

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.

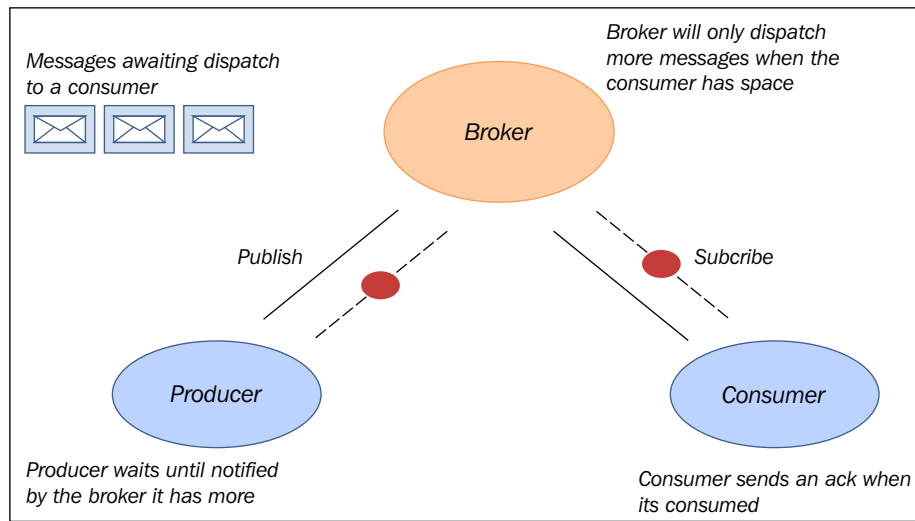
The fundamental problem that arises from the use of this kind of technology is mainly focused on the proper management of traffic (that is devoid of errors both in transmission and reception) of any kind (data, jobs, commands, and so on). Further, a problem stems from a fundamental characteristic of 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.



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:

```

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)
-----
* * * * *
* * * * *
[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:

```

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:

```

[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 15:19:38.276: 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 as 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 keyword 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.

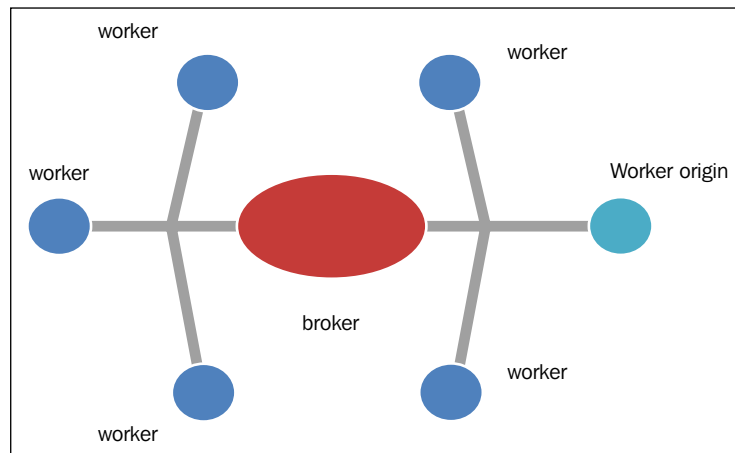
There's more...

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 $\geq 0.6.2$ or `setuptools` ≥ 0.7
- ▶ Greenlet $\geq 0.3.4$
- ▶ `pymq` $\geq 13.1.0$ and `libmq` $\geq 3.2.0$
- ▶ 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:

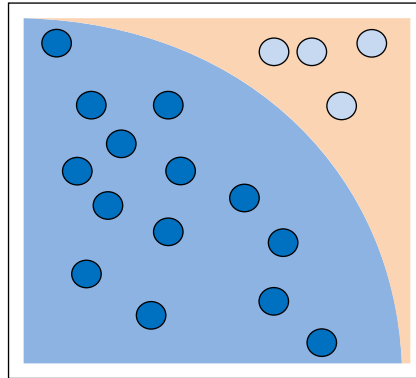
```
C:\Python CookBook\Chapter 5 - Distributed Python\chapter 5 -
codes>python -m scoop pi_calculus_with_montecarlo_method.py
[2015-06-01 15:16:32,685] launcher INFO SCOOP 0.7.2 dev on win32
using Python 3.3.0 (v3.3.0:bd8afb90e
bf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)], API: 1013
[2015-06-01 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 $\pi / 4$, 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 $\pi/4$ 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 `simulateWordload` 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.

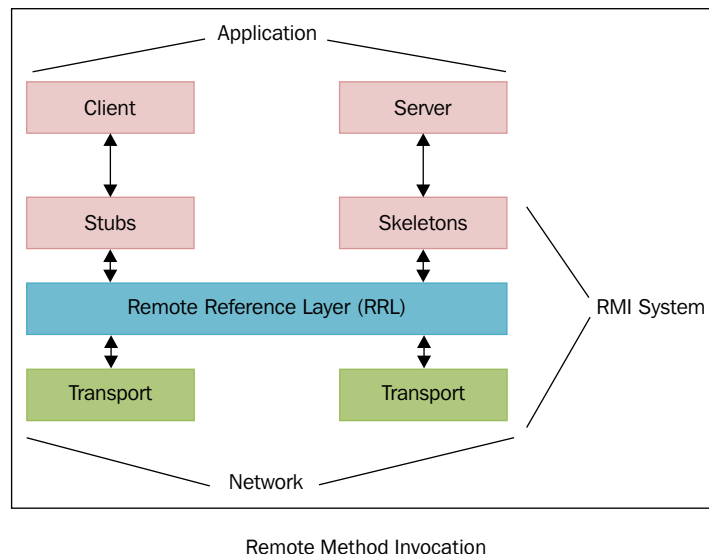
The serial implementation of `mapReduce` is, as follows:

```
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.



As you can see from the preceding figure, Pyro4 allows us to manage and distribute objects in the client-server style. This means that the main parts of a Pyro4 system may switch from a client called to a remote object to an object called to serve a function. It is important to note that during the remote calling, there are always two distinct parts that a client and server accepts and executes the client call. Finally, the entire management of this mechanism is provided by Pyro4 in a distributed way.

Getting ready

The installation procedure is quite simple with the pip installer; from your command shell, type: `pip install pyro4`.

Otherwise, you can download the complete package from <https://github.com/irmen/Pyro4> and install the package with the Python `setup.py` install command from the package directory.

For our examples, we'll use a Python3.3 distro on a Windows machine.

How to do it...

In this example, we'll see how to build and use a simple client-server communication using the Pyro4 middleware. So, we must have two Python scripts.

The code for the server (`server.py`) is:

