The Many Faces of Concurrency in Python

Paradigms and tools for building high-performing systems

Sagiv Malihi

smalihi@cisco.com

@sagiv

Who Am I?

- Sagiv Malihi
- SW Architect – SON Group
- @sagiv
- smalihi at cisco

What are We Building?

Self Optimizing (cellular) Networks

- Connect to all antennas to constantly make adjustments
- Read & analyze tons of statistics
- Synchronize several physical locations

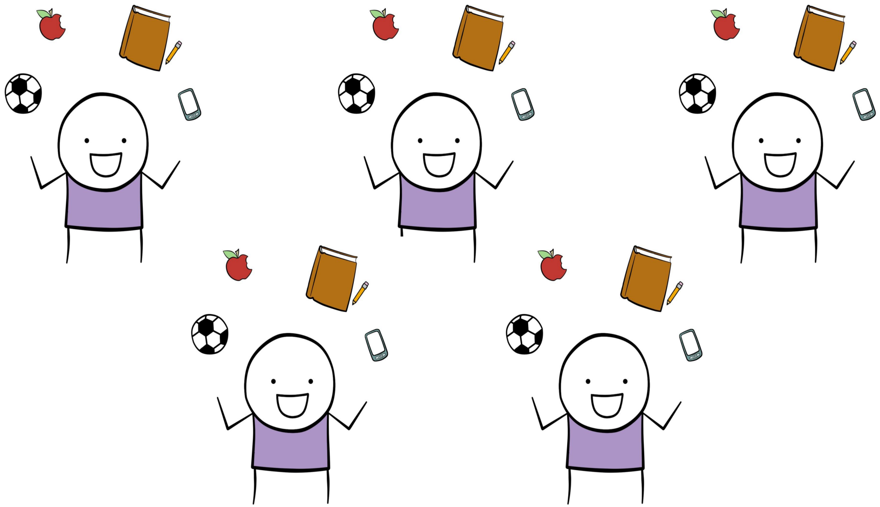
the system had to grow

- Pelephone as 1st customer
- followed by AT&T (!)
- now already in tens of operators

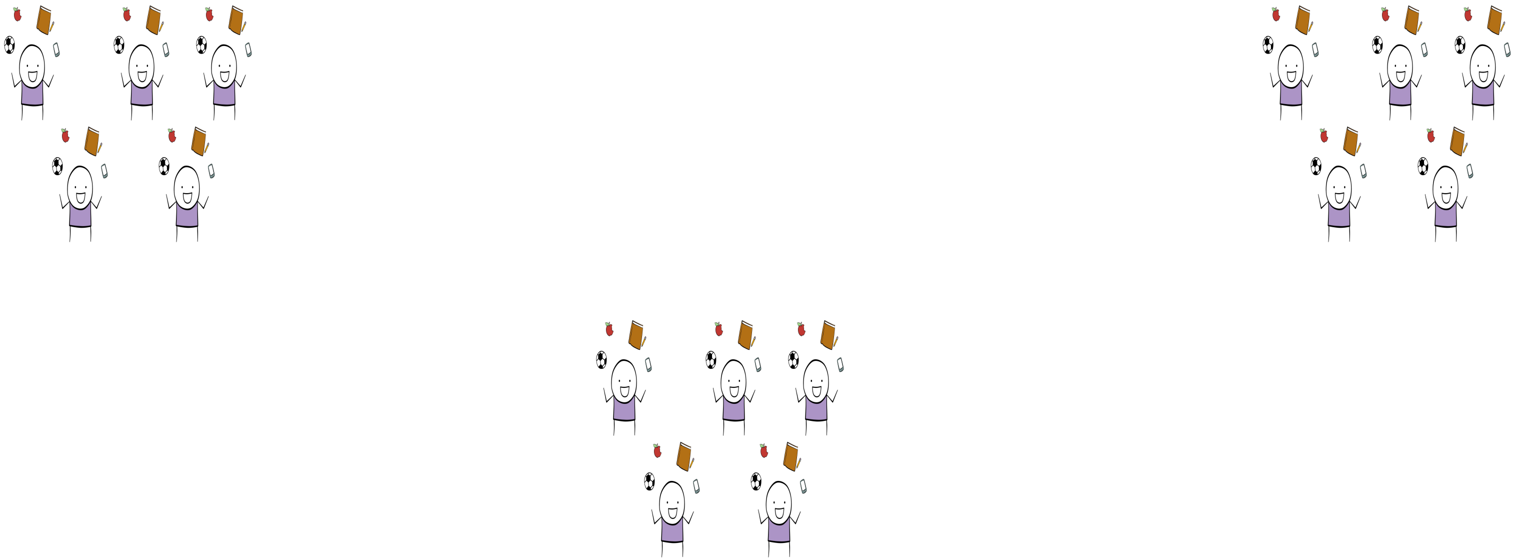
We Needed to Scale!

