SCALABILITY REPORT

ECE 568 Engineering Robust Server Software

Author
Zixiao Ma (zm96)
Shuqi Shen (ss1481)

April 8, 2024

1 Experimental Results

In this experiment, we tested different number of tasks on different number of CPU cores. The experimental results are shown below. Our application uses a thread pool with a fixed number of 10 threads.

1.1 Visualizing Experimental Data

1.1.1 Table of Data for Scalability

Cores	**Tasks**	**Time / s**
1	30	5.383
1	100	6.378
1	200	7.372
1	500	10.632
1	1000	14.437
2	30	3.149
2	100	3.896
2	200	5.357
2	500	8.336
2	1000	12.882
4	30	2.874
4	100	3.558
4	200	5.250
4	500	8.376
4	1000	12.826

Table 1: Table of Data for Scalability

1.1.2 Line graph of data for scalability

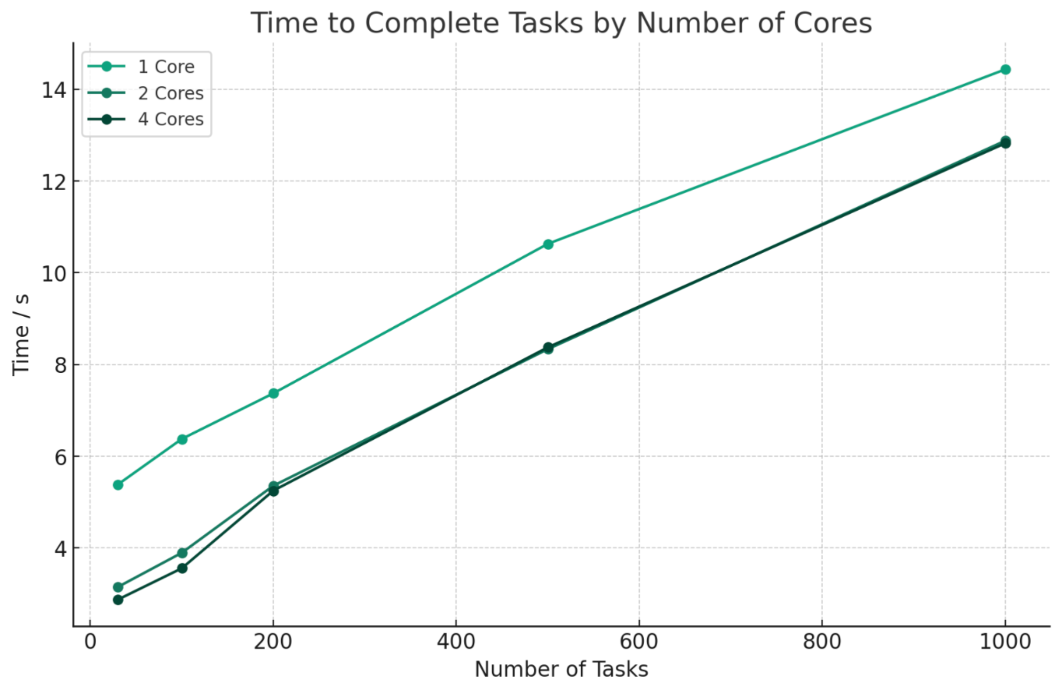


Figure 1: Line Graph of Experimental Results

1.2 Performance Description

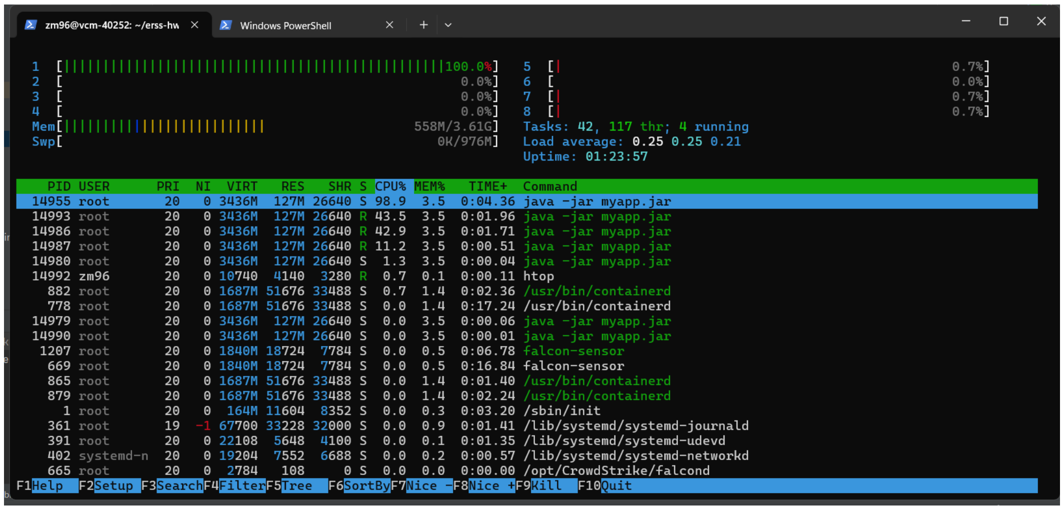
Increasing the number of cores significantly reduces the time taken for tasks. This improvement is most noticeable when moving from 1 core to 2 cores, and still significant but less drastic when moving to 4 cores.The diminishing returns effect becomes apparent with higher core counts. The jump from 1 to 2 cores offers a larger performance gain compared to the jump from 2 to 4 cores.