```
import Pyro4

class Server(object):
    def welcomeMessage(self, name):
        return ("Hi welcome " + str (name))

def startServer():
    server = Server()
    daemon = Pyro4.Daemon()
    ns = Pyro4.locateNS()
    uri = daemon.register(server)
    ns.register("server", uri)
    print("Ready. Object uri =", uri)
    daemon.requestLoop()

if __name__ == "__main__":
    startServer()
```

The code for the client (`client.py`) is as follows:

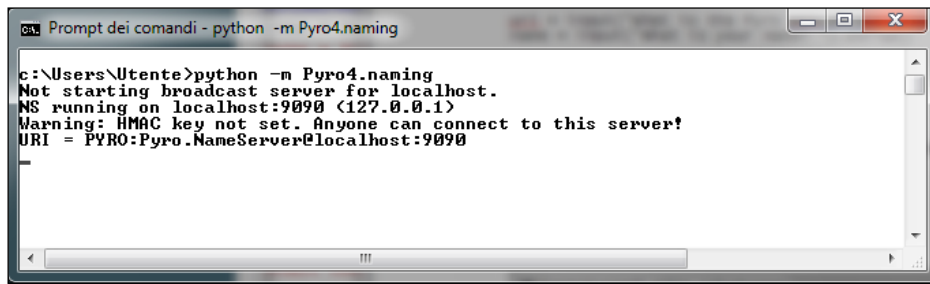
```
import Pyro4

uri = input("What is the Pyro uri of the greeting object? ").strip()
name = input("What is your name? ").strip()
server = Pyro4.Proxy("PYRONAME:server")
print(server.welcomeMessage(name))
```

To run the example, we need a Pyro name server running. To do this, you can type the following command from Command Prompt:

```
python -m Pyro4.naming
```

After this, you'll see the following message:



This means that the name server is running in your network. Then, you can start the server and the client scripts in two separate console windows. To run the server, just type the following:

```
python server.py
```

Now, you'll see something similar to what is shown in the following screenshot:



To run the client, just type:

```
python client.py
```

After this, a message like this will appear:

```
insert the PYRO4 server URI (help : PYRONAME:server)
```

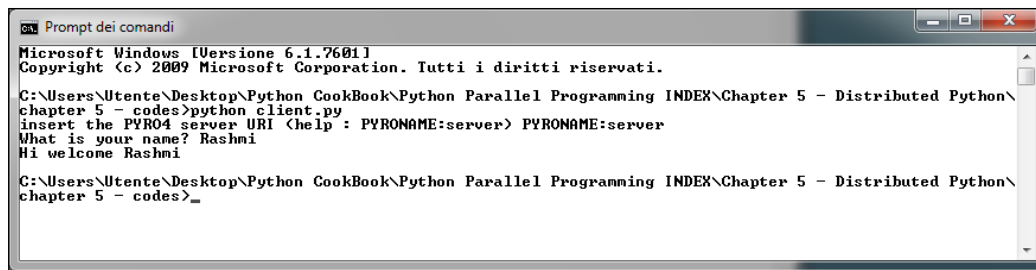
After the correct insertion, you must insert the name of the Pyro4 server, that is, PYRONAME:server:

```
insert the PYRO4 server URI (help : PYRONAME:server) PYRONAME:server
```

You'll see the following message asking you to type your name:

```
What is your name? Rashmi
```

Finally, you'll see a welcome message, Hi welcome Rashmi, as shown in the following screenshot:



```
Prompt dei comandi
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 - Distributed Python\chapter 5 - codes>python client.py
insert the PYRO4 server URI (help : PYRONAME:server) PYRONAME:server
What is your name? Rashmi
Hi welcome Rashmi

C:\Users\Utente\Desktop\Python CookBook\Python Parallel Programming INDEX\Chapter 5 - Distributed Python\chapter 5 - codes>_
```

How it works...

The server contains the object (the `Server` class) that can be accessed remotely. In our example, this object only has the `welcomeMessage()` method that returns a string with the name inserted in the client session:

```
class Server(object):
    def welcomeMessage(self, name):
        return ("Hi welcome " + str(name))
```

To start the server (the `startServer()` function), we must follow some simple steps:

1. Build the instance (named `server`) of the `Server` class: `server = Server()`.

2. Make a Pyro daemon: `daemon = Pyro4.Daemon()`. Pyro4 uses daemon objects to dispatch incoming calls to appropriate objects. A server must create one daemon that manages everything from its instance. Each server has a daemon that knows about all the Pyro objects that the server provides.
3. To execute this script, we have to run a Pyro name server. So, we have to locate this name server that runs: `ns = Pyro4.locateNS()`.
4. Then, we need to register the server as Pyro Object object. It will be known only inside the Pyro daemon: `uri = daemon.register(server)`. It returns the URI for the registered object.
5. Finally, we can register the object server with a name in the name server:
`ns.register("server", uri)`.
6. The function ends with a call to daemon's `eventloop` method. It starts the event loop of the server and waits for calls.

The Pyro4 API enables the developer to distribute objects in a transparent way. Our client scripts send the requests to the server program to execute the `welcomeMessage()` method. The remote call is performed first by creating a Proxy object. In fact, Pyro4 clients use proxy objects to forward method calls to the remote objects and pass results back to the calling code:

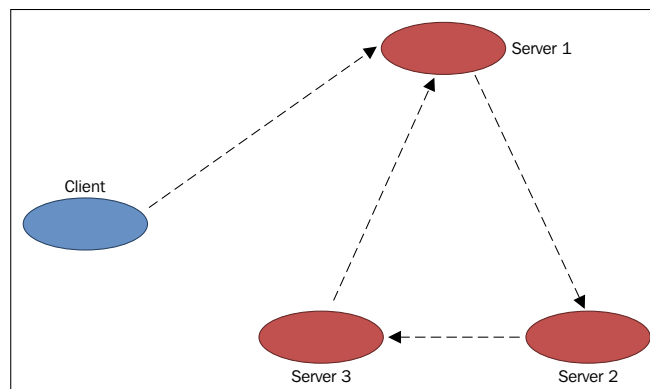
```
server = Pyro4.Proxy("PYRONAME:server")
```

Now, we'll call the server's method that prints a welcome message:

```
print(server.welcomeMessage(name))
```

Chaining objects with Pyro4

In this recipe, we'll show you how to create a chain of objects, which call each other, with Pyro4. Let's suppose that we want to build a distributed architecture like this:



Chaining an object with Pyro4

We have four objects: a client and three servers disposed in a chain topology, as shown in the preceding figure. The client forwards a request to **Server1** and starts the chain call, forwarding the request to **Server2**. Then, it calls the next object in the chain and **Server3**. The chain call ends when **Server3** calls **Server1** again.

The example we're going to show highlights the aspects of the management of remote objects, which can be easily extended to handle more complex distributed architectures.

How to do it...

To implement a chain of objects with Pyro4, we need five Python scripts. The first one is the client (`client.py`). Here is the code for it:

```
from __future__ import print_function
import Pyro4

obj = Pyro4.core.Proxy("PYRONAME:example.chain.A")
print("Result=%s" % obj.process(["hello"]))
```

Each server will be characterized by the parameter `this`, which identifies it in the chain, and the parameter `next`, which defines the next server (that is, subsequent to `this`) in the chain.

For a visualization of the implemented chain you can see the figure associated with this recipe.

