Pushing Data through Pipelines with Coroutines

Axel Sirota
MACHINE LEARNING ENGINEER
@AxelSirota

Delegate That Task! Yield from Explained

Some Definitions First

Subgenerator -> A generator that does work.

Delegating generator -> Delegates to a subgenerator via a yield from

Caller -> Client code that calls the delegating generator

Yield From Example

```
def delegating_without_yield():
    for c in ['Hello', 'Pluralsighters']:
        yield c


def delegating_with_yield():
    yield from ['Hello', 'Pluralsighters']


In [2]: for i in module5.delegating_without_yield():
   ...:     print(f'-->{i}')
   ...:
-->Hello
-->Pluralsighters

In [3]: for i in module5.delegating_with_yield():
   ...:     print(f'-->{i}')
   ...:
   ...:
-->Hello
-->Pluralsighters
```

◀ **# Yield from an iterable**

◀ **# They are equal!**

Yield From Example

```
def chain(*iterables):
    for it in iterables:
        yield from it
```

◀ **# Yield from an iterable**

```
In [5]: list(module5.chain('ABC', [1,2,3]))
Out[5]: ['A', 'B', 'C', 1, 2, 3]
```

◀ **# It's like it yield from iterates the iterable!**

Yield From Example

```
'#Python Cookbook courtesy
def flatten(items, ignore_types=(str,
bytes)):
   for x in items:
       if isinstance(x, Iterable) and not
isinstance(x, ignore_types):
           yield from flatten(x)
       else:
           yield x
```

- # For every element check if its iterable
- # In which case yield from itself
- # If not, just yield it

```
In [6]: list(module5.flatten(['Pluralsight ','is ',['a ',
['great ','platform ','to ']], 'learn!']))
Out[6]: ['Pluralsight ', 'is ', 'a ', 'great ', 'platform ',
'to ', 'learn!']
```

- # It effectively walks the nested iterables and flattens it!

Flatten Explanation

```
'#Python Cookbook courtesy
def flatten(items, ignore_types=(str,
bytes)):
   for x in items:
       if isinstance(x, Iterable) and not
isinstance(x, ignore_types):
           yield from flatten(x)
       else:
           yield x
```

x = 'Pluralsight'

◀ **# x is a string, so we go to else block**

◀ **# We yield 'Pluralsight'**

Flatten Explanation

```
'#Python Cookbook courtesy
def flatten(items, ignore_types=(str,
bytes)):
   for x in items:
      if isinstance(x, Iterable) and not
isinstance(x, ignore_types):
         yield from flatten(x)
      else:
         yield x
```

x = 'is'

◀ **# x is a string, so we go to else block**

◀ **# We yield 'is'**

Flatten Explanation

```
'#Python Cookbook courtesy
def flatten(items, ignore_types=(str,
bytes)):
   for x in items:
      if isinstance(x, Iterable) and not
isinstance(x, ignore_types):
         yield from flatten(x)
      else:
         yield x
```

x = ['a ',['great ','platform ','to ']]

- **# x is a list, therefore we enter the block**
- **# We will yield from flatten(['a ', ['great ','platform ','to ']])**

So we stop and let flatten(['a ',['great ','platform ','to ']]) yield values.

Flatten Explanation

```
‘#Python Cookbook courtesy
def flatten(items, ignore_types=(str,
bytes)):
   for x in items:
      if isinstance(x, Iterable) and not
isinstance(x, ignore_types):
         yield from flatten(x)
      else:
         yield x
```

x = 'a '

◀ **# x is a string, so we go to else block**

◀ **# We yield ‘a’ from inside, this lets the outer flatten yield ‘a’**

Up to this point the yielded values are: ‘Pluralsight’, ‘is’ and ‘a’

```
‘#Python Cookbook courtesy
def flatten(items, ignore_types=(str, 
bytes)):
   for x in items:
      if isinstance(x, Iterable) and not 
isinstance(x, ignore_types):
         yield from flatten(x)
      else:
         yield x
```

Flatten Explanation

x = ['great ','platform ','to ']

- **# x is a list, therefore we enter the block**
- **# We will yield from flatten(['great ','platform ','to ‘])**

So we stop and let flatten(['great ','platform ','to ‘]) yield values.