2 Scalability Analysis

2.1 Analysis of the Performance Divergence on Different CPU Cores

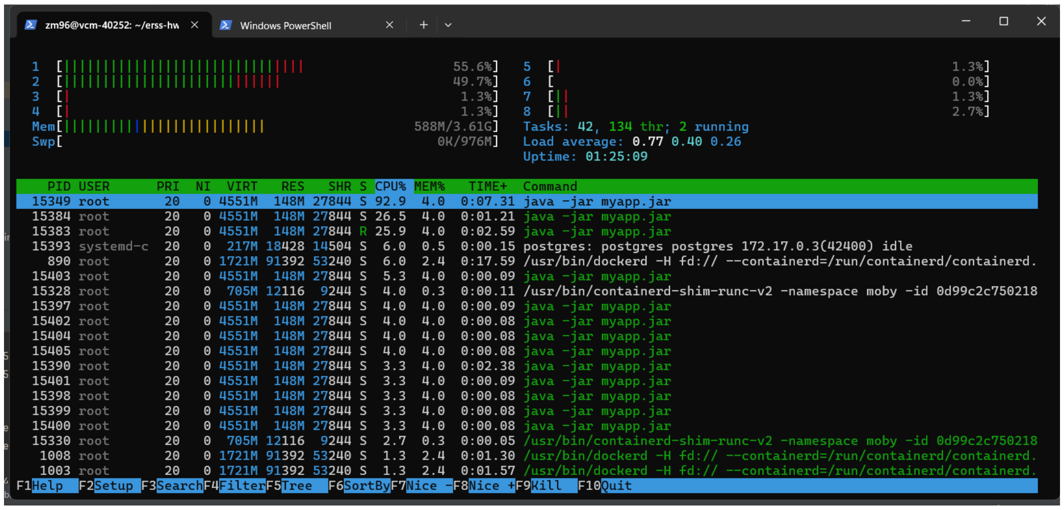
As it is mentioned above, when moving from 1 core to 2 cores, the performance improves a lot, and the performance does not change much when moving from 2 cores to 4 cores. The analysis of this experimental observation suggests that this is due to the CPU utilization drop.

From the provided screenshots and the data on CPU utilization, it is observed that when the server is operating with a single core, the CPU utilization is near maximum. This suggests that the CPU is fully occupied with the workload, leading to a bottleneck where the process cannot be executed any faster because there is no more CPU capacity available. When the server operates with two and then four cores, there appears to be some unused capacity, as indicated by the lower CPU utilization percentages. Therefore, the completion times of tasks do not significantly change as might be expected with the additional cores.