```
► server_1.py:
from __future__ import print_function
import Pyro4
import chainTopology

this = "1"
next = "2"

servername = "example.chainTopology." + this

daemon = Pyro4.core.Daemon()
obj = chainTopology.Chain(this, next)
uri = daemon.register(obj)
ns = Pyro4.naming.locateNS()
ns.register(servername, uri)

# enter the service loop.
print("server_%s started " % this)
daemon.requestLoop()
```

```
▶ server_2.py:
from __future__ import print_function
import Pyro4
import chainTopology

this = "2"
next = "3"

servername = "example.chainTopology." + this

daemon = Pyro4.core.Daemon()
obj = chain.chainTopology(this, next)
uri = daemon.register(obj)
ns = Pyro4.naming.locateNS()
ns.register(servername, uri)

# enter the service loop.
print("server_%s started " % this)
daemon.requestLoop()

▶ server_3.py:

from __future__ import print_function
import Pyro4
import chainTopology

this = "3"
next = "1"

servername = "example.chainTopology." + this

daemon = Pyro4.core.Daemon()
obj = chain.chainTopology(this, next)
uri = daemon.register(obj)
ns = Pyro4.naming.locateNS()
ns.register(servername, uri)

# enter the service loop.
print("server_%s started " % this)
daemon.requestLoop()
```

The last script is the chain object, as shown in the following code:

```
► chainTopology.py:

from __future__ import print_function
import Pyro4

class Chain(object):
    def __init__(self, name, next):
        self.name = name
        self.nextName = next
        self.next = None

    def process(self, message):
        if self.next is None:
            self.next = Pyro4.core.Proxy("PYRONAME:example.chain."
\
                                         + self.nextName)

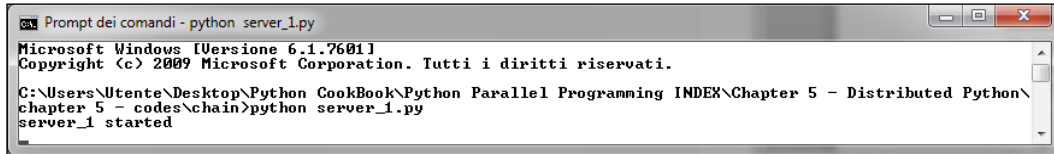
        if self.name in message:
            print("Back at %s; the chain is closed!" % self.name)
            return ["complete at " + self.name]
        else:
            print("%s forwarding the message to the object %s" \
                  % (self.name, self.nextName))
            message.append(self.name)
            result = self.next.process(message)
            result.insert(0, "passed on from " + self.name)
            return result
```

To execute this example, start by running the Pyro4 name server:

```
C:>python -m Pyro4.naming
Not starting broadcast server for localhost.
NS running on localhost:9090 (127.0.0.1)
Warning: HMAC key not set. Anyone can connect to this server!
URI = PYRO:Pyro.NameServer@localhost:9090
```

Then, run the three servers. In three separate Command Prompts, type the `python server_name.py` command.

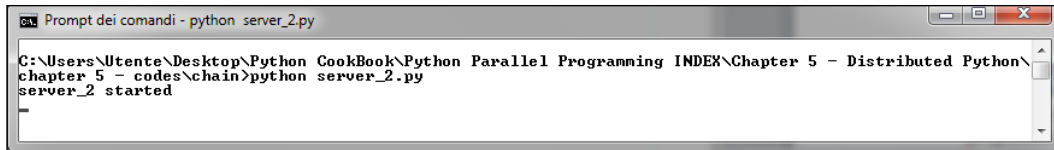
A message like this should appear after this for `server_1`:



```
Prompt dei comandi - python server_1.py
Microsoft Windows [Versione 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tutti i diritti riservati.

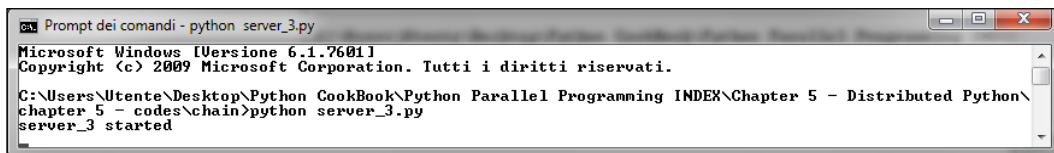
C:\Users\Utente\Desktop\Python CookBook\Python Parallel Programming INDEX\Chapter 5 - Distributed Python\
chapter 5 - codes\chain>python server_1.py
server_1 started
```

For `server_2`, something similar to what is shown in the following screenshot will appear:



```
Prompt dei comandi - python server_2.py
C:\Users\Utente\Desktop\Python CookBook\Python Parallel Programming INDEX\Chapter 5 - Distributed Python\
chapter 5 - codes\chain>python server_2.py
server_2 started
```

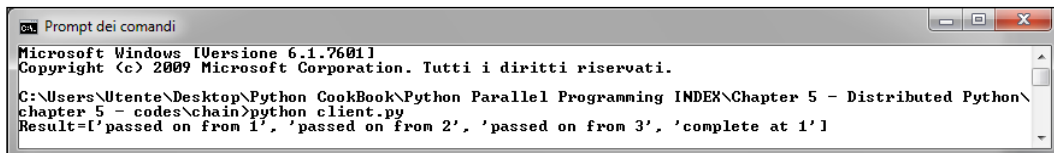
A message similar to what is shown in the following screenshot should appear for `server_3`:



```
Prompt dei comandi - python server_3.py
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 - Distributed Python\
chapter 5 - codes\chain>python server_3.py
server_3 started
```

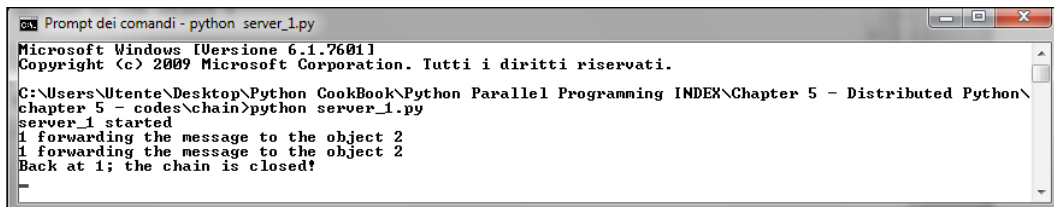
Finally, you must run the `client.py` script from another command shell:



```
Prompt dei comandi
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 - Distributed Python\
chapter 5 - codes\chain>python client.py
Result='passed on from 1', 'passed on from 2', 'passed on from 3', 'complete at 1'
```

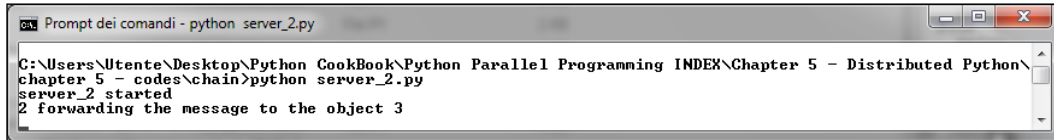
The preceding message shows as a result the forwarding request passed across the three servers, when it comes back to `server_1` the task is completed. Also, here, we can focus on the behavior of the object servers when the request is forwarded to the next object in the chain. To see what happens next, refer to the message below the start message in the following screenshot for `server_1`:



```
Prompt dei comandi - python server_1.py
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 - Distributed Python\
chapter 5 - codes\chain>python server_1.py
server_1 started
1 forwarding the message to the object 2
1 forwarding the message to the object 2
Back at 1; the chain is closed!
```

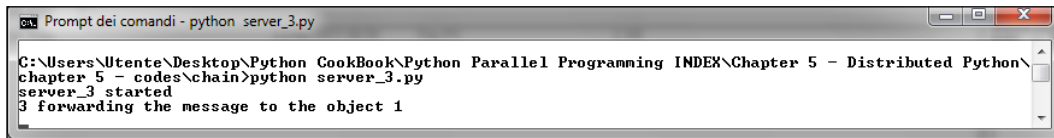
The result of `server_2` is as follows:



```
Prompt dei comandi - python_server_2.py

C:\Users\Utente\Desktop\Python CookBook\Python Parallel Programming INDEX\Chapter 5 - Distributed Python\chapter 5 - codes\chain>python server_2.py
server_2 started
2 forwarding the message to the object 3
```

The result of `server_3` is as follows:



```
Prompt dei comandi - python_server_3.py

C:\Users\Utente\Desktop\Python CookBook\Python Parallel Programming INDEX\Chapter 5 - Distributed Python\chapter 5 - codes\chain>python server_3.py
server_3 started
3 forwarding the message to the object 1
```

How it works...

The core of this example is the `Chain` class that we defined in the `chainTopology.py` script. It allows communication between the three servers. In fact, each server calls the class to find out which is the next element in the chain (refer to the method `process` in `chainTopology.py`). Also, it executes the call with the `Pyro4.core.proxy` statement:

```
if self.next is None:
    self.next = Pyro4.core.Proxy("PYRONAME:example.
chainTopology." + self.nextName)
```

If the chain is closed (the last call is done from `server_3` to `server_1`), a closing message is printed out:

```
if self.name in message:
    print("Back at %s; the chain is closed!" % self.name)
    return ["complete at " + self.name]
```

A forwarding message is printed out if there is a next element in the chain:

```
print("%s forwarding the message to the object %s" % (self.name, self.
nextName))

message.append(self.name)
result = self.next.process(message)
result.insert(0, "passed on from " + self.name)
return result
```

The code for the server is the same and only differs on the definition of the current element and the next element of the chain, for example, this is the definition for the first server (`server_1`):

```
this = "1"
next = "2"
```

The remaining lines of the following code define, in the same manner as the previous example, the communication with the next element in the chain:

```
servername = "example.chainTopology." + this
daemon = Pyro4.core.Daemon()
obj = chainTopology.Chain(this, next)
uri = daemon.register(obj)
ns = Pyro4.naming.locateNS()
ns.register(servername, uri)
daemon.requestLoop()
```

Finally, in the client script, we start the process by calling the first element (`server 1`) in the chain:

```
obj = Pyro4.core.Proxy("PYRONAME:example.chainTopology.1")
```

Developing a client-server application with Pyro4

In this recipe, we'll show you how to build a simple client-server application with Pyro4. The application that we'll show here is not complete, but is equipped with all the methods that will successfully complete and improve it.

A client-server application indicates a network architecture in which, generally, a client computer or terminal connects to a server for the use of a certain service, such as the sharing of a certain resource hardware/software with other clients and relying on the underlying protocol architecture. In our system, the server manages an online shopping site, while the clients manage the customers that are registered on this site and connect to it for shopping.

How to do it...

For the sake of simplicity, we have three scripts. The first one represents the object `client` in which we have customer management, the second script is the object `shop`, and the third script is the object `server`.

For the server (`server.py`), the code is as follows:

```
#
#   The Shops server
#

from __future__ import print_function
import Pyro4
import shop

ns = Pyro4.naming.locateNS()
daemon = Pyro4.core.Daemon()
uri = daemon.register(shop.Shop())
ns.register("example.shop.Shop", uri)
print(list(ns.list(prefix="example.shop.").keys()))
daemon.requestLoop()
```

The code for the client (`client.py`) is as follows:

```
from __future__ import print_function
import sys
import Pyro4

# A Shop client.
class client(object):
    def __init__(self, name , cash):
        self.name = name
        self.cash = cash
    def doShopping_deposit_cash(self, Shop):
        print("\n*** %s is doing shopping with %s:"\
              % (self.name, Shop.name()))
        print("Log on")
        Shop.logOn(self.name)
        print("Deposit money %s" %self.cash)
        Shop.deposit(self.name, self.cash)
        print("balance=%.2f" % Shop.balance(self.name))
        print("Deposit money %s" %self.cash)
        Shop.deposit(self.name, 50)
        print("balance=%.2f" % Shop.balance(self.name))
        print("Log out")
        Shop.logOut(self.name)

    def doShopping_buying_a_book(self, Shop):
        print("\n*** %s is doing shopping with %s:"\
```