Flatten Explanation

```
'#Python Cookbook courtesy
def flatten(items, ignore_types=(str,
bytes)):
   for x in items:
      if isinstance(x, Iterable) and not
isinstance(x, ignore_types):
         yield from flatten(x)
      else:
         yield x
```

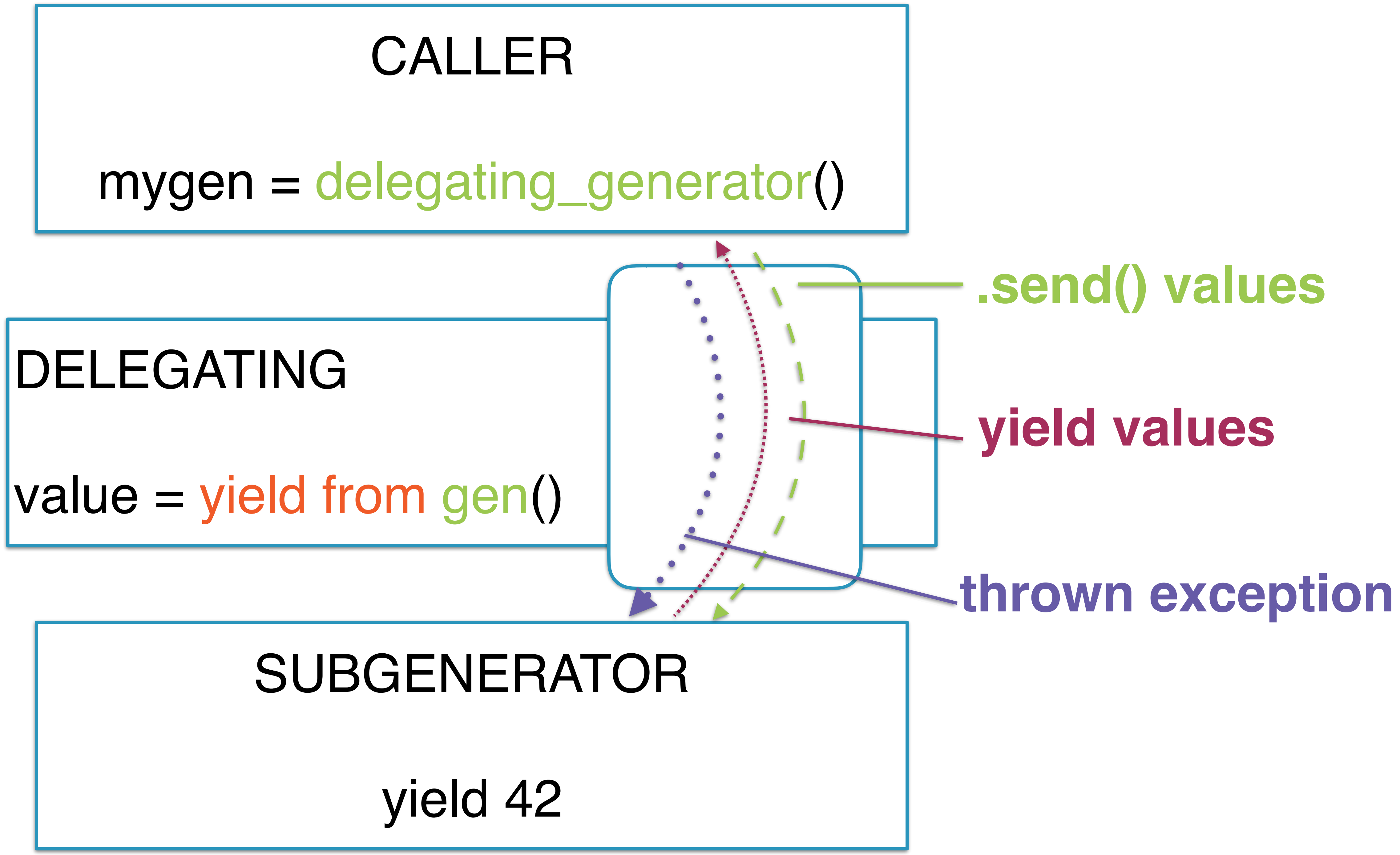
x = 'great '