Concurrency
vs.
Parallelism
vs.
Distributed System

Concurrency – running multiple tasks in overlapping time periods



Parallelism – when multiple tasks actually take place at the same time (e.g. on separate cores)



Distributed Systems – execute tasks in parallel over several machines (in different locations)

Concurrency


Connecting to thousands of cell towers
Continuously tweaking tilt, coverage, handovers…

threading is not a good choice

- shared mem + switching = races
- the GIL prevents true parallelism
- threads are resource-intensive

threading does have an upside

- using threads is easy
- IO is still concurrent
- c extensions can release the GIL
- IronPython / Jython are GIL-less

coroutines to the rescue!

- Predictable
- Lightweight
- Many libraries (incl. asyncio in stdlib)

the basic idea is simple

```
def task1():
    s = socket(...)
    while True:
        yield socket
        print socket.read()

def task2():
    i = 1
    while True:
        yield Sleep(1)
        print i
        i += 1
```

```
def eventloop(*tasks):
    tasks = {task.next(): task
                 for task in tasks}
    while True:
        sockets, sleeps =
           filter_tasks(tasks)
        ready = select(sockets,
                       min(sleeps))
        tasks = call_task_next(tasks,
                               ready)

eventloop(task1(), task2())
```

python 3.5 async/await api

```
import asyncio
async def slow_func():
    await asyncio.sleep(1)
    return “answer”
async def failed_func():
    await asyncio.sleep(1)
    raise Exception(...)
```

