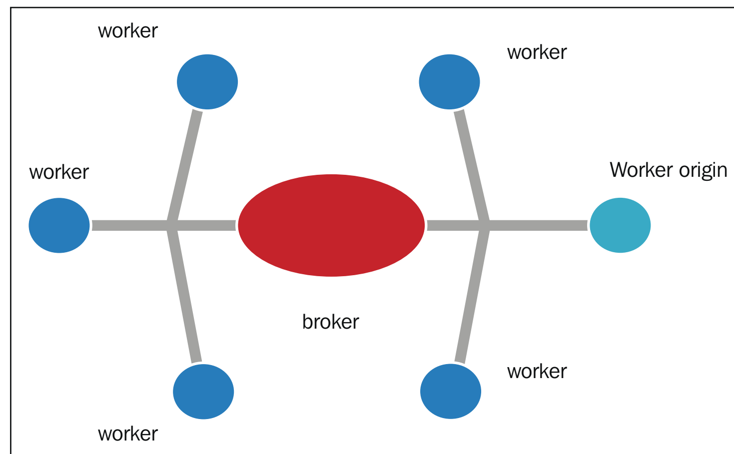
There's more...

If RabbitMQ operates under its default configuration, Celery can connect with no other information other than `amqp://scheme`.

Scientific computing with SCOOP

Scalable Concurrent Operations in Python (**SCOOP**) is a Python module to distribute concurrent tasks (called **Futures**) on heterogeneous computational nodes. Its architecture is based on the **ØMQ** package, which provides a way to manage Futures between the distributed systems. The main application of SCOOP resides in scientific computing that requires the execution of many distributed tasks using all the computational resources available.

To distribute its futures, SCOOP uses a variation of the broker patterns:



The SCOOP architecture

The central element of the communication system is the broker that interacts with all the independent workers to dispatch messages between them. The Futures are created in the worker elements instead of the central node (the broker) with a centralized serialization procedure. This makes the topology architecture more reliable and makes performance better. In fact, the broker's main workload consists of networking and interprocess I/O between workers with relatively low CPU processing time.

Getting ready

The SCOOP module is available at `https://github.com/soravux/scoop/` and its software dependencies are as follows:

- Python >= 2.6 or >= 3.2
- Distribute >= 0.6.2 or setuptools >= 0.7
- Greenlet >= 0.3.4
- pyzmq >= 13.1.0 and libzmq >= 3.2.0
- SSH for remote execution

SCOOP can be installed on Linux, Mac, and Windows machines. Like Disco, its remote usage requires an SSH software, and it must be enabled as a password-less authentication between every computing node. For a complete reference about the SCOOP installation procedure, you can read the information guide at `http://scoop.readthedocs.org/en/0.7/install.html`.

On a Windows machine, you can install SCOOP simply by typing the following command:

```
pip install SCOOP
```

Otherwise, you can type the following command from SCOOP's distribution directory:

```
Python setup.py install
```

How to do it...

SCOOP is a library full of functionality that is primarily used in scientific computing problems. Among the methods used to find a solution to these problems that are computationally expensive, there is the Monte Carlo algorithm. A complete discussion of this method would take up many pages of a book, but in this example, we want to show you how to parallelize a Monte Carlo method for the solution of the following problem, the calculation of the number π, using the features of SCOOP. So, let's consider the following code:

```
import math
from random import random
from scoop import futures
from time import time
```

```
def evaluate_number_of_points_in_unit_circle(attempts):
    points_fallen_in_unit_disk = 0
    for i in range (0,attempts) :
        x = random()
        y = random()
        radius = math.sqrt(x*x + y*y)
        #the test is ok if the point fall in the unit circle
        if radius < 1 :
            #if ok the number of points in a disk is increased
            points_fallen_in_unit_disk = \
                    points_fallen_in_unit_disk + 1
    return points_fallen_in_unit_disk


def pi_calculus_with_Montecarlo_Method(workers, attempts):
    print("number of workers %i - number of attempts %i"
%(workers,attempts))
    bt = time()
    #in this point we call scoop.futures.map function
    #the evaluate_number_of_points_in_unit_circle \
    #function is executed in an asynchronously way
    #and several call this function can be made concurrently
    evaluate_task = \
                futures.map(evaluate_points_in_circle,
                       [attempts] * workers)
    taskresult= sum(evaluate_task)
    print ("%i points fallen in a unit disk after " \
           %(Taskresult/attempts))
    piValue = (4. * Taskresult/ float(workers * attempts))

    computationalTime = time() - bt
    print("value of pi = " + str(piValue))
    print ("error percentage = " + \
           str((((abs(piValue - math.pi)) * 100) / math.pi)))
    print("total time: " + str(computationalTime))

if __name__ == "__main__":
    for i in range (1,4):
        #let's fix the numbers of workers...only two,
        #but it could be much greater
        pi_calculus_with_Montecarlo_Method(i*1000, i*1000)
        print(" ")
```