```

        % (self.name, Shop.name()))
    print("Log on")
    Shop.logOn(self.name)
    print("Deposit money %s" %self.cash)
    Shop.deposit(self.name, self.cash)
    print("balance=%.2f" % Shop.balance(self.name))
    print ("%s is buying a book for %s$\\"
           %(self.name,37))
    Shop.buy(self.name,37)
    print("Log out")
    Shop.logOut(self.name)

if __name__ == '__main__':
    ns = Pyro4.naming.locateNS()
    uri = ns.lookup("example.shop.Shop")
    print(uri)
    Shop = Pyro4.core.Proxy(uri)
    meeta = client('Meeta',50)
    rashmi = client('Rashmi',100)
    rashmi.doShopping_buying_a_book(Shop)
    meeta.doShopping_deposit_cash(Shop)
    print("")
    print("")
    print("")
    print("")

    print("The accounts in the %s:" % Shop.name())
    accounts = Shop.allAccounts()
    for name in accounts.keys():
        print("  %s : %.2f\\"
              % (name, accounts[name]))

```

This is the code for the object shop (shop.py):

```

class Account(object):
    def __init__(self):
        self._balance = 0.0

    def pay(self, price):
        self._balance -= price

    def deposit(self, cash):
        self._balance += cash

```

```
def balance(self):
    return self._balance

class Shop(object):
    def __init__(self):
        self.accounts = {}
        self.clients = ['Meeta', 'Rashmi', 'John', 'Ken']

    def name(self):
        return 'BuyAnythingOnline'

    def logOn(self, name):
        if name in self.clients :
            self.accounts[name] = Account()
        else :
            self.clients.append(name)
            self.accounts[name] = Account()

    def logOut(self, name):
        print('logout %s' %name)

    def deposit(self, name, amount):
        try:
            return self.accounts[name].deposit(amount)
        except KeyError:
            raise KeyError('unknown account')

    def balance(self, name):
        try:
            return self.accounts[name].balance()
        except KeyError:
            raise KeyError('unknown account')

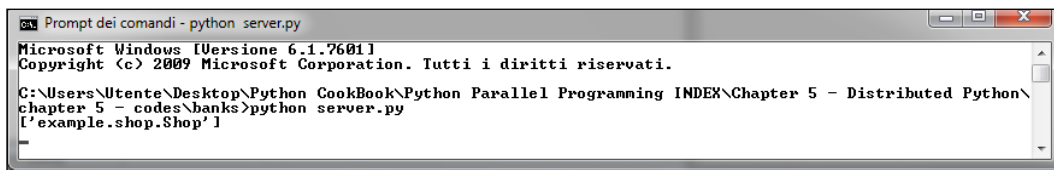
    def allAccounts(self):
        accs = {}
        for name in self.accounts.keys():
            accs[name] = self.accounts[name].balance()
        return accs

    def buy(self, name, price):
        balance = self.accounts[name].balance()
        self.accounts[name].pay(price)
```

To execute the code, you must first enable the Pyro4 name sever:

```
C:>python -m Pyro4.naming
Not starting broadcast server for localhost.
NS running on localhost:9090 (127.0.0.1)
Warning: HMAC key not set. Anyone can connect to this server!
URI = PYRO:Pyro.NameServer@localhost:9090
```

Then, start the server by using the `python server.py` command. A shell like the one shown in the following screenshot will appear when you do this:



Finally, you should start the customer simulation with the following command:

```
python client.py
```

The following text will be printed out with the use of the following command:

```
C:\Users\Utente\Desktop\Python CookBook\Python Parallel Programming
INDEX\Chapter 5 - Distributed Python\
chapter 5 - codes\banks>python client.py
PYRO:obj_8c4a5b4ae7554c2c9feee5b0113902e0@localhost:59225
```

```
*** Rashmi is doing shopping with BuyAnythingOnline:
```

```
Log on
Deposit money 100
balance=100.00
Rashmi is buying a book for 37$
Log out
```

```
*** Meeta is doing shopping with BuyAnythingOnline:
```

```
Log on
Deposit money 50
balance=50.00
```

```
Deposit money 50
balance=100.00
Log out
```

The accounts in the BuyAnythingOnline:

```
Meeta : 100.00
Rashmi : 63.00
```

This output shows a simple session for two customers, Meeta and Rashmi.

How it works...

The server side of the application must locate the `Shop()` object, calling the statement:

```
ns = Pyro4.naming.locateNS()
```

Then, it must enable a communication channel:

```
daemon = Pyro4.core.Daemon()
uri = daemon.register(shop.Shop())
ns.register("example.shop.Shop", uri)
daemon.requestLoop()
```

The `shop.py` script contains classes for account and shop management. The `shop` class manages each account. It provides methods to log in and log out, manage customer's money, and to buy items:

```
class Shop(object):

    def logOn(self, name):
        if name in self.clients :
            self.accounts[name] = Account()
        else :
            self.clients.append(name)
            self.accounts[name] = Account()

    def logOut(self, name):
        print('logout %s' %name)

    def deposit(self, name, amount):
        try:
            return self.accounts[name].deposit(amount)
        except KeyError:
            raise KeyError('unknown account')
```

```

def balance(self, name):
    try:
        return self.accounts[name].balance()
    except KeyError:
        raise KeyError('unknown account')

def buy(self, name, price):
    balance = self.accounts[name].balance()
    self.accounts[name].pay(price)

```

Each customer has their own `Account` object that provides methods for customer deposit management:

```

class Account(object):
    def __init__(self):
        self._balance = 0.0

    def pay(self, price):
        self._balance -= price

    def deposit(self, cash):
        self._balance += cash

    def balance(self):
        return self._balance

```

Finally, the `client.py` script contains the class that is used to start the simulation. In the main program, we instantiate two customers, Rashmi and Meeta:

```

meeta = client('Meeta', 50)
rashmi = client('Rashmi', 100)
rashmi.doShopping_buying_a_book(Shop)
meeta.doShopping_deposit_cash(Shop)

```

They deposit some cash end on the site and then start with their shopping as shown:

- Rashmi buys a book:

```

def doShopping_buying_a_book(self, Shop):
    Shop.logOn(self.name)
    Shop.deposit(self.name, self.cash)
    Shop.buy(self.name, 37)
    Shop.logOut(self.name)

```

- ▶ Meeta twice deposits \$100 in her account:

```
def doShopping_deposit_cash(self, Shop):
    Shop.logOn(self.name)
    Shop.deposit(self.name, self.cash)
    Shop.deposit(self.name, 50)
    Shop.logOut(self.name)
```

- ▶ At the end of the simulation, the main program reports the count's deposit of Meeta and Rashmi:

```
print("The accounts in the %s:" % Shop.name())
accounts = Shop.allAccounts()
for name in accounts.keys():
    print("  %s : %.2f" \
          % (name, accounts[name]))
```

Communicating sequential processes with PyCSP

PyCSP is a Python module based on communicating sequential processes, which is a programming paradigm developed to build concurrent programs via message passing. The PyCSP module is characterized by:

- ▶ The exchange of messages between processes
- ▶ The possibility of using a thread to use shared memory
- ▶ The exchange of messages is done through channels

The channels allow:

- ▶ An exchange of values between processes
- ▶ The synchronization of processes

PyCSP allows the use of different channel types: One2One, One2Any, Any2One, and Any2One. These names indicate the number of writers and readers that can communicate over the channel.

Getting ready

PyCSP can be installed using the `pip` installer via the following command:

```
pip install python-csp
```

Also, it is possible to download the entire distribution from GitHub (<https://github.com/futurecore/python-csp>).

Download it and then type the following from the installation directory:

```
python setup.py install
```

For our examples, we will use the Python Version 2.7

How to do it...

In this first example, we want to introduce the basic concepts of PyCSP, the processes, and channels. So, we have defined two processes named counter and printer. We now want to see how to define the communication between these processes:

Let's consider the following code:

```
from pycsp.parallel import *

@process
def processCounter(cout, limit):
    for i in xrange(limit):
        cout(i)
    poison(cout)

@process
def processPrinter(cin):
    while True:
        print cin(),

A = Channel('A')
Parallel(
    processCounter(A.writer(), limit=5),
    processPrinter(A.reader())
)

shutdown()
```

To execute this code, simply press the run button on the Python2.7 IDLE. An output like this will be shown after this:

```
Python 2.7.9 (default, Dec 10 2014, 12:28:03) [MSC v.1500 64 bit (AMD64)]
on win32

Type "copyright", "credits" or "license()" for more information.

>>> =====RESTART=====
>>>
0 1 2 3 4
>>>
```

How it works...

In this example, we used the functions defined in the `pycsp.parallel` module:

```
from pycsp.parallel import *
```

This module has the `Any2Any` channel type, which allows multiple processes, which are attached to the ends of the channels, to communicate through it. To create the channel A, we use the following statement:

```
A = Channel('A')
```

This new channel is automatically hosted in the current Python interpreter. For each Python interpreter that imports the `pycsp.parallel` module, only a port that handles all the channels started in the Python interpreter will be listed. However, this module does not provide a name server available for the channels. So to connect to a hosted channel, you must know the right location.

For example, to connect the channel B to the localhost port 8888, we input the following code:

```
A = pycsp.Channel('B', connect=('localhost', 8888))
```

In PyCSP, we have three basic ways to manage a channel:

- ▶ `channel.Disconnect()`: This allows the Python interpreter to quit. It is used in a client-server setting, where a client wants to be Disconnected after it receives a reply from a server.
- ▶ `channel.reader()`: This creates and returns the reader end of the channel.
- ▶ `channel.writer()`: This creates and returns the writer end of the channel.

To indicate a process, we use the `@process` decorator. In PyCSP, each generated CSP process is implemented as a single OS thread. In this example, we have two processes: a counter and a printer. The process counter has two arguments: `cout` to redirect its output and `limit`, which defines the total number of items to be printed:

```
@process
def counter(cout, limit):
    for i in xrange(limit):
        cout(i)
    poison(cout)
```

The poison statement, `poison(cout)`, means that the channel end is poisoned. This means that all subsequent reads and writes on this channel will throw an exception that can be used to end the current procedure or Disconnect the channel. We also note that the poisoning may cause a race condition if there are multiple concurrent procedures.

The process printer only has one argument, which is the item to print, defined in the `cin` variable:

```
@process
def printer(cin):
    while True:
        print cin(),
```

The core of the script is in the following line of code:

```
A = Channel('A')
```

This defines the A channel, which permits communication between the two processes.

Finally, the `Parallel` statement is as follows:

```
Parallel(
    counter(A.writer(), limit=10),
    printer(A.reader())
)
```

This starts all the processes and blocks them only if the counter and process have terminated communication with each other. This statement represents the basic idea of CSP: concurrent processes synchronize with each other by synchronizing their I/O through the channel A. The way to do this is to allow I/O to occur only when a process counter states that it is ready to output to a process printer specifically and the process printer states that it is ready to input from a process counter. If one of these happens without the other being true, the process is put in a wait queue until the other process is ready.

Each PyCSP application creates a server thread to manage the incoming communication over the channels. So, it is always necessary to terminate each PyCSP application with a call to the `shutdown()` method:

```
shutdown()
```

PyCSP provides two methods to trace its execution:

- ▶ `TraceInit(<filename>, stdout=<True | False>)`: This is used to start the trace
- ▶ `TraceQuit()`: This is used to stop the trace

These must be placed in the following schema:

```
from pycsp.common.trace import *

TraceInit("trace.log")

"""
    PROCESSES TO BE TRACED
"""

TraceQuit()
shutdown()
```

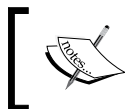
For our example, we have built the log trace (with a limit count equal to three):

```
{'chan_name': 'A', 'type': 'Channel'}
{'chan_name': 'A', 'type': 'ChannelEndWrite'}
{'chan_name': 'A', 'type': 'ChannelEndRead'}
{'processes': [{'func_name': 'processCounter', 'process_id':
'9cb4b3720ed111e5bb4c0024813d643d.processCounter'}, {'func_name':
'processPrinter', 'process_id': '9cb63a0f0ed111e5993a0024813d643d.
processPrinter'}], 'process_id': '9c42428f0ed111e59ba10024813d643d.__
INIT__', 'type': 'BlockOnParallel'}
{'func_name': 'processCounter', 'process_id':
'9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'type':
'StartProcess'}
{'func_name': 'processPrinter', 'process_id':
'9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'type':
'StartProcess'}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'BlockOnWrite', 'id': 0}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'BlockOnRead', 'id': 0}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'DoneRead', 'id': 0}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'DoneWrite', 'id': 0}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'BlockOnWrite', 'id': 1}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'BlockOnRead', 'id': 1}
```