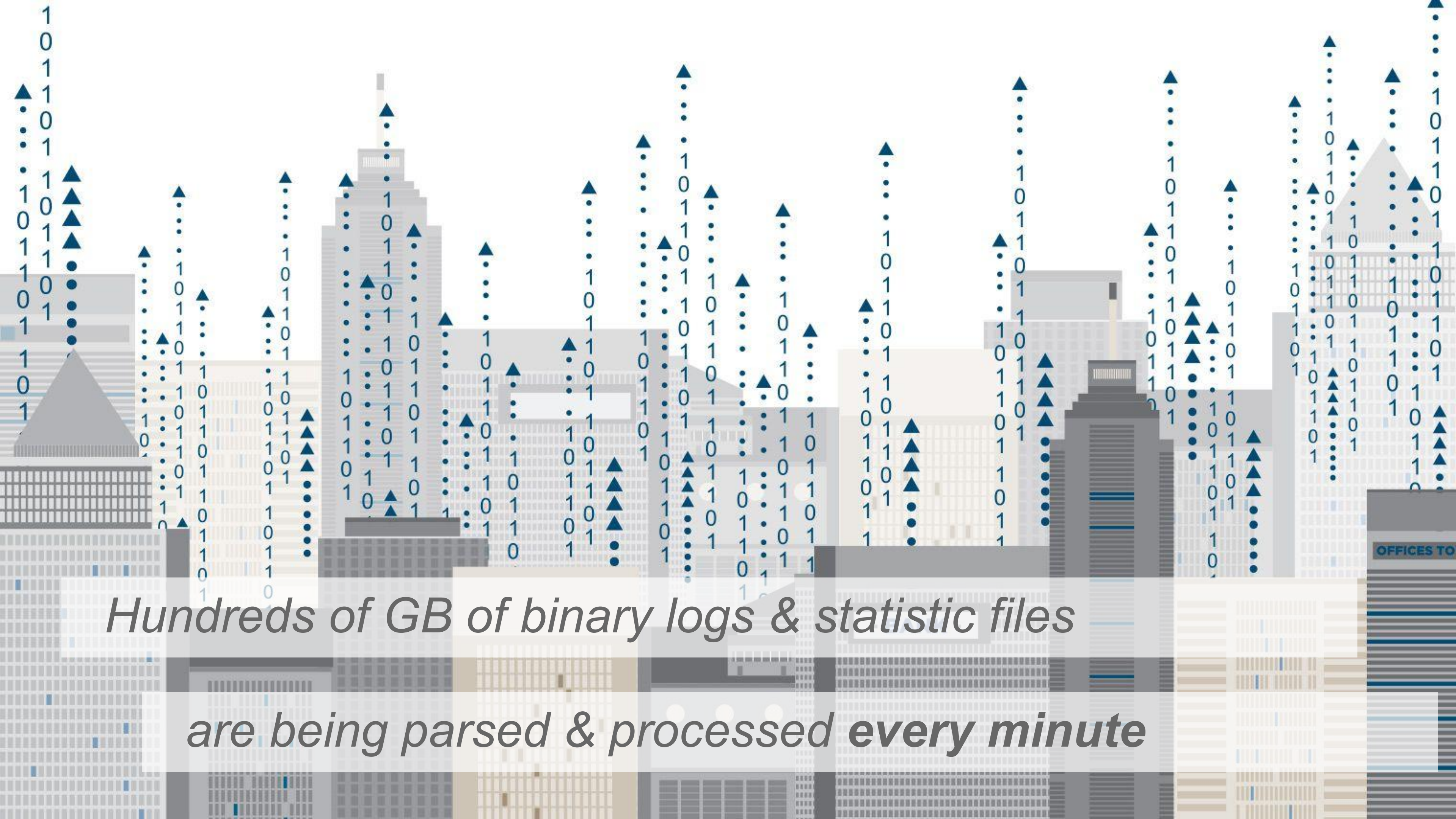
```
async def test():
    response = slow_func()
    try:
        await failed_func()
    except Exception as e:
        print(e, await response)

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

gevent’s magic is a good tradeoff

```
from gevent import monkey
monkey.patch_all()
```

Parallelism



Hundreds of GB of binary logs & statistic files

are being parsed & processed ***every minute***

multiprocessing is like magic!

```
from multiprocessing import
 Process, Pipe

def f(conn):
  conn.send(“hello world”)
  conn.close()
```

```
parent_conn, child_conn = Pipe()

p = Process(target=f,
            args=(child_conn,))