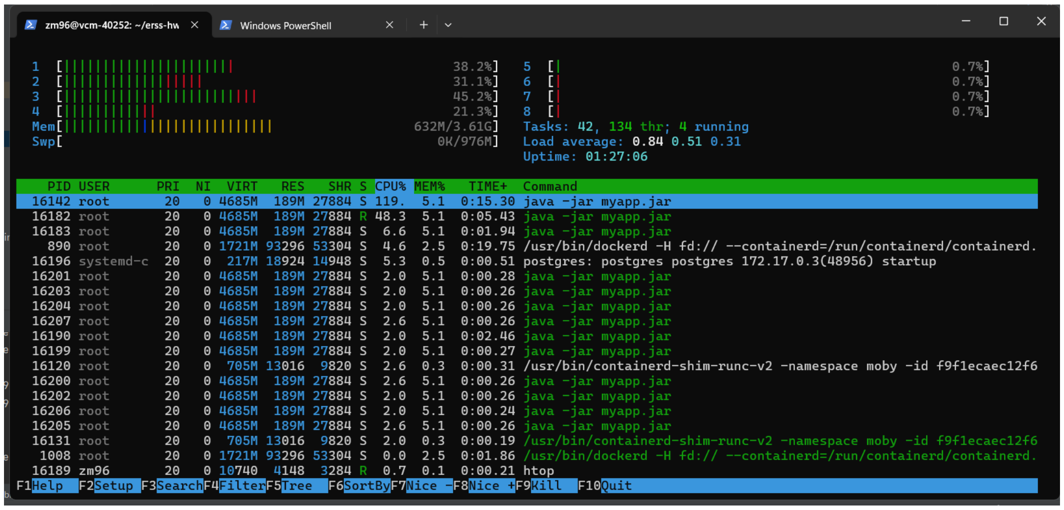
```
zm96@vcm-40252: ~/erss-hw    Windows PowerShell

  1  [||||||||||||||||||||||||||||||||||||||||100.0%]   5  [|                                          0.7%]
  2  [                                          0.0%]   6  [                                           0.0%]
  3  [                                          0.0%]   7  [|                                          0.7%]
  4  [                                          0.0%]   8  [|                                          0.7%]
  Mem[||||||||||||||||||||||          558M/3.61G]   Tasks: 42, 117 thr; 4 running
  Swp[                                    0K/976M]   Load average: 0.25 0.25 0.21
                                                     Uptime: 01:23:57

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  14955 root        20   0 3436M  127M 26640 S 98.9  3.5  0:04.36 java -jar myapp.jar
  14993 root        20   0 3436M  127M 26640 R 43.5  3.5  0:01.96 java -jar myapp.jar
  14986 root        20   0 3436M  127M 26640 R 42.9  3.5  0:01.71 java -jar myapp.jar
  14987 root        20   0 3436M  127M 26640 R 11.2  3.5  0:00.51 java -jar myapp.jar
  14980 root        20   0 3436M  127M 26640 S  1.3  3.5  0:00.04 java -jar myapp.jar
  14992 zm96        20   0 10740  4140  3280 R  0.7  0.1  0:00.11 htop
    882 root        20   0 1687M 51676 33488 S  0.7  1.4  0:02.36 /usr/bin/containerd
    778 root        20   0 1687M 51676 33488 S  0.0  1.4  0:17.24 /usr/bin/containerd
  14979 root        20   0 3436M  127M 26640 S  0.0  3.5  0:00.06 java -jar myapp.jar
  14990 root        20   0 3436M  127M 26640 S  0.0  3.5  0:00.01 java -jar myapp.jar
   1207 root        20   0 1840M 18724  7784 S  0.0  0.5  0:06.78 falcon-sensor
    669 root        20   0 1840M 18724  7784 S  0.0  0.5  0:16.84 falcon-sensor
    865 root        20   0 1687M 51676 33488 S  0.0  1.4  0:01.40 /usr/bin/containerd
    879 root        20   0 1687M 51676 33488 S  0.0  1.4  0:02.24 /usr/bin/containerd
      1 root        20   0  164M 11604  8352 S  0.0  0.3  0:03.20 /sbin/init
    361 root        19  -1 67700 33228 32000 S  0.0  0.9  0:01.41 /lib/systemd/systemd-journald
    391 root        20   0 22108  5648  4100 S  0.0  0.1  0:01.35 /lib/systemd/systemd-udevd
    402 systemd-n   20   0 19204  7552  6688 S  0.0  0.2  0:00.57 /lib/systemd/systemd-networkd
    665 root        20   0  2784   108     0 S  0.0  0.0  0:00.00 /opt/CrowdStrike/falcond
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 2: CPU utilization for 1 CPU core



```
zm96@vcm-40252: ~/erss-hw    Windows PowerShell

  1  [||||||||||||||||||||||||||||||                 55.6%]   5  [|                                         1.3%]
  2  [|||||||||||||||||||||||||||                    49.7%]   6  [                                          0.0%]
  3  [|                                               1.3%]   7  [||                                        1.3%]
  4  [|                                               1.3%]   8  [||                                        2.7%]
  Mem[||||||||||||||||||||||||||                 588M/3.61G]   Tasks: 42, 134 thr; 2 running
  Swp[                                            0K/976M]   Load average: 0.77 0.40 0.26
                                                              Uptime: 01:25:09

