```
# -*- coding: iso-latin-1 -*-

# Esempio di "calcolo distribuito". Riadattato da:
# http://eli.thegreenplace.net/2012/01/24/
# /distributed-computing-in-python-with-multiprocessing

# Questo programma calcola i fattori di un certo numero di interi
# distribuendo il calcolo su più macchine.
#
# La comunicazione tra le varie macchine si ottiene grazie ad alcuni
# "manager" che sono in grado di fornire dei "proxy" che danno accesso
# ad una risorsa condivisa (in questo caso dei dizionari).

# Il programma lancia un server se eseguito con l'opzione "-s", un
# client se eseguito con l'opzione "-c".  Il server crea due code che
# contengono rispettivamente delle liste con i numeri da fattorizzare
# (job_q) e dei dizionari con i numeri fattorizzati come chiavi e i
# relativi fattori come valori (res_q).  I client "pescano" le liste
# da fattorizzare da job_q e inseriscono i loro risultati in res_q.

import os
import time
import Queue
import multiprocessing as mp
import argparse
from multiprocessing.managers import SyncManager
from factorize import factorize_naive

def factorizer_worker(job_q, res_q):
    # Prendo da JOB_Q i numeri  da fattorizzare, ne calcolo i fattori
    # con factorize_naive ed inserisco il risultato (un dizionario)
    # nella coda RES_Q. Questa è una semplice funzione che non ha
    # nulla a che vedere con multi processi.
    while True:
        try:
            job = job_q.get_nowait()
            out_dict = {n: factorize_naive(n) for n in job}
            res_q.put(out_dict)
        except Queue.Empty:
            return

def mp_factorizer(job_q, res_q, proc_count):
    # Genero (lancio ed aspetto) PROC_COUNT processi che eseguono
    # FACTORIZER_WORKER e che usano tutti le due stesse code
    # (magicamente sincronizzate) da cui ottenere i numeri da
    # fattorizzare (JOB_Q) e in cui inserire i risultati del calcolo
    # (RES_Q).
    pp = [mp.Process(target=factorizer_worker,
                     args=(job_q, res_q))
        for i in range(proc_count)]

    for p in pp: p.start()
    for p in pp: p.join()

def make_server_manager(port, authkey):
    # Il manager è un processo che ascolta su una certa porta (PORT)
    # ed accetta delle connessioni dai client (con chiave di
    # autorizzazione AUTHKEY).  Questi client possono eseguire due
    # metodi "registrati" dal server ed ottenere così le due code
    # JOB_Q e RES_Q di cui hanno bisogno per lavorare.  In realtà non
    # ottengono direttamente i due oggetti Queue, ma dei proxy
    # sincronizzati!

    job_q = Queue.Queue()
    res_q = Queue.Queue()

    class JobQueueManager(SyncManager):
        pass
```

```
    # "lambda: job_q" è una funzione anonima senza argomenti che
    # restituisce sempre job_q.
    JobQueueManager.register('get_job_q', callable=lambda: job_q)
    JobQueueManager.register('get_res_q', callable=lambda: res_q)

    return JobQueueManager(address=('', port), authkey=authkey)

def runserver(port, authkey, base, count):
    man = make_server_manager(port, authkey)
    man.start()
    print ("Server process started. pid: %d port: %s key: '%s'." % (
        os.getpid(), port, authkey))
    job_q = man.get_job_q()
    res_q = man.get_res_q()

    # Costruisco una lista di N numeri dispari "grandi"e la spezzo in
    # più liste di lunghezza CHUNKSIZE da rendere disponibili ai
    # singoli client che vorranno "lavorarci su".
    nums = make_nums(base, count)
    chunksize = 43
    for i in range(0, len(nums), chunksize):
        job_q.put(nums[i:i + chunksize])

    # Siccome non ho nessun "collegamento" con i client, l'unico modo
    # che ho per sapere che hanno finito il loro lavoro è controllare
    # la lunghezza della coda dei risultati.
    res_count = 0
    res_dict = {}
    while res_count < count:
        out_dict = res_q.get()
        res_dict.update(out_dict)
        res_count += len(out_dict)

    # Sleep a bit before shutting down the server - to give clients
    # time to realize the job queue is empty and exit in an orderly
    # way.
    time.sleep(2)
    man.shutdown()
    return res_dict

def make_client_manager(ip, port, authkey):
    # Create a manager for a client.  This manager connects to a
    # server on the given address and exposes the get_job_q and
    # get_res_q methods for accessing the shared queues from the
    # server.  Return a manager object.

    class ServerQueueManager(SyncManager):
        pass

    ServerQueueManager.register('get_job_q')
    ServerQueueManager.register('get_res_q')

    manager = ServerQueueManager(address=(ip, port), authkey=authkey)
    manager.connect()

    print 'Client connected to %s:%s' % (ip, port)
    return manager

def runclient(ip, port, authkey):
    # Il client crea un client_manager da cui ottiene i due proxy alle
    # code e poi esegue mp_factorizer per generare PROC_COUNT processi
    # che fattorizzino.
    proc_count = 2
    man = make_client_manager(ip, port, authkey)
    job_q = man.get_job_q()
    res_q = man.get_res_q()
    mp_factorizer(job_q, res_q, proc_count)
```

```
def make_nums(base, count):
    # Un metodo semplice per restituire una lista di N numeri dispari
    # "grandi".
    return [base + i * 2 for i in range(count)]

def parse_args(args):

    parser = argparse.ArgumentParser(
        description="Multiprocess factorization.")

    g = parser.add_mutually_exclusive_group(required=True)
    g.add_argument('-s', '--server', action="store_true", default=False,
            help='Start a server process (not a client one)')
    g.add_argument('-c', '--client', action="store_true", default=False,
            help='Start a client process (not a server one)')

    add = parser.add_argument

    add('-i', '--ip-address', type=str, default="127.0.0.1",
        help="The IP address of the server this client should connect to.")

    add('-b', '--base-number', type=int, default=999999,
        help="The smallest number to factorize.")

    add('-n', '--numbers-count', type=int, default=999,
        help="The number of numbers to factorize.")

    add('-p', '--per-client-processes', type=int, default=2,
        help="The per client number of processes to spawn.")

    return parser.parse_args(args)

def main(args):

    options = parse_args(args)

    # this should also be options!
    port = 5000
    key  = "foo"

    if options.server:

        print ("Running server on port %d with key '%s'" % (
            port, key))
        print ("Factorizing %d odd numbers starting from %d" % (
            options.numbers_count, options.base_number))
        start = time.time()
        d = runserver(port, key,
                      options.base_number, options.numbers_count)
        passed = time.time() - start
        for k in sorted(d):
            print k, d[k]
        print ("Factorized %d numbers in %.2f seconds." % (
            options.numbers_count, passed))

    elif options.client:

        runclient(options.ip_address, port, key)

    return 0

if __name__ == '__main__':

    import sys
    sys.exit(main(sys.argv[1:]))
```