◀ **# x is a string, so we go to else block**

◀ **# We yield 'great' from inside inside, this lets the inside flatten yield 'great'; making outer flatten yield 'great'**

Up to this point the yielded values are: 'Pluralsight', 'is', 'a' and 'great'

Yield From Explained



```
def caller():
    mygen = delegating_gen()
    print(f'-->{next(mygen)}')
    print(f'-->{next(mygen)}')
    try:
        print(f'--> Take my ValueError subgen!')
        mygen.throw(ValueError)
    except GenkidamaException:
        print(f'-->I got your genkidama! \n Now I will close you')
        mygen.close()

def delegating_gen():
    print(f'**> I am the delegating gen, I dont do much here!')
    try:
        yield from subgen()
    except ValueError:
        print('** I got a Value Error')

def subgen():
    try:
        print(f'I am a subgenerator and I am yielding!')
        yield 1
        yield 2
        yield 3
   except ValueError:
        print('HA! I got you! Get my Genkidama caller!')
        raise GenkidamaException('Take that!')
```

More Yield From!

- # the variable mygen is a reference of subgen()! This will be clear on the prints
- #We throw an exception!
- # And close the subgen in case we get an exception thrown.

- # We catch ValueError by sending a GenkidamaExcewption

More Yield From!

```
In [14]: caller()
**> I am the delegating gen, I dont do
much here!

I am a subgenerator and I am yielding!

-->1
-->2
--> Take my ValueError subgen!

HA! I got you! Get my Genkidama caller!

-->I got your genkidama!

 Now I will close you
```

- **# Check that we got the first print on the next call**
- **# But we need not prime the subgen()**
- **# The yielded values are the ones from subgen!**
- **#We throw an exception!**
- **# It was sent to subgen!**
- **#And the sent exception was sent to caller directly!**

More Yield From!

```
def caller():
    mygen = delegating_gen()
    while True:
        try:
            print(f'Next value: {next(mygen)}')
        except StopIteration as e:
            print(f'Caller print: {e.value}')
            break

def delegating_gen():
    try:
        magic_value = yield from subgen()
    except StopIteration as e:
        print(f'Delegating generator print: {e.value}')
    else:
        print(f'Delegating generator final print:
{magic_value}')
        return 'Goodbye'

def subgen():
    yield 1
    yield 2
    return 'Hello'
```

◀ # Which return will we get?

◀ # Will we get a StopIteration?

◀ # Or it will be handled by yield from?

More Yield From!

```
In [24]: caller()
Next value: 1
Next value: 2


Delegating generator final print:Hello


Caller print: Goodbye
```

- **# We get yielded the values from subgen**
- **# yield from handle internally the StopIteration! It got assigned to magic_value!**
- **# And the caller caught the delegating_generator() return statement**

Yield from enables to delegate work to a subgenerator via a pipe

The caller can yield, send or throw to the reference of the sub generator

The return statement in the yield from is magically handled and set as an expression

Flow That Data with Coroutines

Pipelines With Coroutines

```
@coroutine
def filter_first_list(target):
    while True:
        try:
            list_yielded = yield
            filtered_element = list_yielded[1:]
            target.send(filtered_element)
        except StopIteration as e:
            print('Filter: I am done!')
            return e.value

@coroutine
def double_first_list(target):
    while True:
        try:
            list_yielded = yield
            first_element = list_yielded[0]
            list_yielded= [first_element*2, *list_yielded[1:]]
            target.send(list_yielded)
        except StopIteration as e:
            print('Double: I am done!')
            return e.value

@coroutine
def square_list():
    list_yielded = yield
    new_list = [a**2 for a in list_yielded]
    return new_list
```