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  15349 root        20   0 4551M  148M 27844 S 92.9  4.0  0:07.31 java -jar myapp.jar
  15384 root        20   0 4551M  148M 27844 S 26.5  4.0  0:01.21 java -jar myapp.jar
  15383 root        20   0 4551M  148M 27844 R 25.9  4.0  0:02.59 java -jar myapp.jar
  15393 systemd-c   20   0  217M 18428 14504 S  6.0  0.5  0:00.15 postgres: postgres postgres 172.17.0.3(42400) idle
    890 root        20   0 1721M 91392 53240 S  6.0  2.4  0:17.59 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.
  15403 root        20   0 4551M  148M 27844 S  5.3  4.0  0:00.09 java -jar myapp.jar
  15328 root        20   0  705M 12116  9244 S  4.0  0.3  0:00.11 /usr/bin/containerd-shim-runc-v2 -namespace moby -id 0d99c2c750218
  15397 root        20   0 4551M  148M 27844 S  4.0  4.0  0:00.09 java -jar myapp.jar
  15402 root        20   0 4551M  148M 27844 S  4.0  4.0  0:00.08 java -jar myapp.jar
  15404 root        20   0 4551M  148M 27844 S  4.0  4.0  0:00.08 java -jar myapp.jar
  15405 root        20   0 4551M  148M 27844 S  4.0  4.0  0:00.08 java -jar myapp.jar
  15390 root        20   0 4551M  148M 27844 S  3.3  4.0  0:02.38 java -jar myapp.jar
  15401 root        20   0 4551M  148M 27844 S  3.3  4.0  0:00.09 java -jar myapp.jar
  15398 root        20   0 4551M  148M 27844 S  3.3  4.0  0:00.08 java -jar myapp.jar
  15399 root        20   0 4551M  148M 27844 S  3.3  4.0  0:00.08 java -jar myapp.jar
  15400 root        20   0 4551M  148M 27844 S  3.3  4.0  0:00.08 java -jar myapp.jar
  15330 root        20   0  705M 12116  9244 S  2.7  0.3  0:00.05 /usr/bin/containerd-shim-runc-v2 -namespace moby -id 0d99c2c750218
   1008 root        20   0 1721M 91392 53240 S  1.3  2.4  0:01.30 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.
   1003 root        20   0 1721M 91392 53240 S  1.3  2.4  0:01.57 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 3: CPU utilization for 2 CPU cores



```
zm96@vcm-40252: ~/erss-hw    Windows PowerShell

  1  [||||||||||||||||||||||                         38.2%]   5  [|                                         0.7%]
  2  [|||||||||||||||||                              31.1%]   6  [|                                         0.7%]
  3  [|||||||||||||||||||||||||                      45.2%]   7  [|                                         0.7%]
  4  [||||||||||||                                   21.3%]   8  [|                                         0.7%]
  Mem[|||||||||||||||||||||||||||                632M/3.61G]   Tasks: 42, 134 thr; 4 running
  Swp[                                            0K/976M]   Load average: 0.84 0.51 0.31
                                                              Uptime: 01:27:06

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  16142 root        20   0 4685M  189M 27884 S 119.  5.1  0:15.30 java -jar myapp.jar
  16182 root        20   0 4685M  189M 27884 R 48.3  5.1  0:05.43 java -jar myapp.jar
  16183 root        20   0 4685M  189M 27884 S  6.6  5.1  0:01.94 java -jar myapp.jar
    890 root        20   0 1721M 93296 53304 S  4.6  2.5  0:19.75 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.
  16196 systemd-c   20   0  217M 18924 14948 S  5.3  0.5  0:00.51 postgres: postgres postgres 172.17.0.3(48956) startup
  16201 root        20   0 4685M  189M 27884 S  2.0  5.1  0:00.28 java -jar myapp.jar
  16203 root        20   0 4685M  189M 27884 S  2.0  5.1  0:00.26 java -jar myapp.jar
  16204 root        20   0 4685M  189M 27884 S  2.0  5.1  0:00.26 java -jar myapp.jar
  16207 root        20   0 4685M  189M 27884 S  2.6  5.1  0:00.26 java -jar myapp.jar
  16190 root        20   0 4685M  189M 27884 S  2.0  5.1  0:02.46 java -jar myapp.jar
  16199 root        20   0 4685M  189M 27884 S  2.0  5.1  0:00.27 java -jar myapp.jar
  16120 root        20   0  705M 13016  9820 S  2.6  0.3  0:00.31 /usr/bin/containerd-shim-runc-v2 -namespace moby -id f9f1ecaec12f6
  16200 root        20   0 4685M  189M 27884 S  2.6  5.1  0:00.26 java -jar myapp.jar
