### Distributed task processing using Celery

**Narahari (Hari) Allamraju**

**PyCon India 2013**

# 

# 

# Introduction

Celery is a distributed task queue for Python. It is different from MapReduce because instead of applying a given mapper function on a large set of data and then aggregating (reducing) the results, in Celery you define small self contained tasks and then execute them in large number across a set of worker nodes. Each task can work on just a set of arguments which will maybe later form a part of a larger solution.

Celery is written in Python and is very easy to use with minimal setup required for the workers or for the process creating these tasks that are executed on the workers.

For more in depth details on Celery you can visit [www.celeryproject.com](http://www.celeryproject.com).

This document is a tutorial on Celery - how to setup Celery, how to use it for simple cases and how we can do some complex workflows with Celery. The assumption is that you know Python, any level of expertise, and can understand a little bit about how and why we would like to distribute task execution - never the less we will start with a brief introduction of why we would like to distribute task execution.

We will be looking at Celery, using Redis as broker, and we will be looking at a Celery monitoring tool called Flower. Celery can work with any backend for passing around the tasks and task results - which broker backend we use will ideally depend on the use case and the performance requirements that we have. However for the sake of this tutorial we will use Redis because it is very simple to setup and use.

So lets begin.

# 

# A very quick overview of why we want to distribute task execution and what to watch out for

When we take up a problem to solve, we write some code for that and we run it on a given data set and we get a result. Depending on the data we have many ways in which we can process it or iterate over it but in the end the program is run on a machine and we get the result. So where does distribution of tasks come in?

As we all know problems aren’t that simple in the real world anymore - you need to read a socket or do a database connection to get data, wait for something or do data processing without holding up a resource etc etc - and we solve this using threads and all those mechanisms. This works pretty well in most cases and in most languages other than Python too. But of you take a large problem and keep on going at it solving a little bit at a time, then you are efficient only up to a certain point. There comes a point where the problem becomes too big that you cannot solve it with reasonable efficiency if you just took one optimized process and ran it on the problem no matter how many resources you throw at it.

If your problem dealt with a large data set and an operation that you want to perform on it - you will go the MapReduce way where you split the data into chunks and then take your small operation and apply it to all the chunks to get some result. Then maybe you will aggregate the results from each of this or you will just keep the results.

But what will you do if you problem is not a large data set and a small operation on that but a large number of small operations that need to be done efficiently? If you were trying to solve it the old way you probably queued the operations in some way and had a process work on the queue and finish the tasks. Now this works but as the queue gets bigger it becomes very inefficient.

This is where you come to task distribution and Celery (or some other such tool). You build the same queue of operations, but now you build each operation in such a way that it is completely self contained and independent of where or what executes the task, may be idempotent too, and then you put this on the queue. And instead of running a single process, you will start a set of process that all connect to the same queue, pick a task and execute that task. Since if you have N worker process then your total execution time has been reduced by a factor of N.

Of course you will have to factor the overhead of splitting and distributing the tasks and maybe collating the results and handling the failures if any but overall you will be very efficient in your task execution as compared if you had a single process reading your queue.

Couple of things to note about task distribution, mainly around how you should be designing your tasks and how they will be executed etc -

1. The tasks will be distributed across a set of workers and in most cases there is no guarantee that two tasks will run on the same worker node or that they will run one after the other - so the rule of thumb is that each task should be a self contained task with no dependencies on other tasks.
2. It is also good to avoid waiting for external dependencies in each task because that will mean that the task cannot be completed and the worker node will have to wait for the resource - this will defeat the purpose of distributing tasks if they have to end up waiting anyways.
3. Another thing is that sometimes because the task failed or because the framework which is distributing the tasks, Celery for example, cannot determine if the task was completed properly and may try to execute the task again. In these cases it makes sense to ensure that the tasks are idempotent - that is no matter how many times you run the task you get the same result. If the task has to manipulate some state then it makes sense to check the state in the task code and update it only if it has not been already updated.
4. Depending on the framework, when the task is sent to a worker, the code for the task might or might not be sent across. The framework might just send the task name and the arguments. So you need to ensure that all your worker nodes have visibility to the same version of the code.
5. As an extension to #4, if you are making changes to the tasks while the distributed workers are running, then it will help to have a mechanism to restart the workers so that they pick up the latest code.

These things apply to any general framework that helps you distribute task execution. With this in mind, lets jump into Celery now.

