568 Homework4 Report

Sijie Zhou(sz232)
Ching-Hang Hsu(ch450)

Introduction

In homework 4, we build a server-client based program that can handle the transaction-request from the client with Python. To be more specific, for the server part, we clean the previous database and create a new database every time, and create a serversocket to keep accepting the sending request from the client. We also write a custom method called parse_xml, which can differentiate the request types (ex: Account, Position, Order, Query, Cancel) and process each of them separately with different sub-methods. For each process for different types, there would be different operations for the database. Take one type, Order, as an example, Order would do the inspection to check whether current order could match with other previous orders in the database. More specifically, there would be two different conditions(Buying, Selling) in the submethod. Buying and Selling would have quite different SQL query operations for the database.

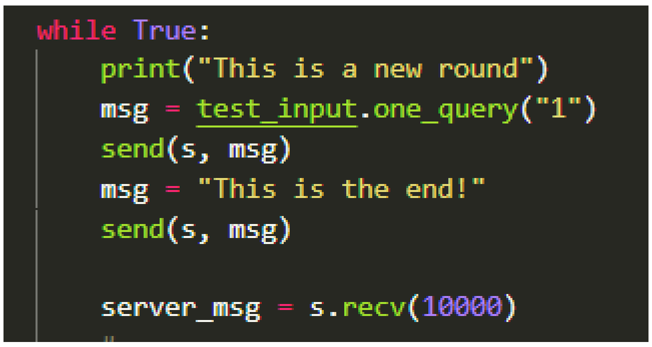
Other than the socket part, we also use the multiprocessing to try to make our code run faster. Owing to the characteristic of Python, there is no "real" multithreading package that could be used. We use the Pool object to help us to assign a new child process to the new client.

For the XML part, we write xml_generator.py to assemble different XML pieces to a complete context. In short, whenever the client or the server needs to send the information to the other side, we always use the functions inside xml_generator.py to combine the whole information .

Performance Analysis

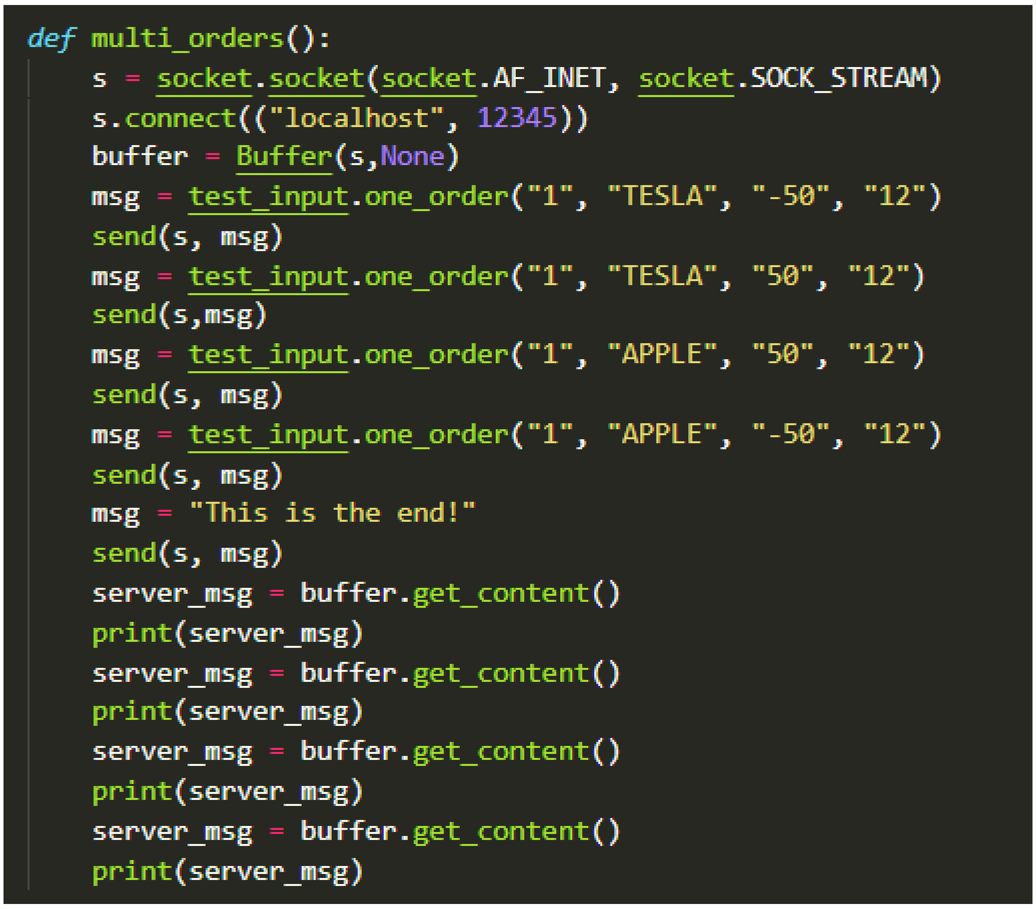
To test our code, we try to divide the testing into two categories: *simple command* and *complicated commands*.

