5
Distributed Python

In this chapter, we will cover the following recipes:

- Using Celery to distribute tasks
- How to create a task with Celery
- Scientific computing with SCOOP
- Handling map functions with SCOOP
- Remote method invocation with Pyro4
- Chaining objects with Pyro4
- Developing a client-server application with Pyro4
- Communicating sequential processes with PyCSP
- Using MapReduce with Disco
- A remote procedure call with RPyC

Introduction

The basic idea of distributed computing is to break each workload into an arbitrary number of tasks, usually indicated with the name, into reasonable pieces for which a computer in a distributed network will be able to finish and return the results flawlessly. In distributed computing, there is the absolute certainty that the machines on your network are always available (latency difference, unpredictable crash or network computers, and so on). So, you need a continuous monitoring architecture.

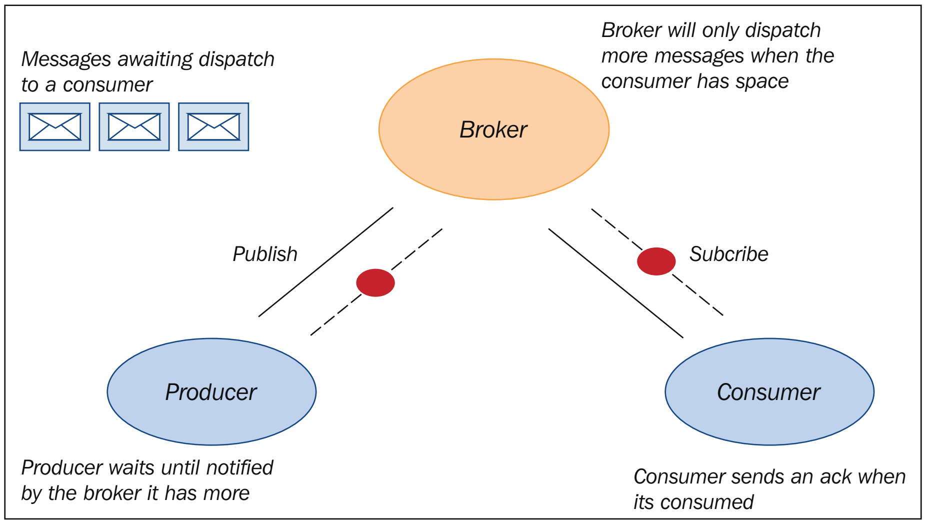
The fundamental problem that arises from the use of this kind of technology is mainly focused on the proper management of traffic (that is devoid of errors both in transmission and reception) of any kind (data, jobs, commands, and so on). Further, a problem stems from a fundamental characteristic of distributed computing: the coexistence in the network of machines that support different operating systems which are often incompatible with others. In fact, the need to actually use the multiplicity of resources in a distributed environment has, over time, led to the identification of different calculation models. Their goal is essentially to provide a framework for the description of the cooperation between the processes of a distributed application. We can say that, basically, the different models are distinguished according to a greater or lesser capacity to use the opportunities provided by the distribution. The most widely used model is the client-server model. It allows processes located on different computers to cooperate in real time through the exchange of messages, thereby achieving a significant improvement over the previous model, which requires the transfer of all the files, in which computations are performed on the data offline. The client-server model is typically implemented through remote procedure calls, which extend the scope of a local call, or through the paradigm of distributed objects (Object-Oriented Middleware).This chapter will then present some of the solutions proposed by Python for the implementation of these computing architectures. We will then describe the libraries that implement distributed architectures using the OO approach and remote calls, such as Celery, SCOOP, Pyro4, and RPyC, but also using different approaches, such as PyCSP and Disco, which are the Python equivalent of the MapReduce algorithm.

Using Celery to distribute tasks

Celery is a Python framework used to manage a distributed task, following the Object-Oriented Middleware approach. Its main feature consists of handling many small tasks and distributing them on a large number of computational nodes. Finally, the result of each task will then be reworked in order to compose the overall solution.

To work with Celery, we need the following components:

- The Celery module (of course!!)
- A message broker. This is a Celery-independent software component, the middleware, used to send and receive messages to distributed task workers. A message broker is also known as a message middleware. It deals with the exchange of messages in a communication network. The addressing scheme of this type of middleware is no longer of the point-to-point type but is a message-oriented addressing scheme. The best known is the Publish/Subscribe paradigm.