◀ **# This is a middle point in our pipeline, it receives a value, transforms it and send it somewhere else**

◀ **# Same here**

◀ **# This is a sink**

```
In [43]: def get_data(coroutine, iterable):
   ...:     try:
   ...:         coroutine.send(iterable)
   ...:     except StopIteration as e:
   ...:      print(e.value)
   ...:

In [44]:
get_data(filter_first_list(double_first_list(square_list())),
[2,3,4,5])
Double: I am done!
Filter: I am done!
[36, 16, 25]

In [45]: get_data(filter_first_list(square_list()), [2,3,4,5])
Filter: I am done!
[9, 16, 25]

In [46]: get_data(double_first_list(square_list()), [2,3,4,5])
Double: I am done!
[16, 9, 16, 25]

In [49]:
get_data(double_first_list(filter_first_list(filter_first_list(square
_list()))), [2,3,4,5])
Filter: I am done!
Double: I am done!
[16, 25]
```

Pipelines With Coroutines

- **# Iteration does not start the pipeline!**
- **# We can combine it how we want it**

Filtering Pipelines

```
@coroutine
def filter_by_pattern(pattern, target):
    print(f'I am going to filter lines by pattern
{pattern}')
    while True:
        line = yield
        if pattern in line:
            target.send(line)


def iterate_file(file, target):
    with open(file) as f:
        for line in f:
            target.send(line)
    print('Im done!')


@coroutine
def coprint():
    while True:
        line = yield
        print(f'-->{line}\n')
```

◀ **# We can manipulate data in the middle of the pipeline!**

◀ **# Check that the driver is not iteration from the coroutine**

◀ **# This is a sink**

Filtering Pipelines

```
In [6]: module5.iterate_file('filter_example.txt',
module5.filter_by_pattern('favicon',
module5.coprint()))

-->140.180.132.213 - - [24/Feb/2008:00:08:59
-0600] "GET /favicon.ico HTTP/1.1" 404 133


-->75.54.118.139 - - [24/Feb/2008:00:15:42 -0600]
"GET /favicon.ico HTTP/1.1" 404 133


-->128.143.38.83 - - [24/Feb/2008:00:31:39
-0600] "GET /favicon.ico HTTP/1.1" 404 133

...
```

◀ **# We only get favicon lines**

Broadcasting Pipelines

```
@coroutine
def broadcast(targets):
    while True:
        item = yield
        for target in targets:
            target.send(item)
```

◀ **# Send to all targets the same element!**

Broadcasting Pipelines

```
In [4]: printer = module5.coprint()

In [5]: my_filter_1 = module5.filter_by_pattern('python', printer)
I am going to filter lines by pattern python

In [6]: my_filter_2 = module5.filter_by_pattern('favicon', printer)
I am going to filter lines by pattern favicon

In [7]: my_filter_3 = module5.filter_by_pattern('images', printer)
I am going to filter lines by pattern images

In [9]: broadcaster = module5.broadcast([my_filter_1, my_filter_2,
my_filter_3])

In [10]: module5.iterate_file('filter_example',
broadcaster)

-->140.180.132.213 - - [24/Feb/2008:00:08:59 -0600]
"GET /favicon.ico HTTP/1.1" 404 133

-->75.54.118.139 - - [24/Feb/2008:00:15:41 -0600]
"GET /images/Davetubes.jpg HTTP/1.1" 200 6002
```

◀ **# We create a broadcaster to all filters in the middle of the pipeline: Plug and Play style**

◀ **# We plug it as source of the filters**

◀ **# Magic! It works**

From Pipelines to Concurrency: How Coroutines Changed the World

Can we enable concurrency without threads?