```
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'DoneRead', 'id': 1}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'DoneWrite', 'id': 1}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'BlockOnWrite', 'id': 2}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'BlockOnRead', 'id': 2}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'DoneRead', 'id': 2}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'DoneWrite', 'id': 2}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'Poison', 'id': 3}
{'func_name': 'processCounter', 'process_id':
'9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'type': 'QuitProcess'}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'BlockOnRead', 'id': 3}
{'func_name': 'processPrinter', 'process_id':
'9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'type': 'QuitProcess'}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'Poison', 'id': 3}
{'processes': [{'func_name': 'processCounter', 'process_id':
'9cb4b3720ed111e5bb4c0024813d643d.processCounter'}, {'func_name':
'processPrinter', 'process_id': '9cb63a0f0ed111e5993a0024813d643d.
processPrinter'}], 'process_id': '9c42428f0ed111e59ba10024813d643d.__
INIT__', 'type': 'DoneParallel'}
{'type': 'TraceQuit'}
```

There's more...

CSP is a formal language used to describe the interactions of concurrent processes. It falls under the mathematical theory of competition, which is known as algebra processes. It has been used in practice as a tool for the specification and verification of the competition aspects of a wide variety of systems. The rules of the CSP-inspired programming language Occam are now widely used as a parallel programming language.



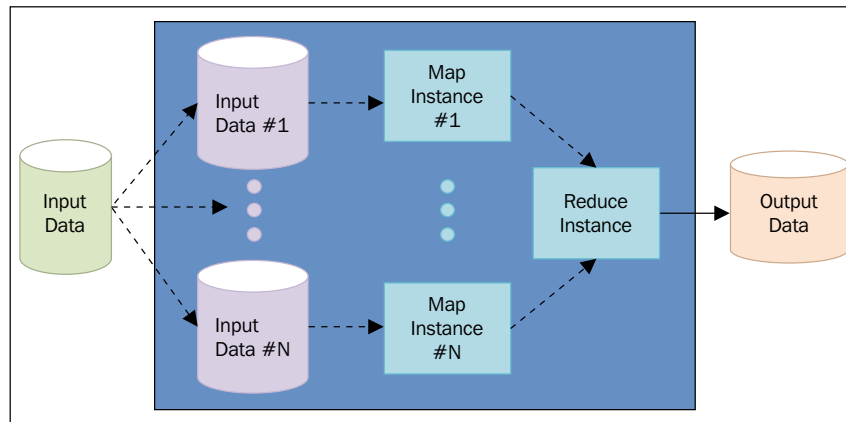
For those of you who are interested in CSP's theory, we suggest you go through Hoare's original book, which is available online at <http://www.usingcsp.com/cspbook.pdf>.

Using MapReduce with Disco

Disco is a Python module based on the MapReduce framework introduced by Google, which allows the management of large distributed data in computer clusters. The applications written using Disco can be performed in the economic cluster of machines with a very short learning curve. In fact, the technical difficulties related to the processes that are distributed as load balancing, job scheduling, and the communications protocol are completely managed by Disco and hidden from the developer.

The typical applications of this module are essentially as follows:

- ▶ Web indexing
- ▶ URL access counter
- ▶ Distributed sort



The MapReduce schema

The MapReduce algorithm implemented in Disco is as follows:

- ▶ **Map:** The master node takes the input data, breaks it into smaller subtasks, and distributes the work to the slave nodes. The single map node produces the intermediate result of the `map()` function in the form of pairs `[key, value]` stored on a distributed file whose location is given to the master at the end of this step.
- ▶ **Reduce:** The master node collects the results. It combines the pairs `[key, value]` in the lists of values that share the same key and sorts them for the key (lexicographical and increasing or user-defined). The pairs of the form `[key, IteratorList (value, value, ...)]` are passed to the nodes that run the reducer function `reduce()`.

Moreover, the output data that is stored on files, can be the input for a new map and reduce procedure, allowing, in this way, to concatenate more MapReduce jobs.

Getting ready



The Disco module is available at <https://github.com/Discoproject/Disco>.

You need a Linux/Unix distribution to install it.

The following are the prerequisites (on each server):

- ▶ The SSH daemon and client
- ▶ Erlang/OTP R14A or newer version
- ▶ Python 2.6.6 or newer version, Python 3.2 or newer version

Finally, to install Disco, type the following lines:

```
git clone git://github.com/Discoproject/Disco.git $Disco_HOME
cd $Disco_HOME
make
cd lib && python setup.py install --user && cd ..
bin/Disco nodaemon
```

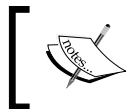
The next step is to enable a password-less login for all servers in the Disco clusters. For a single machine installation, you must run the following command:

```
ssh-keygen -N '' -f ~/.ssh/id_dsa
```

Then, type the following:

```
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Now if you try to log in to all servers in the cluster or localhost, you will not need to give a password nor answer any questions after the first log in attempt.



For any questions about Disco's installation, refer to <http://Disco.readthedocs.org/en/latest/intro.html>.

In the next example, we have used a Python 2.7 distro on a Linux machine.

How to do it...

In the following example, we examine a typical MapReduce problem using the Disco module. Given a text, we must count all the occurrences of some words in the text:

```
from Disco.core import Job, result_iterator

def map(line, params):
    import string
    for word in line.split():
        strippedWord = word.translate\
            (string.maketrans("", ""), string.punctuation)
        yield strippedWord, 1

def reduce(iter, params):
    from Disco.util import kvgroup
    for word, counts in kvgroup(sorted(iter)):
        yield word, sum(counts)

if __name__ == '__main__':
    job = Job().run(input="There are known knowns.\
        These are things we know that we know.\
        There are known unknowns. \
        That is to say,\
        there are things that \
        we know we do not know.\
        But there are also unknown unknowns.\
        There are things \
        we do not know we do not know",
        map=map,
        reduce=reduce)

    sort_in_numerical_order = \
        open('SortNumerical.txt', 'w')
    sort_in_alphabetically_order = \
        open('SortAlphabetical.txt', 'w')

    wordCount = []
    for word, count in \
        result_iterator(job.wait(show=True)):
        sort_in_alphabetically_order.write('%s \t %d\n' %
            (str(word), int(count)) )
```

```

wordCount.append((word,count))

sortedWordCount =sorted(wordCount, \
                        key=lambda count: count[1],\
                        reverse=True)

for word, count in sortedWordCount:
    sort_in_numerical_order.write('%s \t %d\n'\
                                % (str(word), int(count)) )

sort_in_alphabetically_order.close()
sort_in_numerical_order.close()

```

After running the script, we have the two resulting files that we've reported in the following table:

Sortnumerical.txt		SortAlphabetical.txt	
6	are	also	1
6	know	are	6
6	we	but	1
5	there	do	3
3	do	is	1
3	not	know	6
3	that	known	2
3	things	knowns	1
2	known	not	3
2	unknowns	say	1
1	also	to	1
1	but	that	3
1	is	there	5
1	knowns	these	1
1	say	things	3
1	to	unknown	1
1	these	unknowns	2
1	unknown	we	6

How it works...

The core of this example are the `map` and `reduce` functions. The `map` function Disco has two arguments line that represent the sentence to be analyzed. However, `params` will be ignored in this example.