  16202 root        20   0 4685M  189M 27884 S  2.0  5.1  0:00.26 java -jar myapp.jar
  16206 root        20   0 4685M  189M 27884 S  2.0  5.1  0:00.24 java -jar myapp.jar
  16205 root        20   0 4685M  189M 27884 S  2.6  5.1  0:00.26 java -jar myapp.jar
  16131 root        20   0  705M 13016  9820 S  2.0  0.3  0:00.19 /usr/bin/containerd-shim-runc-v2 -namespace moby -id f9f1ecaec12f6
   1008 root        20   0 1721M 93296 53304 S  0.0  2.5  0:01.86 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.
  16189 zm96        20   0 10740  4148  3284 R  0.7  0.1  0:00.21 htop
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 4: CPU utilization for 4 CPU cores

2.2 Analysis of the Performance Divergence on Different Task Numbers

As it is mentioned above, as the number of tasks increases, the response times increase linearly. The analysis of this experimental observation suggests that this is due to the bottlenecks of IO including Disk I/O, Network I/O and Database I/O.

2.2.1 Disk I/O Impact

Increased Read/Write Operations: As the number of tasks increases, if each task involves disk I/O operations, the total number of read and write operations also increases. This can lead to queuing delays as the disk works to process a larger queue of operations.

Concurrency and Access Times: Disks have a limit on how many concurrent operations they can handle. As more tasks attempt simultaneous access, the time to complete each operation can increase due to the need for the disk to switch between tasks.

2.2.2 Network I/O Impact

Bandwidth Saturation: With more tasks potentially sending or receiving data, the demand for network bandwidth can exceed supply, leading to congestion and increased latency.

Packet Queuing: Network devices may need to queue packets for transmission if the volume exceeds the immediate processing capability, which can result in increased network I/O latency.

2.2.3 Database I/O Impact

Connection and Query Load: More tasks often mean more database connections and queries. If a database cannot handle the increased load efficiently, it can become a bottleneck, with tasks waiting in line to access the database. Lock Contention: In a multi-task environment, tasks may compete for locks on database resources. As the number of tasks increases, so does lock contention, which can delay the completion of database operations.

In summary, as the number of tasks increases, the potential for I/O bottlenecks becomes more pronounced due to increased demands on disk, network, and database resources. These bottlenecks can result in tasks taking longer to complete, which may explain why the time does not decrease proportionally with additional CPU cores, as seen in the provided data. The increase in time with the increase in the number of tasks, even with more CPU cores available, underscores the importance of I/O in overall system performance.

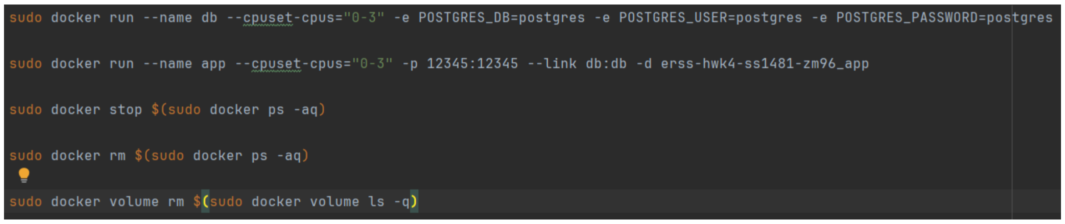
3 Instructions on Experimental Result Reproduction

3.1 Run the Server

To run the server with different number of CPU cores, you should firstly use **sudo docker-compose up** and then **sudo docker-compose down -v** to make sure that all images are created. After that, you should use the first two commands in testing/commands.sh to run the application with specific CPU cores. After each test, you should run the last three commands in commands.sh to clean the stored and cached data for following tests.

3.2 Run the Tesing client

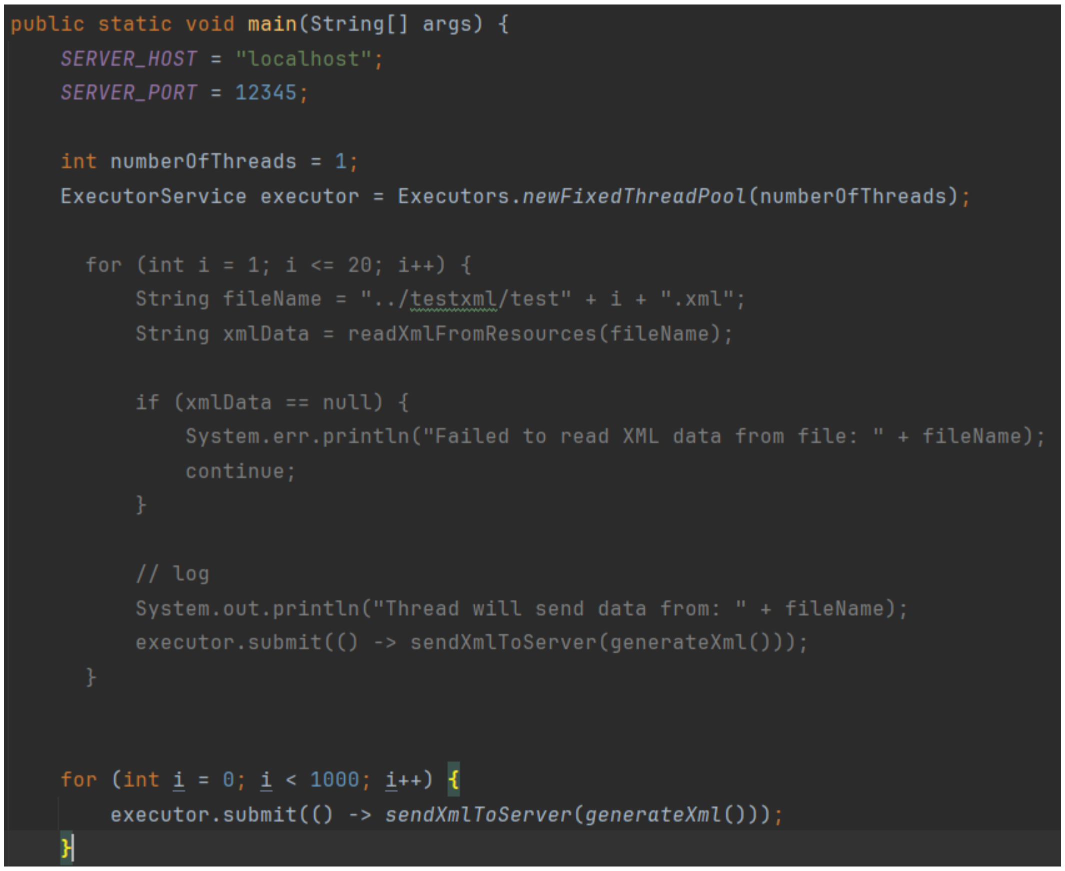
To reproduce the experimental result, you should open testing/client/SocketClient.java and go to the main function. If you prefer to run the test in the IDE like IDEA in local machine, make sure the working directory is testing/client, and change SERVER_HOST to the IP address of the server in the main function. If you prefer to run the test in the same server of the application, change SERVER_HOST to the localhost in the main function, and enter the testing/client directory and then compile and run SocketClient.java.