To run a SCOOP program, you must open Command Prompt and type the following instructions:

```
python –m scoop name_file.py
```

For our script, we'll expect output like this:

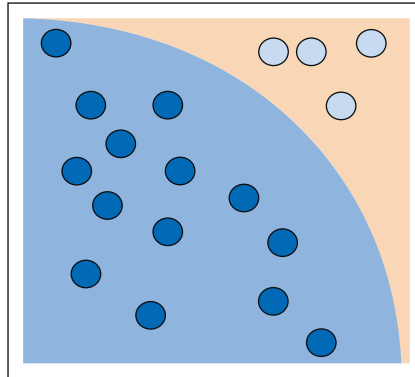
```
C:\Python CookBook\Chapter 5 - Distributed Python\chapter 5 -
codes>python -m scoop pi_calculus_with_montecarlo_method.py
[2015-06-01 15:16:32,685] launcher  INFO    SCOOP 0.7.2 dev on win32
using Python 3.3.0 (v3.3.0:bd8afb90e
bf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)], API: 1013
[2015-06-01 15:16:32,685] launcher  INFO    Deploying 2 worker(s) over 1
host(s).
[2015-06-01 15:16:32,685] launcher  INFO    Worker d--istribution:
[2015-06-01 15:16:32,686] launcher  INFO       127.0.0.1:       1 +
origin
Launching 2 worker(s) using an unknown shell.
number of workers 1000 - number of attempts 1000
785 points fallen in a unit disk after
value of pi = 3.140636
error percentage = 0.03045122952842962
total time: 10.258585929870605

number of workers 2000 - number of attempts 2000
1570 points fallen in a unit disk after
value of pi = 3.141976
error percentage = 0.012202295220195048
total time: 20.451170206069946

number of workers 3000 - number of attempts 3000
2356 points fallen in a unit disk after
value of pi = 3.1413777777777776
error percentage = 0.006839709526630775
total time: 32.3558509349823

[2015-06-01 15:17:36,894] launcher  (127.0.0.1:59239) INFO    Root
process is done.
[2015-06-01 15:17:36,896] launcher  (127.0.0.1:59239) INFO    Finished
cleaning spawned subprocesses.
```

The correct value of pi becomes more precise as we increase the number of attempts and workers.



Monte Carlo evaluation of π: counting points inside the circle

How it works...

The code presented in the preceding section is just one of the many implementations of the Monte Carlo method for the calculation of π. The `evaluate_ points_in_circle ()` function is taken randomly and then given a point of coordinates (*x*, *y*), and then it is determined whether or not this point falls within the circle of the unit area.

Whenever the `points_fallen_in_unit_disk` condition is verified, the variable is incremented. When the inner loop of the function ends, it will represent the total number of points falling within the circle. This number is sufficient to calculate the value of pi. In fact, the probability that the point falls within the circumference is π / 4, that is the ratio between the area of the unit circle, equal to π and the area of the circumscribed square equal to 4.