# 

# Installation and basics units of Celery

Celery is pretty simple to install. You can use a virtualenv or just install it into your python installation directly. It is as simple as

pip install celery

There are also bundles that will install all the dependencies for your choice of a broker. So if Redis was the backend that you want to use as a broker then you can install using

pip install -U celery-with-redis

For more options please see the Celery website.

When you install the package, you will get a bunch of Celery binaries. The main ones that we will cover are

celery - this is what you use to start workers, scheduler etc

celeryd - you use this to run the worker as a daemon

celerybeat - you use this to start the worker for scheduled periodic tasks

We will mostly cover the celery binary in this tutorial. You can read more about the others on the website.

Another tool to install is a monitoring tool for Celery called ‘flower’. You can install it using the following command.

pip install -U flower

This has an inbuilt web server that provides a web based monitoring interface for all the workers that you will be running. This provides a very neat and detailed view with charts etc.

# 

# 

# A simple use case

Lets start with a simple use case of how we can use Celery - this is a very basic example, and we will get into a more complex example later.

Lets dive in to the code directly -

Example 1

from celery import Celery

celery = Celery('tut1',broker='redis://')

@celery.task

def add(a,b):

print a+b

Example 2

from celery import Celery

celery = Celery('tut2',broker='redis://',backend='redis://')

@celery.task

def add(a,b):

print a+b

@celery.task

def mult(a,b):

return a\*b

Very simple examples - lets execute them and see how they work.

How exactly do we invoke a task?

* T.delay(arg, kwarg=value)
  + always a shortcut to .apply\_async.
* T.apply\_async((arg, ), {'kwarg': value})
* T.apply\_async(countdown=10)
  + executes 10 seconds from now.
* T.apply\_async(eta=now + timedelta(seconds=10))
  + executes 10 seconds from now, specifed using eta
* T.apply\_async(countdown=60, expires=120)
  + executes in one minute from now, but expires after 2 minutes.
* T.apply\_async(expires=now + timedelta(days=2))
  + expires in 2 days, set using [datetime](http://docs.python.org/dev/library/datetime.html#datetime.datetime).

# Dissecting how it all works

Now that we have seen a simple example in action, lets dig into how the whole thing actually works. This will help us design better tasks when we go to more complex problems.

The main part of both these examples was the Celery application. Not going very deep into the history, in the old versions there was a celery.task module that was used to create the celery tasks instead of the way we did - this is still available for backwards compatibility, but we will not get into using that.

The Celery application is what defines how Celery defines the tasks and where it executes them. Celery always creates a default app in case you do not set one - but it is advisable not to depend on that and create your own app. The app does 3 main things in our examples -

* It defines the application name which is used to name the tasks and put them in proper name spaces
* It defines the backend or broker that we are using to pass on the messages for the tasks to the workers
* And in the second example it defines the backend that is to be used to store any results from the execution of the tasks here.

When we instantiate the Celery() application, it creates a new configuration - a new instance - and is therefore thread safe. The first argument is the name of the application and the namespace used to name the tasks. For example if we did not give this and we ran this example with a \_\_main\_\_ function then it will name the task as \_\_main\_\_.add and \_\_main\_\_.mult.

Since we have passed in the application name, the tasks will be names as tut1.add, tut2.add and tut2.mult. So if we were to run tut1 and tut2 in the same shell, then there is no conflict for the add task.

When we add the decorator on the functions to create tasks from them, the application is not creating the task - the task is lazily created when we execute it, and it applies some finalization logic. It creates an instance of the Task class which handles the life cycle of task execution in Celery including storing the results etc.

When we execute a task, the code for the task is not passed to the worker. It is expected that all the worker nodes are looking at the same version of the code. The task name, the parameters and any other information like retry attempts etc are serialized by Celery into pickle format or any serializer that you may specify, like json, and sent as a message on the backend. This message stays alive and does not disappear until the task is acknowledged as completed. So it means that say the worker that was doing the task dies, then someone else can pick up the task.

As you can imagine, the task cannot return a value to the calling code - so our add example just prints to the logs, and the mult example will store the result in the backend - if we ran it without the backend in the Celery application arguments, it would fail to store the result. The Task class handles all the internals of how to find and get this stored result for us when we need it.

Obviously, since tasks can execute on any worker in any order, they cannot depend on order of execution. Another possibility is that when a worker dies, and another worker picks up the task message for execution again, some of the work might have been done by the previous worker - in case your task stores or changes more data, apart from just the result, then it is better to check in the task whether that is already done to avoid issues.