Messages awaiting dispatch to a consumer
Broker
Broker will only dispatch more messages when the consumer has space
Publish
Subcribe
Producer
Consumer
Producer waits until notified by the broker it has more
Consumer sends an ack when its consumed

The message broker architecture

Celery supports many types of message brokers—the most complete of which are RabbitMQ and Redis.

How to do it...

To install Celery, we use the `pip` installer. In Command Prompt, just type the following:

```
pip install celery
```

After this, we must install the message broker. There are several choices available for us to do this, but in our examples, we use RabbitMQ, which is a message-oriented middleware (also called broker messaging), that implements the **Advanced Message Queuing Protocol** (**AMQP**). The RabbitMQ server is written in Erlang, and it is based on the **Open Telecom Platform** (**OTP**) framework for the management of clustering and failover. To install RabbitMQ, download and run Erlang (`http://www.erlang.org/download.html`), and then just download and run the RabbitMQ installer (`http://www.rabbitmq.com/download.html`). It takes a few minutes to download and will set up RabbitMQ and run it as a service with a default configuration.

Finally, we install Flower (`http://flower.readthedocs.org`), which is a web-based tool used to monitor tasks (running progress, task details, and graphs and stats).

To install it, just type the following from Command Prompt:

```
pip install -U flower
```

Then, we can verify the Celery installation. In Command Prompt, just type the following:

```
C:\celery --version
```

After this, the text shown as follows should appear:

```
3.1.18 (Cipater)
```

The usage of Celery is pretty simple, as shown:

```
Usage: celery <command> [options]
```

Here, the options are as shown:

```
Options:
  -A APP, --app=APP     app instance to use (e.g. module.attr_name)
  -b BROKER, --broker=BROKER
                        url to broker.  default is 'amqp://guest@
localhost//'
  --loader=LOADER       name of custom loader class to use.
  --config=CONFIG       Name of the configuration module
  --workdir=WORKING_DIRECTORY
                        Optional directory to change to after
detaching.
  -C, --no-color
  -q, --quiet
  --version             show program's version number and exit
  -h, --help            show this help message and exit
```

See also

- For more complete details about the Celery installation procedure, you can visit `www.celeryproject.com`

How to create a task with Celery

In this recipe, we'll show you how to create and call a task using the Celery module. Celery provides the following methods that make a call to a task:

- `apply_async(args[, kwargs[, …]])`: This task sends a task message
- `delay(*args, **kwargs)`: This is a shortcut to send a task message, but does not support execution options

The `delay` method is better to use because it can be called as a regular function:

```
task.delay(arg1, arg2, kwarg1='x', kwarg2='y')
```

While using `apply_async` you should write:

```
task.apply_async (args=[arg1, arg2] kwargs={'kwarg1': 'x','kwarg2':
'y'})
```

How to do it...

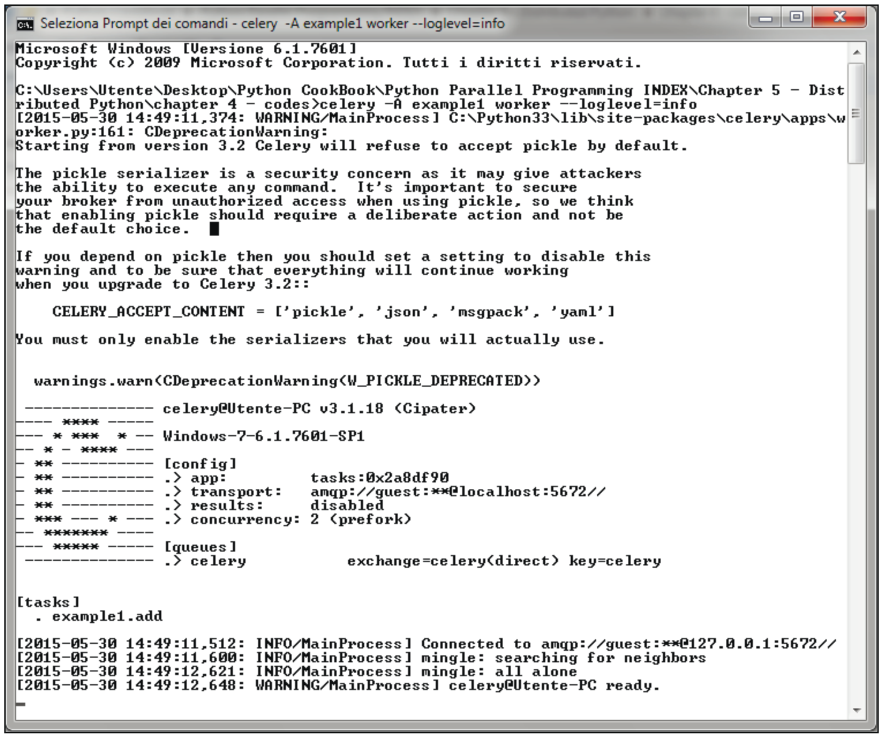
To perform this simple task, we implement the following two simple scripts:

```
###
## addTask.py :Executing a simple task
###

from celery import Celery

app = Celery('addTask',broker='amqp://guest@localhost//')

@app.task
def add(x, y):
    return x + y
while the second script is :

###
#addTask.py : RUN the AddTask example with
###

import addTask

if __name__ == '__main__':
    result = addTask.add.delay(5,5)
```