For the *simple command* one, we only use one query command as the request from the client to send to the server. It looks like this in our performance_test.py:



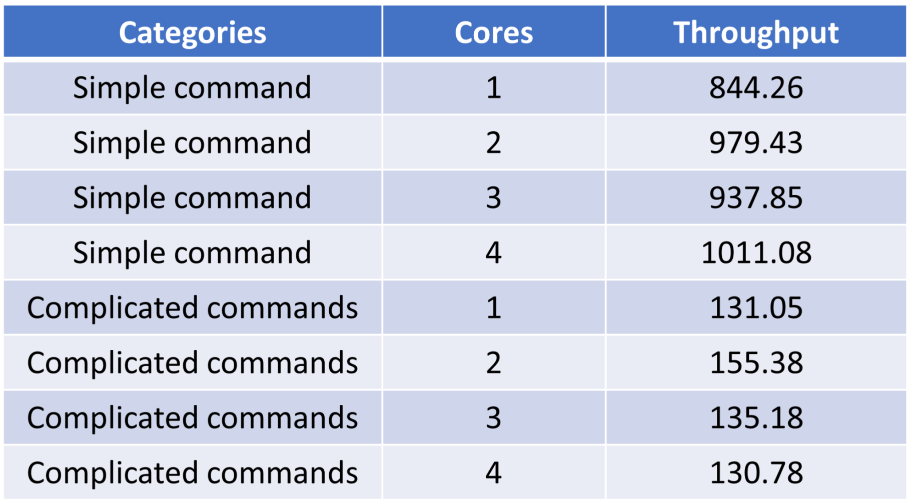
```
while True:
    print("This is a new round")
    msg = test_input.one_query("1")
    send(s, msg)
    msg = "This is the end!"
    send(s, msg)

    server_msg = s.recv(10000)
```

For the *complicated commands*, we use 4 orders to be the request from the client, and each two requests of them would be matched with each other. It looks like this in our multiprocess_client.py:



```python
def multi_orders():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("localhost", 12345))
    buffer = Buffer(s,None)
    msg = test_input.one_order("1", "TESLA", "-50", "12")
    send(s, msg)
    msg = test_input.one_order("1", "TESLA", "50", "12")
    send(s,msg)
    msg = test_input.one_order("1", "APPLE", "50", "12")
    send(s, msg)
    msg = test_input.one_order("1", "APPLE", "-50", "12")
    send(s, msg)
    msg = "This is the end!"
    send(s, msg)
    server_msg = buffer.get_content()
    print(server_msg)
    server_msg = buffer.get_content()
    print(server_msg)
    server_msg = buffer.get_content()
    print(server_msg)
    server_msg = buffer.get_content()
    print(server_msg)
```

For our testing experiments, we get the result for both categories with different cores CPU:



Categories	Cores	Throughput
Simple command	1	844.26
Simple command	2	979.43
Simple command	3	937.85
Simple command	4	1011.08
Complicated commands	1	131.05
Complicated commands	2	155.38
Complicated commands	3	135.18
Complicated commands	4	130.78

According to the result graph, we could notice that if the server processes the *simple command*, the performance for the server would increase corresponding to the increase for the CPU cores. We think that it is reasonable, because the server just only implements the simple SQL query line, and it is hard to meet some serious latency during the running. However, for the *complicated commands*, we could easily find that the throughput would not increase according to the increase of the CPU cores. Although it looks a little bit weird for this result, we think it would also be reasonable. To be more specific, complicated commands contain more operations related to the database, and it is easy to have some bottleneck there. Even though we have multiple CPU cores to help us to speedup with the multiprocessing, there would still wait for the end of the operations of other processes. Accordingly, the consumption for the time would be really alike with different CPU cores.