p.start()
print(parent_conn.recv())
```

magic is not always a good thing

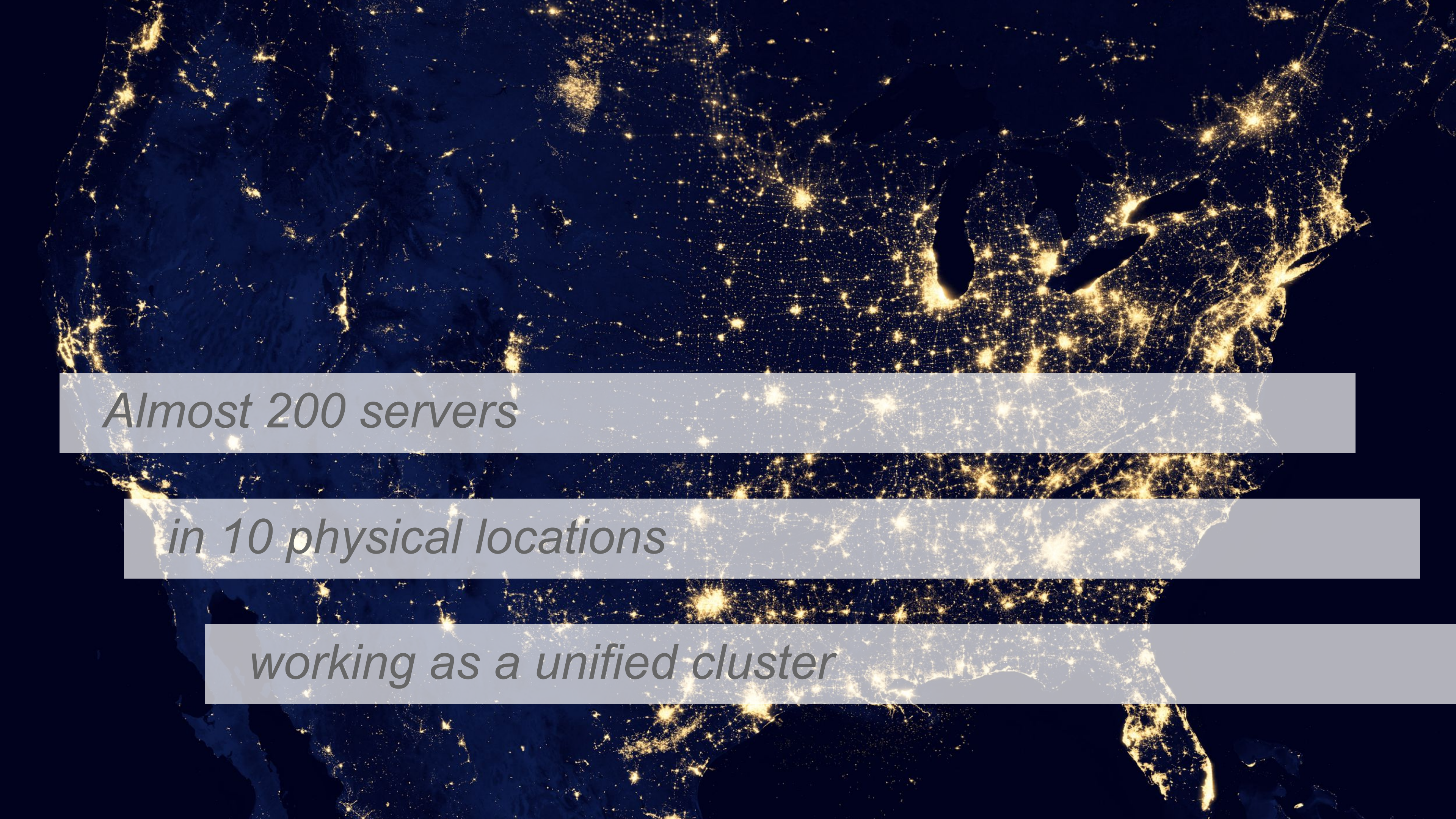
- multiprocessing fork()'s
- does not play well with gevent
- or threads
- or large datasets in memory
- but fixed in python 3.4 !

using subprocess is easier

```
from slaveprocess // uses subprocess + RPyC
   import run_in_process

def f():
  return ("hello world")
print (run_in_process(f))
```

Distributed Systems


Almost 200 servers
in 10 physical locations
working as a unified cluster

distributed DB can really help

- keeps a single-point-of-truth
- on all servers
- can act as a communications channel
- we used mongodb

try to avoid locking

- locks generally lead to deadlocks
- optimistic transaction model
- nodes change the ‘network image’
- verifying consistency before ‘commit’

Summary

- IO bound apps -
  - avoid threads, consider gevent
  - or asyncio
- CPU bound apps -
  - subprocess + RPyC
- distributed apps -
  - let a DB do the hard work

Thank You!

Questions?

TOMORROW starts here.