We must note again that the RabbitMQ service starts automatically on our server upon installation. So, to execute the Celery worker server, we simply type the following command from Command Prompt:

```
celery -A addTask worker --loglevel=info
```

The output is shown in the first Command Prompt:

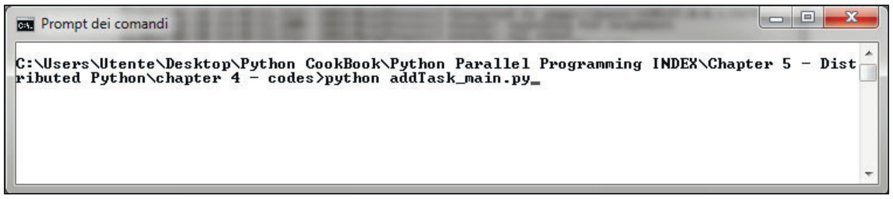


Seleziona Prompt dei comandi - celery -A example1 worker --loglevel=info

```
Microsoft Windows [Versione 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tutti i diritti riservati.

C:\Users\Utente\Desktop\Python CookBook\Python Parallel Programming INDEX\Chapter 5 - Dist
ributed Python\chapter 4 - codes>celery -A example1 worker --loglevel=info
[2015-05-30 14:49:11,374: WARNING/MainProcess] C:\Python33\lib\site-packages\celery\apps\w
orker.py:161: CDeprecationWarning:
Starting from version 3.2 Celery will refuse to accept pickle by default.

The pickle serializer is a security concern as it may give attackers
the ability to execute any command.  It's important to secure
your broker from unauthorized access when using pickle, so we think
that enabling pickle should require a deliberate action and not be
the default choice.

If you depend on pickle then you should set a setting to disable this
warning and to be sure that everything will continue working
when you upgrade to Celery 3.2::

    CELERY_ACCEPT_CONTENT = ['pickle', 'json', 'msgpack', 'yaml']

You must only enable the serializers that you will actually use.


  warnings.warn(CDeprecationWarning(W_PICKLE_DEPRECATED))

 -------------- celery@Utente-PC v3.1.18 (Cipater)
---- **** -----
--- * ***  * -- Windows-7-6.1.7601-SP1
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         tasks:0x2a8df90
- ** ---------- .> transport:   amqp://guest:**@localhost:5672//
- ** ---------- .> results:     disabled
- *** --- * --- .> concurrency: 2 (prefork)
-- ******* ----
--- ***** ----- [queues]
 -------------- .> celery           exchange=celery(direct) key=celery


[tasks]
  . example1.add

[2015-05-30 14:49:11,512: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.1:5672//
[2015-05-30 14:49:11,600: INFO/MainProcess] mingle: searching for neighbors
[2015-05-30 14:49:12,621: INFO/MainProcess] mingle: all alone
[2015-05-30 14:49:12,648: WARNING/MainProcess] celery@Utente-PC ready.
```

Let's note the warnings in the output to disable pickle as a serializer for security concerns. The default serialization format is pickle simply because it is convenient (it supports the task of sending complex Python objects as task arguments). Whether you use pickle or not, you may want to turn off this warning by setting the `CELERY_ACCEPT_CONTENT` configuration variable; for reference, take a look at `http://celery.readthedocs.org/en/latest/configuration.html`.

