would tell you when a gRPC event would be avaialable.
  - This allowed us to return the control to the loop for Asyncio.
- Eureka!!!

Solution 1, our own IO manager implementation

Our first implementation looked promising, based on

- implementing our own **custom IO manager**
- using the **callback completion queue**



Making sync stack compatible with async

Sync and Async compatibility

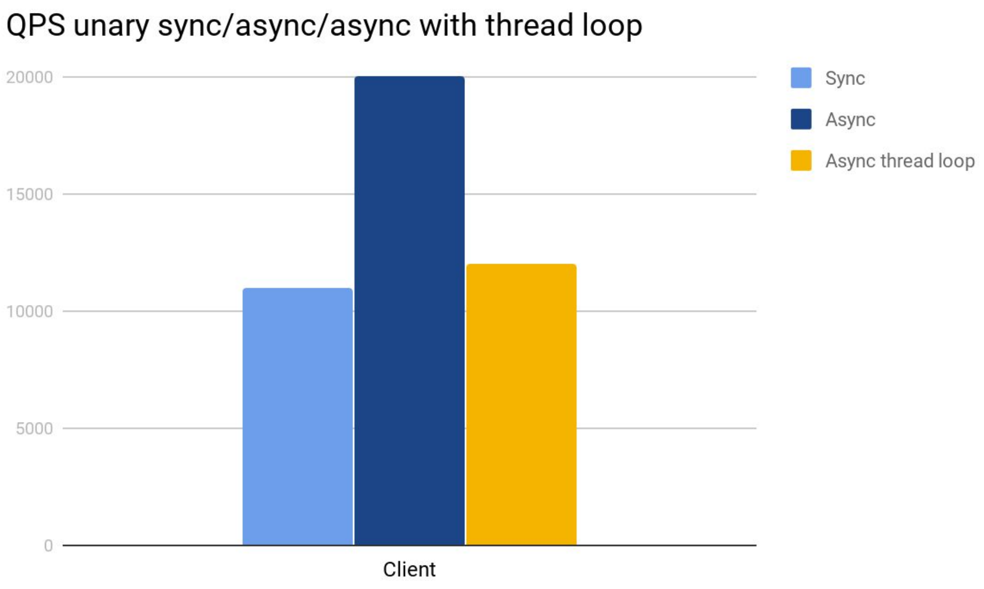
- Synchronous stack was still there, and **it will be there for a long time**
- Sync and Async coexistence was a must
  - An async server might use a library which behind the scenes might use the synchronous version of gRPC
- How the hell this could be addressed?

Sync and Async compatibility

- Rewriting the whole sync stack on top of the async one
  - Could end up blocking the loop in anyway
  - Forced to us to rewrite a large amount of code
- Modifying the gRPC C++ implementation for allowing to have multiple IO managers running at the same time.
  - Implied many changes in the core of the gRPC which could affect other languages
- Run all gRPC IO events in a separated Asyncio thread
  - Allowed to us block the current loop (main thread)
  - The amount of changes needed was affordable
  - Doubts about how performance might be affected

Sync and Async compatibility

It worked but had a very **negative impact in the performance**



Solution 2, poller thread

Solution 2, poller thread implementation

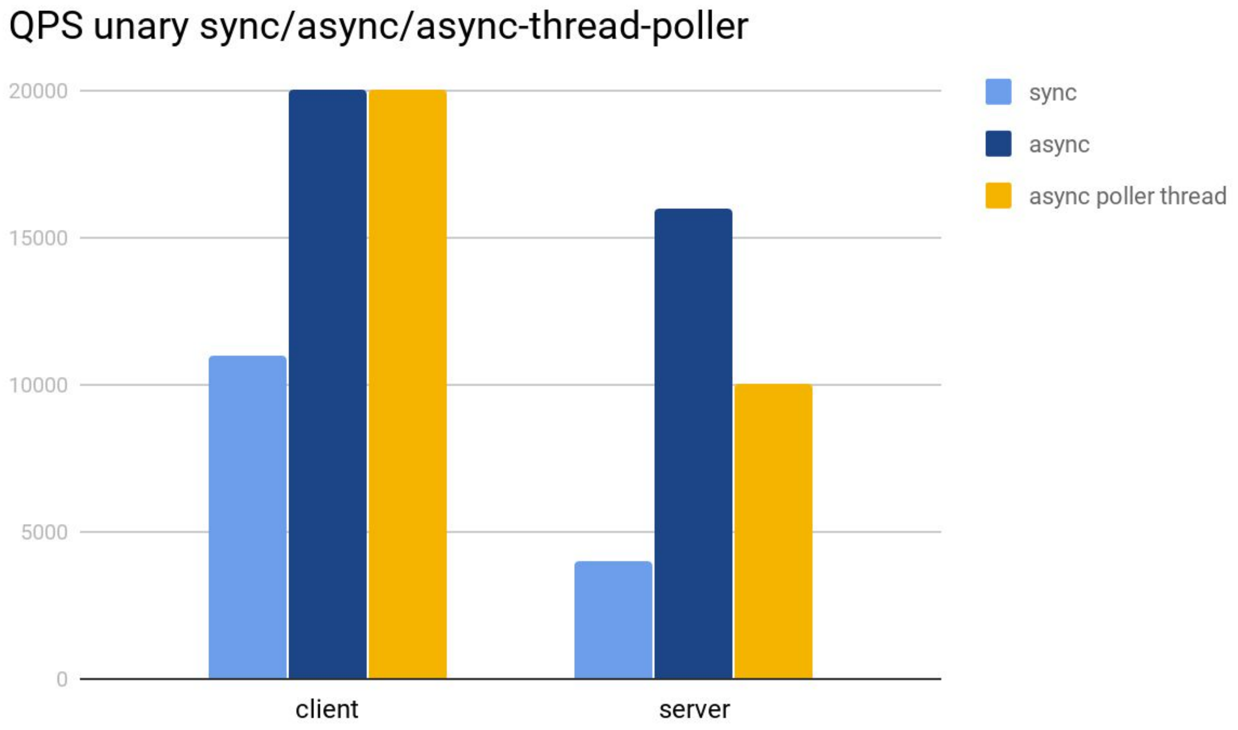
- Discard the usage of the callback completion queue
- Discard the usage of an ad-hoc IO manager
- gRPC Asyncio Python application would start a **separated thread for polling gRPC events**
- This thread won't use any Python object, during the polling
  - Avoid GIL contention
- Events would be added into a C++ queue
- Asyncio loop will be woken up by writing into a socket
  - Again not using any Python objects at all

Solution 2, poller thread implementation

The solution had really good benefits

- Remove the burden of having to maintain a new IO manager
- Any little detail implemented by the C++ gRPC IO manager will be there
    - Unix sockets
    - etc.
- Performance degradation affordable, still a nice boost compared to the synchronous stack.
- Eureka!

Solution 2, poller thread implementation



Thanks!!! QA