```
sudo docker run --name db --cpuset-cpus="0-3" -e POSTGRES_DB=postgres -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres

sudo docker run --name app --cpuset-cpus="0-3" -p 12345:12345 --link db:db -d erss-hwk4-ss1481-zm96_app

sudo docker stop $(sudo docker ps -aq)

sudo docker rm $(sudo docker ps -aq)

sudo docker volume rm $(sudo docker volume ls -q)
```

Figure 5: commands.sh

To reproduce the scalability test, uncomment the code below and change 1000 to the number of tasks you want to test. To test the functionalities, we have provided comprehensive tests in the testing/testxml directory, and you can test them in the loop code above the scalability testing code, or read and send contents in whatever test files as you like. After you shut the program down, the total operation time will be printed.



```java
public static void main(String[] args) {
    SERVER_HOST = "localhost";
    SERVER_PORT = 12345;

    int numberOfThreads = 1;
    ExecutorService executor = Executors.newFixedThreadPool(numberOfThreads);

      for (int i = 1; i <= 20; i++) {
          String fileName = "../testxml/test" + i + ".xml";
          String xmlData = readXmlFromResources(fileName);

          if (xmlData == null) {
              System.err.println("Failed to read XML data from file: " + fileName);
              continue;
          }

          // log
          System.out.println("Thread will send data from: " + fileName);
          executor.submit(() -> sendXmlToServer(generateXml()));
      }


    for (int i = 0; i < 1000; i++) {
        executor.submit(() -> sendXmlToServer(generateXml()));
    }
```

Figure 6: Main function of SocketClient.java