CS5590-0012

Group 7 - Programming Assignment 1

Team Members:

- Olena Khrystenko :: ovkncp@umsystem.edu
- Joe Moon :: jmn5y@umsystem.edu
- Jack Zhang :: zz9g4@umsystem.edu

Assignment Github: https://github.com/OlenaKhrystenko/master-worker

Program Description:

The program consists of three files(client.py, master.py, worker.py) along with two JSON data files (data-am.json, data-nz.json) and in order to function properly, two conditions must be met:

1. The master.py must be brought online first as it receives/registers workers; and
2. There must be a minimum of one worker instance

(See the github README file for instructions on operating the RPC program.)

On creating the RPC program, three features were implemented to improve functionality and provide robustness through error handling:

1. **Registering a Worker**

The program has been modified so that no workers are hard coded but rather must register with the master as they come online. This is achieved by specifying their own port number followed by the master port (e.g. **python3 worker.py <worker port> <master port>**). On initiation, a call is made to a registered master function (registerWorker) that records the worker's port number in a dictionary which may later be called upon when needing to execute a task from the client.

```
125     # Register Worker with Master
126     with xmlrpc.client.ServerProxy(f'http://localhost:{sys.argv[2]}/') as register:
127         if register.registerWorker(port):
128             print('\tRegistered with the master.  Ready to take on workload.')
129         else:
130             print("\tFailed to register worker with the master.")
```

Figure 1: worker.py

Checks are implemented to ensure that each worker port number is unique (lines 34-35 below) and that appropriate error cases are handled as necessary. Registered workers are recorded into the master.py "workers" dictionary.

Also, a specific JSON file does not have to be assigned to a worker but is done so dynamically depending on the input.

```python
# Adds a worker to a list of available workers able to receive tasks
def registerWorker(worker_port):
    try:
        int(worker_port)    # integer check, otherwise raise exception
        if worker_port in workers:
            return False    # Port number already in use
        else:
            workers.update(
                {worker_port: xmlrpc.client.ServerProxy(f"http://localhost:
                {worker_port}")})
            load_tracker.update({worker_port: 0})
            print(f'\tRegistering worker on port {worker_port} with MASTER...')
            return True
    except ValueError:
        print(f'Failed to recognize {worker_port} as a valid port number')
```

Figure 2: master.py

2. Load Balancing

After receiving a task from the client, the master will evaluate the load of each registered worker and assign that task to the worker with the lowest load. (Load is defined as the number of tasks performed). This is accomplished by each worker having a global "load_counter" variable that increments on each function call (see below).

```python
def getbyname(name):
    global load_counter
    load_counter += 1
```

Figure 3: worker.py

Upon each successful function call, the load count is embedded in the return object which is then recorded and tracked from the master's "load_tracker" dictionary with the worker as the key and the load count as the value.

```python
    return {
        'error': False,
        'load': load_counter,
        'result': data_table[name]
```

Figure 4: worker.py

```python
    result = s.getbyname(name)
    nameList.append(result.get('result'))

    # Update tracker data
    load_tracker[active_worker] = result['load']
```

Figure 5: master.py

The lowest load is applied on each function call by calling the "lowestLoad" function which evaluates the dictionary of registered workers and returns the port number (key) of the worker with the lowest load count (value) to perform the client's task.

```python
# Searches the 'workers' dictionary for the worker with the lowest load
def lowestLoad():
    return min(load_tracker, key=load_tracker.get)
```

Figure 6: master.py

3. **Rerouting Failed Workers**

As the program is running, should a worker fail, the program has been set up to capture that failure and reroute the client request to the next available worker with the lowest load. This is achieved by establishing an exception specific to a connection failure, which then removes that failed worker from the master's "workers" and "load_tracker" dictionaries, before calling the next worker with the same parameter.

```
110     except ConnectionRefusedError:
111         removeWorker(active_worker)
112         return getbylocation(location)
```

```
46  # Removes a worker upon a lost connection
47  def removeWorker(worker_port):
48      print(f"{worker_port} failed to connect.")
49      removed_worker = workers.pop(worker_port)
50      load_tracker.pop(worker_port)
51      print(f'Removed {removed_worker} from workers list.')
```

Figure 7: master.py

If there are no workers to call, then that violates the second condition for this program to function correctly.