Dispatching Coroutines

```
@coroutine
def broadcast(targets):
    for target in targets:
        next(target)
    while True:
        item = yield
        try:
             np.random.choice(np.array(targets)).send(item)
        except StopIteration as e:
            raise e

def good_worker(target, worker_id=1):
   while True:
        subarray = yield
        coprint(f'Good worker {worker_id}: Got:{subarray}')
        try:
            if subarray is None:
                target.send(subarray)
            else:
                target.send(sum(subarray))
        except StopIteration as e:
            raise e

def slow_worker(target, worker_id=1):
    # Same as good worker but sleeps 10 seconds before
sending
```

- **# Prime workers**
- **# Select a random worker to dispatch work**
- **# If value is Sentinel, propagate. If not calculate the sum.**
- **# Slow worker simulated higher overload**

Dispatching Coroutines

```
@coroutine
def accumulator():
    coprint(f'Accumulator : Time to accumulate!')
    result = namedtuple('Sum', ['sum'])
    add = 0
    while True:
        value = yield
        if value is None:
            coprint(f'Accumulator : Got None,
reporting!')
            break
        coprint(f'Accumulator : Got {value}!')
        add += value
    coprint(f'Accumulator: Final result was
{result(sum=add)}')
    return result(sum=add)
```

◀ **# Get a value**

◀ **# If its the Sentinel, return the Sum**

◀ **# If not accumulate the sums.**

Dispatching Coroutines

```
In [7]: arraylist = np.random.randint(0, 10, size=20)
  ...: accumulator = module5.accumulator()
   ...: good_worker_1 = module5.good_worker(accumulator)
  ...: module5.dispatch_work(arraylist, [good_worker_1])
Array: [8 4 5 6 0 2 2 3 4 7 7 0 6 0 3 4 8 8 3 8] and its
sum is 88
Accumulator : Time to accumulate!
Good worker: Time to work!
Dispatcher: Subarrays to send are [array([8, 4, 5, 6]), array([0, 2, 2, 3]),
array([4, 7, 7]), array([0, 6, 0]), array([3, 4, 8]), array([8, 3, 8])]
Dispatcher: Sending subarray [8 4 5 6]
Good worker: Got: [8 4 5 6]
Accumulator : Got 23!

... # more prints like this one # ...

Dispatcher: Sending subarray [8 3 8]
Good worker: Got: [8 3 8]
Accumulator : Got 19!
Dispatcher: Subarrays sent, ending!
Good worker: Got: None
Accumulator : Got None, reporting!
Accumulator: Final result was Sum(sum=88)
Dispatcher: Got exception, its value is Sum(sum=88)
```

◀ **# We will calculate the sum of this list, we print its sum for debugging**

◀ **# We catch the Sum and it is correct!**

Dispatching Coroutines

```
Array: [5 4 6 1 5 5 4 1 3 1 8 2 6 4 3 5 9 5 3 7] and its sum is 87
2020-02-09 04:18:32.486284:Accumulator : Time to accumulate!
2020-02-09 04:18:32.486640:Slow worker 0: Time to work!
2020-02-09 04:18:32.486651:Slow worker 1: Time to work!
2020-02-09 04:18:32.486657:Slow worker 2: Time to work!
2020-02-09 04:18:32.486660:Slow worker 3: Time to work!
2020-02-09 04:18:32.486664:Slow worker 4: Time to work!
2020-02-09 04:18:32.486993:Dispatcher: Subarrays to send are [array([5,
4, 6, 1]), array([5, 5, 4, 1]), array([3, 1, 8]), array([2, 6, 4]), array([3, 5, 9]),
array([5, 3, 7])]
2020-02-09 04:18:32.487049:Dispatcher: Sending subarray [5 4 6 1]
2020-02-09 04:18:32.487156:Slow worker 2: Got: [5 4 6 1]
2020-02-09 04:18:37.492152:Accumulator : Got 16!
2020-02-09 04:18:37.492344:Dispatcher: Sending subarray [5 5 4 1]
## more results##
2020-02-09 04:18:57.507412:Slow worker 0: Got: [5 3 7]
2020-02-09 04:19:02.507876:Accumulator : Got 15!
2020-02-09 04:19:02.507908:Dispatcher: Subarrays sent, ending!
2020-02-09 04:19:02.507989:Slow worker 2: Got: None
2020-02-09 04:19:07.511732:Accumulator : Got None, reporting!
2020-02-09 04:19:07.511811:Accumulator: Final result was Sum(sum=87)
2020-02-09 04:19:07.511898:Dispatcher: Got exception, its
value is Sum(sum=87)
```