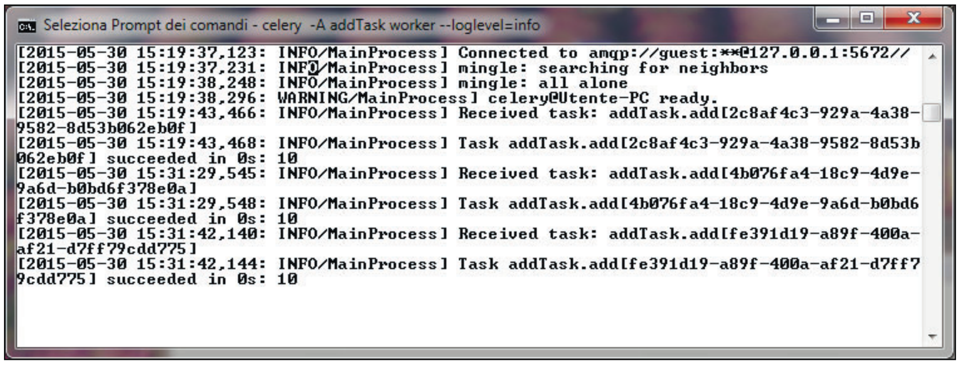
Now, we launch the `addTask_main` script from a second Command Prompt:



Prompt dei comandi

```
C:\Users\Utente\Desktop\Python CookBook\Python Parallel Programming INDEX\Chapter 5 - Dist
ributed Python\chapter 4 - codes>python addTask_main.py
```

Finally, the result from the first Command Prompt should be like this:


Seleziona Prompt dei comandi - celery -A addTask worker --loglevel=info

```
[2015-05-30 15:19:37,123: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.1:5672//
[2015-05-30 15:19:37,231: INFO/MainProcess] mingle: searching for neighbors
[2015-05-30 15:19:38,248: INFO/MainProcess] mingle: all alone
[2015-05-30 15:19:38,296: WARNING/MainProcess] celery@Utente-PC ready.
[2015-05-30 15:19:43,466: INFO/MainProcess] Received task: addTask.add[2c8af4c3-929a-4a38-
9582-8d53b062eb0f]
[2015-05-30 15:19:43,468: INFO/MainProcess] Task addTask.add[2c8af4c3-929a-4a38-9582-8d53b
062eb0f] succeeded in 0s: 10
[2015-05-30 15:31:29,545: INFO/MainProcess] Received task: addTask.add[4b076fa4-18c9-4d9e-
9a6d-b0bd6f378e0a]
[2015-05-30 15:31:29,548: INFO/MainProcess] Task addTask.add[4b076fa4-18c9-4d9e-9a6d-b0bd6
f378e0a] succeeded in 0s: 10
[2015-05-30 15:31:42,140: INFO/MainProcess] Received task: addTask.add[fe391d19-a89f-400a-
af21-d7ff79cdd775]
[2015-05-30 15:31:42,144: INFO/MainProcess] Task addTask.add[fe391d19-a89f-400a-af21-d7ff7
9cdd775] succeeded in 0s: 10
```

The result is `10` (you can read it in the last line), as we expected.

How it works...

Let's focus on the first script, `addTask.py`. In the first two lines of code, we create a Celery application instance that uses the RabbitMQ service ad broker:

```
from celery import Celery
app = Celery('addTask', broker='amqp://guest@localhost//')
```

The first argument in the Celery function is the name of the current module (`addTask.py`) and the second argument is the broker keyboard argument, which indicates the URL used to connect the broker (RabbitMQ). Then, we introduce the task. Each task must be added with the annotation (decorator) `@app.task`.

The decorator helps Celery to identify which functions can be scheduled in the task queue. After the decorator, we create the task that the workers can execute. Our first task will be a simple function that performs the sum of two numbers:

```
@app.task
def add(x, y):
    return x + y
```

In the second script, `AddTask_main.py`, we call our task by using the `delay()` method:

```
if __name__ == '__main__':
    result = addTask.add.delay(5,5)
```

Let's remember that this method is a shortcut to the `apply_async()` method, which gives us greater control of the task execution.