And yes, when a task is completed, the status of the task is updated accordingly and can be seen in the backend. We will see more of this when we look at the monitoring tool later.

# A more complex example

Lets take this with an example so that we get a fair idea on how we will design tasks for Celery if we were building a real life business application. The example we will take is that of FX trading, where at the end of the day, or at some point for example, we will need to calculate the value of all the positions that we have on a particular currency pair. To keep it simple we will use a single currency pair here, but you will see you can easily iterate that handle more than one currency.

The main things that we will consider are

1. Creating a bunch of positions
2. Taking a given price and evaluating all the position values

In order to simulate a real world application, we will make a bunch of positions in the backend, and then we will have the task fetch the data from the backend before it does the main logic. We will not see too much into exception handling and all that but you will see it is pretty easy to add such checks.

Lets get straight into the code

Create the positions -

import redis

import simplejson as json

#the database

db=redis.Redis()

#the positions

positions=[

{'name':'pos\_1','size':100,'price':0.99},

{'name':'pos\_2','size':100,'price':0.91},

{'name':'pos\_3','size':100,'price':0.85},

{'name':'pos\_4','size':100,'price':0.96},

{'name':'pos\_5','size':100,'price':0.79}

]

def create\_positions():

for position in positions:

db.set(position['name'],json.dumps(position))

if \_\_name\_\_=="\_\_main\_\_":

create\_positions()

print json.loads(db.get('pos\_1'))

Process the positions -

import simplejson as json

from celery import Celery

from celery.utils.log import get\_task\_logger

import redis

celery=Celery('positions',broker='redis://',backend='redis://')

logger = get\_task\_logger(\_\_name\_\_)

@celery.task

def get\_initial\_position\_value(name):

db=redis.Redis()

pos=json.loads(db.get(name))

return pos['size']\*pos['price']

@celery.task

def get\_value\_at\_close(name,close):

db=redis.Redis()

pos=json.loads(db.get(name))

return pos['size']\*close

@celery.task

def get\_value\_at\_close\_1(name,close):

db=redis.Redis()

pos=json.loads(db.get(name))

logger.info('Calculating value with close price %s'%(close,))

return pos['size']\*close

Another version of the code with some more checks, which we would have if this were to run as production code - just basic checks here

import simplejson as json

from celery import Celery

from celery.utils.log import get\_task\_logger

import redis

celery=Celery('positions',broker='redis://',backend='redis://')

logger = get\_task\_logger(\_\_name\_\_)

@celery.task

def get\_initial\_position\_value(name):

db=redis.Redis()

pos=json.loads(db.get(name))

if pos is None:

return

else:

return pos['size']\*pos['price']

@celery.task

def get\_value\_at\_close(name,close):

if close is None or close<0:

return None

db=redis.Redis()

pos\_str=db.get(name)

if pos\_str is None:

return None

else:

pos=json.loads(pos\_str)

return pos['size']\*close

@celery.task

def get\_value\_at\_close\_retry(name,close):

if close is None or close<0:

return None

db=redis.Redis()

pos\_str=db.get(name)

if pos\_str is None:

#raise get\_value\_at\_close\_retry.retry(exc=exc) - if we were in a except

raise get\_value\_at\_close\_retry.retry()

else:

pos=json.loads(pos\_str)

return pos['size']\*close

# 

# 

# Canvas - grouping, linking etc

If we think of the above examples in a real production scenario, we will not be checking the value of each position one by one. We will want to run all positions at once and see the values. We would want to run the tasks in parallel because we want to maximize the speed. Similarly we will also want to maybe add up all the results to know the total value or maybe total profit and loss for that day.

This is how we can do that.

import simplejson as json

from celery import Celery

from celery.utils.log import get\_task\_logger

import redis

celery=Celery('positions',broker='redis://',backend='redis://')

logger = get\_task\_logger(\_\_name\_\_)

@celery.task

def get\_pnl(name,close):

if close is None or close<0:

return None

db=redis.Redis()

pos\_str=db.get(name)

if pos\_str is None:

return None

else:

pos=json.loads(pos\_str)

current = pos['size']\*close

initial = pos['size']\*pos['price']

return current-initial

@celery.task

def add\_pnl(val1=0,val2=0):

return val1+val2

This is how we can use these in groups and chains

import pnl

from celery import chain,group

import time

#independent tasks in parallel

res=group(pnl.get\_pnl.si('pos\_%s'%(i,), 0.95) for i in range(1,6))()

print res.ready()

print res.get(timeout=1)