◀ **# With more “concurrency”**

◀ **# In fact we are being sequential!**

◀ **# Of course the Sum is correct**

A Fix: Threads

```
@coroutine
def threaded(target):
    messages = Queue()

    Thread(target=run_target,
args=(messages, target)).start()

    try:
        while True:
            item = yield
            messages.put(item)

    except GeneratorExit:
        messages.put(GeneratorExit)
```

◀ # We work with an atomic message queue

◀ # That will get subarrays

◀ # And send them to the queue for the workers in threads to fetch

◀ # We need to close the threads correctly if we get closed

A Fix: Threads

```
def run_target(queue, target):
    while True:
        item = queue.get()
        if item is GeneratorExit:
            target.close()
            return
        else:
            try:
                target.send(item)
            except StopIteration as e:
                coprint(f'Dispatcher: Got
exception, its value is {e.value}')
```

◀ **# We get elements in the queue. This is not a coroutine**

◀ **# If we get the closing call, we close our target**

◀ **# Else we send the item to the worker**

A Fix? Threads!

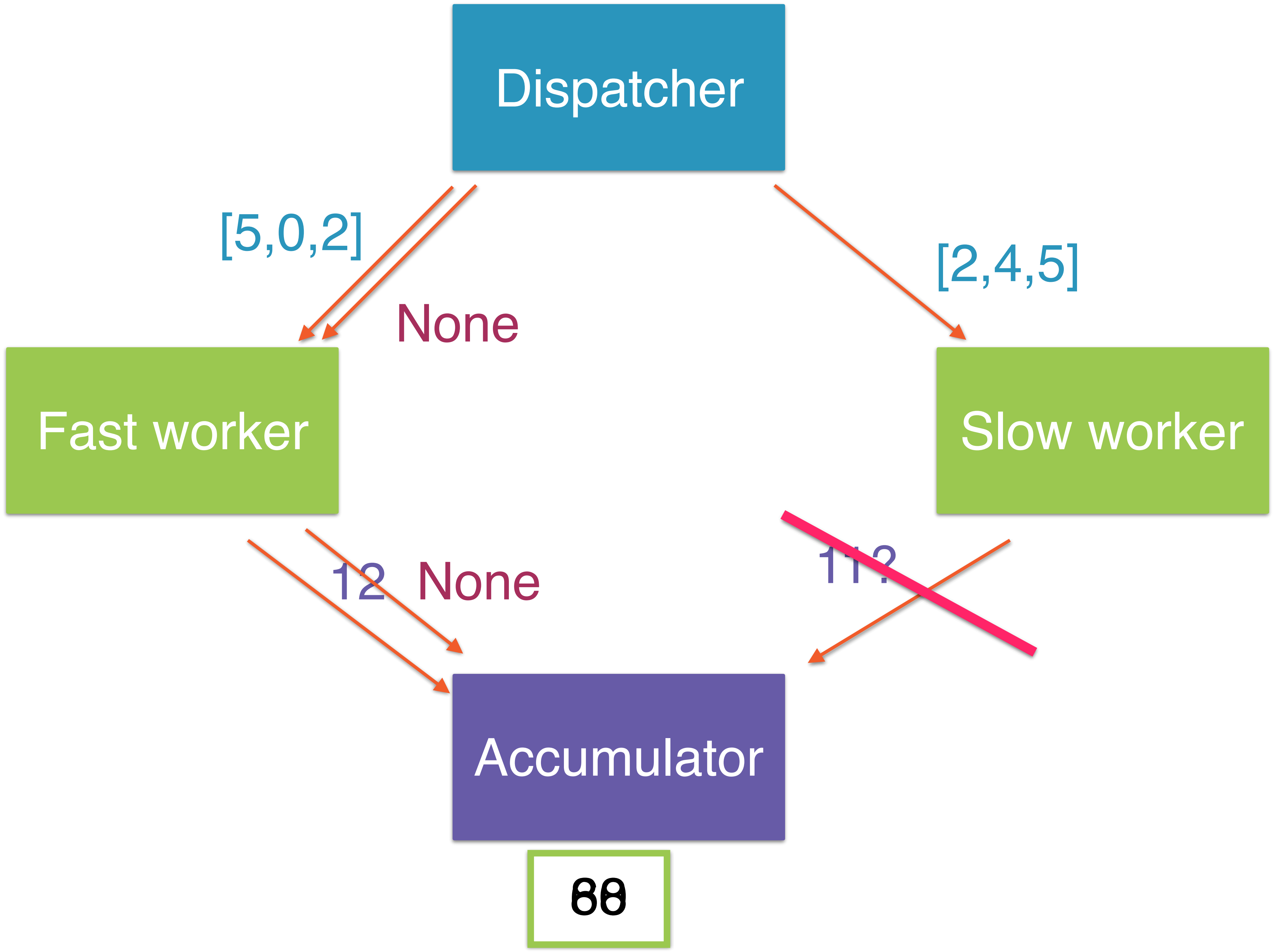
```