Here, the sentence is split in to one or more words, the punctuation symbols are ignored, and all words are converted to lowercase:

```
def map(line, params):
    import string
    for word in line.split():
        strippedWord = word.translate\
            (string.maketrans("", ""), string.punctuation)
        yield strippedWord, 1
```

The result of a `map` function on a line of text is a series of tuples in the form of a key and a value. For example, the sentence "There are known knowns" takes on this form:

```
[("There", 1), ("are", 1), ("known", 1), ("knowns",1)]
```

Let's remember that the MapReduce framework manipulates enormous datasets that are larger than the common memory space in a single machine, so the keyword yield at the end of the `map` function allows Disco to manage datasets in a smarter way. The `reduce` function operates on two arguments, `iter`, that are iterable objects (it acts like a list data structure), while the `params` argument linked in the `map` function is ignored in this example.

Each iterable object is sorted into alphabetical order using the Python function `sorted`:

```
def reduce(iter, params):
    from Disco.util import kvgroup
    for word, counts in kvgroup(sorted(iter)):
        yield word, sum(counts)
```

On the sorted list, we apply Disco's function `kvgroup`. It groups values of consecutive keys, which are compared to be equal. Finally, the occurrence of each word in the text is obtained through the Python function `sum`.

In the main part, we use Disco's `job` function to execute the `mapReduce` function:

```
job = Job().run(input="There are known knowns.\
These are things we know that we know.\
There are known unknowns. \
That is to say,\
there are things that \
we know we do not know.\
But there are also unknown unknowns.\
```

```

        There are things \
        we do not know we do not know",
    map=map,
    reduce=reduce)

```

Finally, the results are ordered into numerical and alphabetical order and they are printed in two output files:

```

sort_in_numerical_order = open('SortNumerical.txt', 'w')

sort_in_alphabetically_order = open('SortAlphabetical.txt', 'w')

```

There's more...

Disco is a very powerful framework that is rich with different functionalities. A full discussion of this module is beyond the scope of this book.



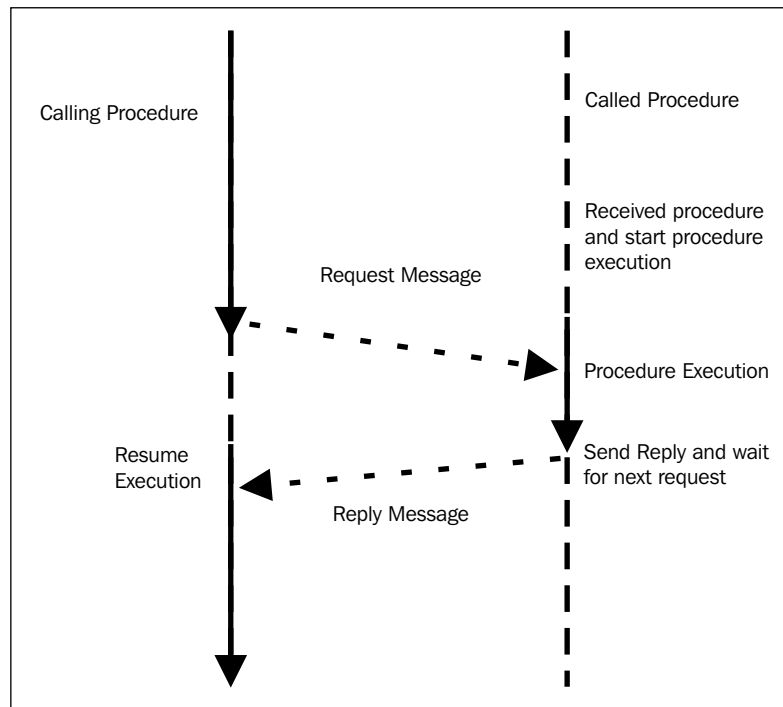
To get a complete introduction, refer to <http://Discoproject.org/>.

A remote procedure call with RPyC

Remote Python Call (RPyC) is a Python module that is used for remote procedure calls as well as for distributed computing. The idea at the base of RPC is to provide a mechanism to transfer control from a program (client) to another (server), similar to what happens with the invocation of a subroutine in a centralized program. The advantages of this approach are that it has very simple semantics and knowledge and familiarity of the centralized mechanism of a function call. In a procedure invocation, the client process is suspended until the process server has performed the necessary computations and has given the results of computations. The effectiveness of this method is due to the fact that the client-server communication takes the form of a procedure call instead of invocations to the transport layer so that all the details of the operation of the network are hidden from the application program by placing them in local procedures called stubs. The main features of RPyC are:

- ▶ In syntactic transparency a remote procedure call can have the same syntax as a local call
- ▶ In semantic transparency a remote procedure call is semantically equivalent to the local one
- ▶ Handling synchronous and asynchronous communication

- Symmetric communication protocol means that both the client and server can serve a request



The remote procedure call model

Getting ready

The installation procedure is quite simple with the `pip` installer. From your command shell, type the following:

```
pip install rpyc
```

Otherwise, you can go to <https://github.com/tomerfiliba/rpyc> and download the complete package (it is a `.zip` file). Finally, to install `rpyc`, you must type the command:
`Python setup.py install` from the package directory.

After the installation, you can just explore this library. In our examples, we will run a client and server on the same machine, `localhost`. Running a server with `rpyc` is very simple: go to the directory `../rpyc-master/bin` of the `rpyc` package directory and then execute the file `rpyc_classic.py`:

```
C:\Python Cookbook\ Chapter 5- Distributed Python\rpyc-master\bin>python rpyc_classic.py
```

After running this script, you'll read on Command Prompt the following output message:

```
INFO:SLAVE/18812:server started on [0.0.0.0]:18812
```

How to do it...

We are now ready for the first example that shows you how to redirect `stdout` of a remote process:

```
import rpyc
import sys
c = rpyc.classic.connect("localhost")
c.execute("print ('hi python cookbook')")
c.modules.sys.stdout = sys.stdout
c.execute("print ('hi here')")
```

By running this script, you'll see the redirected output in the server side:

```
C:\Python Cookbook\Chapter 5- Distributed Python\rpyc-master\bin>python
rpyc_classic.py
INFO:SLAVE/18812:server started on [0.0.0.0]:18812
INFO:SLAVE/18812:accepted 127.0.0.1:6279
INFO:SLAVE/18812:welcome [127.0.0.1]:6279
hi python cookbook
```

How it works...

The first step is to run a client that connects to the server:

```
import rpyc

c = rpyc.classic.connect("localhost")
```

Here, the client-side statement `rpyc.classic.connect (host, port)` creates a socket connection to the given host and port. Sockets define the endpoint of a connection. `rpyc` uses sockets to communicate with other programs, which may be distributed on different computers.

Next, we have the following statement:

```
c.execute("print ('hi python cookbook')")
```

This executes the print statement on the server (a remote `exec` statement).