#like a chain, but stil independent tasks

res1=chain(pnl.get\_pnl.si('pos\_1', 0.95),pnl.get\_pnl.si('pos\_2', 0.95),pnl.get\_pnl.si('pos\_3', 0.95))()

print res1.get()

print res1.parent.get()

print res1.parent.parent.get()

#chain of sub tasks addig up the result

s1=pnl.get\_pnl.delay('pos\_1', 0.95)

s2=pnl.get\_pnl.delay('pos\_2', 0.95)

s3=pnl.get\_pnl.delay('pos\_3', 0.95)

#just wait a few secs

time.sleep(5)

print s1.result,s2.result,s3.result

res2=chain(pnl.add\_pnl.s(0,s1.result),pnl.add\_pnl.s(s2.result),pnl.add\_pnl.s(s3.result))()

print res2.get()

print res2.parent.get()

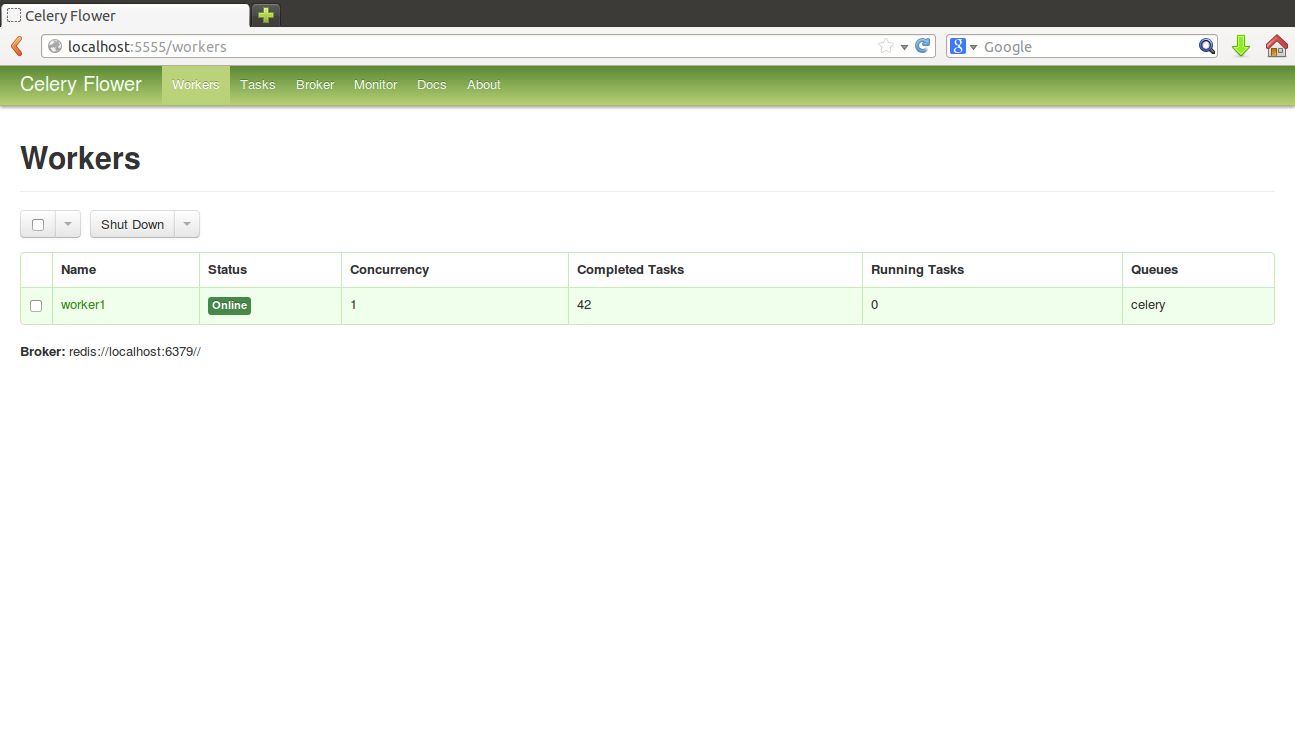
print res2.parent.parent.get()

# 

# 

# Monitoring Celery

You can monitor Celery using flower - it provides a very neat web interface. Lets dig straight into the interface and see how it looks.



# 