Array: [3 1 7 0 7 9 4 1 0 8 7 2 9 3 8 2 9 1 7 3] and its sum is 91
2020-02-09 04:51:38.136259:Good worker 1: Time to work!
2020-02-09 04:51:38.136288:Slow worker 0: Time to work!
2020-02-09 04:51:38.136612:Dispatcher: Subarrays to send are [array([3, 1,
7, 0]), array([7, 9, 4, 1]), array([0, 8, 7]), array([2, 9, 3]), array([8, 2, 9]),
array([1, 7, 3])]
2020-02-09 04:51:38.136664:Dispatcher: Sending subarray [3 1 7 0]
# more sent subarrays#

2020-02-09 04:51:38.137215:Dispatcher: Sending subarray [1 7 3]
2020-02-09 04:51:38.137253:Dispatcher: Subarrays sent, ending!
2020-02-09 04:51:38.137397:Good worker 0: Got: [7 9 4 1]
2020-02-09 04:51:38.137413:Accumulator : Got 21!
2020-02-09 04:51:38.137698:Good worker 0: Got: [1 7 3]
2020-02-09 04:51:38.137711:Accumulator : Got 11!
2020-02-09 04:51:38.137879:Slow worker 0: Got: [3 1 7 0]
2020-02-09 04:51:38.137987:Good worker 1: Got: [0 8 7]
2020-02-09 04:51:38.138006:Accumulator : Got 15!
2020-02-09 04:51:38.138020:Good worker 1: Got: None
2020-02-09 04:51:38.138025:Accumulator : Got None, reporting!
2020-02-09 04:51:38.138051:Accumulator: Final result was Sum(sum=80)
2020-02-09 04:51:38.138086:Dispatcher: Got exception, its
value is Sum(sum=80)
```