So, by calculating the ratio between the number of points fallen inside the disc, `taskresult`, and the number of shots made, *workers * attempts*, you obtain an approximation of π/4 and of course, also of the number π:

```
piValue = (  4. * taskresult / float (workers attempts *))
```

The SCOOP function is as shown:

```
futures.map (evaluate_points_in_circle, [attempts] * workers)
```

This takes care of distributing the computational load between the available workers and at the same time, collects all the results. It executes `evaluate_points_in_circle` in an asynchronous way and makes several calls to `evaluate_points_in_circle` concurrently.

Handling map functions with SCOOP

A common task that is very useful when dealing with lists or other sequences of data is to apply the same operation to each element of the list and then collect the result. For example, a list update may be done in the following way from the Python IDLE:

```
>>>items = [1,2,3,4,5,6,7,8,9,10]
>>>updated_items = []
>>>for x in items:
>>>    updated_items.append(x*2)

>>> updated_items
>>>  [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

This is a common operation. However, Python has a built-in feature that does most of the work.

The Python function `map(aFunction, aSequence)` applies a passed-in function to each item in an iterable object and returns a list containing all the function call results. Now, the same example would be:

```
>>>items = [1,2,3,4,5,6,7,8,9,10]
>>>def multiplyFor2(x):return x*2
>>>print(list(map(multiplyFor2,items)))
>>>[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Here, we passed in the `map` function the user-defined function `multiplyFor2`. It is applied to each item in the items list, and finally, we collect the result in a new list that is printed.

Also, we can pass in a `lambda` function (a function defined and called without being bound to an identifier) as an argument instead of a function. The same example now becomes:

```
>>>items = [1,2,3,4,5,6,7,8,9,10]
>>>print(list(map(lambda x:x*2,items)))
>>>[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

The map built-in function has performance benefits because it is faster than a manually coded for loop.

Getting ready

The SCOOP Python modules define more than one map function that allow asynchronous computation that could be propagated to its workers. These functions are:

- `futures.map((func, iterables, kargs)`: This returns a generator that iterates the results in the same order as its inputs. It can thus act as a parallel substitute for the standard Python `map()` function.
- `futures.map_as_completed(func, iterables, kargs)`: This will yield results as soon as they are made available.
- `futures. scoop.futures.mapReduce(mapFunc, reductionOp, iterables, kargs)`: This allows us to parallelize a reduction function after we apply the `map()` function. It returns a single element.

How to do it…

In this example, we'll compare the `MapReduce` version of SCOOP with its serial implementation:

```
"""
Compare SCOOP MapReduce with a serial implementation
"""
import operator
import time

from scoop import futures


def simulateWorkload(inputData):
    time.sleep(0.01)
    return sum(inputData)

def CompareMapReduce():
    mapScoopTime = time.time()
    res = futures.mapReduce(
        simulateWorkload,
        operator.add,
        list([a] * a for a in range(1000)),
    )
    mapScoopTime = time.time() - mapScoopTime
    print("futures.map in SCOOP executed in {0:.3f}s \
           with result:{1}".format(
```

```
            mapScoopTime,
            res
            )
        )

        mapPythonTime = time.time()
        res = sum(
            map(
                simulateWorkload,
                list([a] * a for a in range(1000))
            )
        )
        mapPythonTime = time.time() - mapPythonTime
        print("map Python executed in: {0:.3f}s \
               with result: {1}".format(
            mapPythonTime,
            res
            )
        )

if __name__ == '__main__':
    CompareMapReduce()
```

To evaluate the script, you must type the following command:

```
python -m scoop map_reduce.py


> [2015-06-12 20:13:25,602] launcher  INFO    SCOOP 0.7.2 dev on win32
using Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC
v.1600 32 bit (Intel)], API: 1013
[2015-06-12 20:13:25,602] launcher  INFO Deploying 2 worker(s) over 1
host(s).
[2015-06-12 20:13:25,602] launcher  INFO Worker d--istribution:
[2015-06-12 20:13:25,602] launcher  INFO 127.0.0.1:       1 + origin
Launching 2 worker(s) using an unknown shell.
futures.map in SCOOP executed in 8.459s with result: 332833500
map Python executed in: 10.034s with result: 332833500
[2015-06-12 20:13:45,344] launcher  (127.0.0.1:2559) INFO    Root process
is done.
[2015-06-12 20:13:45,368] launcher  (127.0.0.1:2559) INFO    Finished
cleaning spawned subprocesses.
```

How it works...

In this example, we compare the SCOOP implementation of the `MapReduce` function with the serial implementation. The core of the script is the `CompareMapReduce()` function that contains the two implementations. Also in this function, we evaluate the execution time according to the following schema:

```
mapScoopTime = tme.time()
         #Run SCOOP MapReduce
mapScoopTime = time.time() – mapScoopTime

mapPythonTime = time.time()
         #Run serial MapReduce
mapPythonTime = time.time() - mapPythonTime
```

Then in the output, we report the resulting time:

```
futures.map in SCOOP executed in 8.459s with result: 332833500
map Python executed in: 10.034s with result: 332833500
```

To obtain the comparable execution time, we simulate a computational workload that introduces a `time.sleep` statement in the `simulatedWordload` function:

```
def simulateWorkload(inputData, chose=None):
    time.sleep(0.01)
    return sum(inputData)
```

The SCOOP implementation of `mapReduce` is as follows:

```
res = futures.mapReduce(
        simulateWorkload,
        operator.add,
        list([a] * a for a in range(1000)),
    )
```

The `futures-mapReduce` function has the following arguments:

- `simulateWork`: This will be called to execute the Futures. We also need to remember that a callable must return a value.
- `operator.add`: This will be called to reduce the Futures results. However, it also must support two parameters and return a single value.
- `list(...... )`: This is the iterable object that will be passed to the callable object as a separate Future.

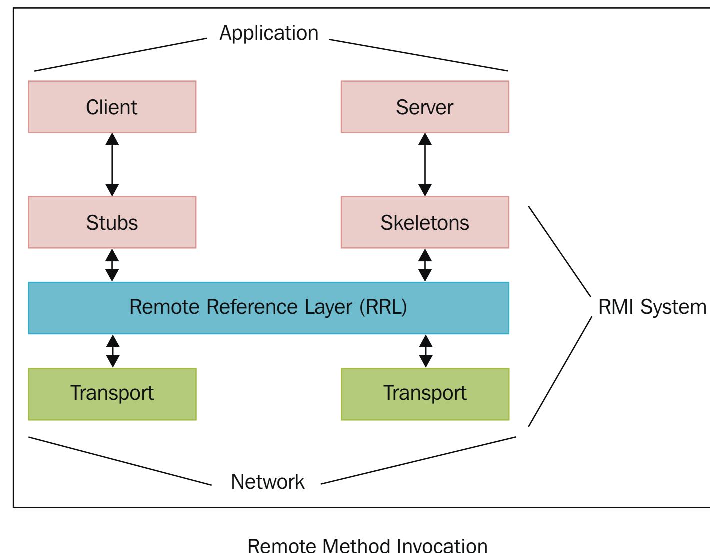
The serial implementation of `mapReduce` is, as follows:

```
res = sum(
    map(
        simulateWorkload,
        list([a] * a for a in range(1000))
    )
)
```

The Python standard `map()` function has two arguments: the `simulateWorkload` function and the `list()` iterable object. However, to reduce the result, we used the Python function `sum`.

Remote Method Invocation with Pyro4

Python Remote Objects (**Pyro4**) is a library that resembles Java's **Remote Method Invocation** (**RMI**), which allows you to invoke a method of a remote object (that belongs to a different process and is potentially on a different machine) almost as if the object were local (that is, it belonged to the same process in which it runs the invocation). In this sense, the Remote Method Invocation technology can be traced from a conceptual point of view. The idea of a **remote procedure call** (**RPC**) is reformulated for the object-oriented paradigm (in which, of course, the procedures are replaced by methods). The use of a mechanism for remote method invocation in an object-oriented system entails the significant advantages of uniformity and symmetry in the project, since it allows us to model the interactions between distributed processes using the same conceptual tool that is used to represent the interactions between the different objects of an application or the method call.



Remote Method Invocation