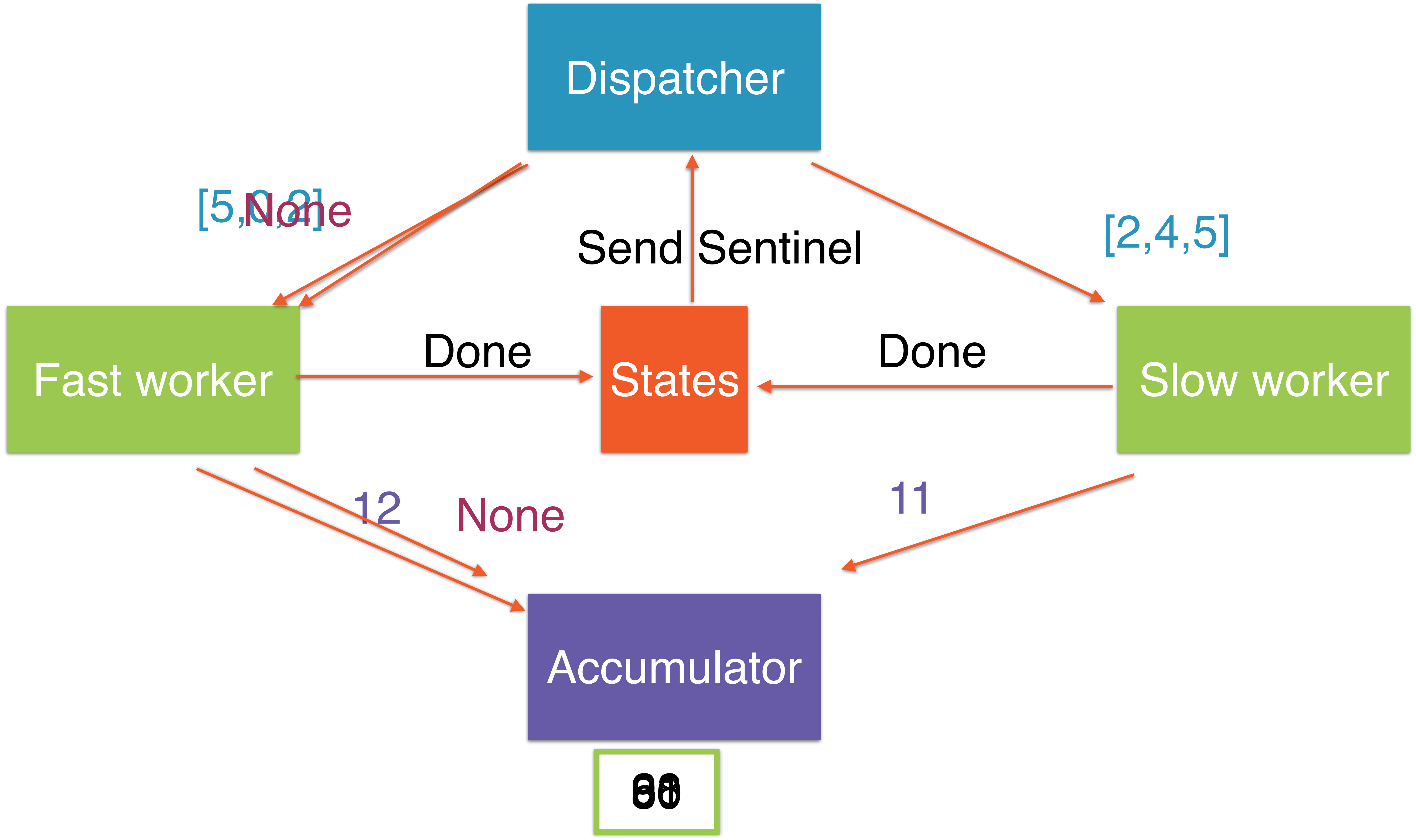
◀ **# Now we have concurrency!**

◀ **# And a BUG! We are missing the slow worker result!**

Explaining the Problem



Fixing the Problem



An Implementation

```
def dispatch_workers(target, workers=0):
# stuff...

    accumulate = accumulator()
    states = Queue()
    for i in range(workers):
        targets.append(
            threaded(
                target(accumulate, worker_id=i),
                 state_queue=states))

    broadcaster = broadcast(targets)
# stuff...
    for _ in range(int(subarrays_number)):
        print(f'Getting states --> {states.get()}')
    broadcaster.send(None)
```

◀ **# We create another queue**

◀ **# We send it to the threads**

◀ **# Later we check status**

Demo

Create a script that will dispatch work to coroutines in a non blocking, single threaded way

Summary

Coroutines can be chained to created data pipelines

Coroutines can be dispatched work as we did with the accumulator

Coroutines are tasks

We created an event loop!

Further references: David Beasley

Generator Tricks for Systems Programmers

A Curious Course on Coroutines and Concurrency.

Generators: The Final Frontier

Further references: James Powell

Generators will free your mind

Python Generators

More about Generators