Coursework 1

Imperial College London

Department of Computing

Distributed Algorithm

Author:
Linshan Li(ll3720)
Zhenghui Wang(zw2520)

Date: February 18, 2021

1 Diagram of structure

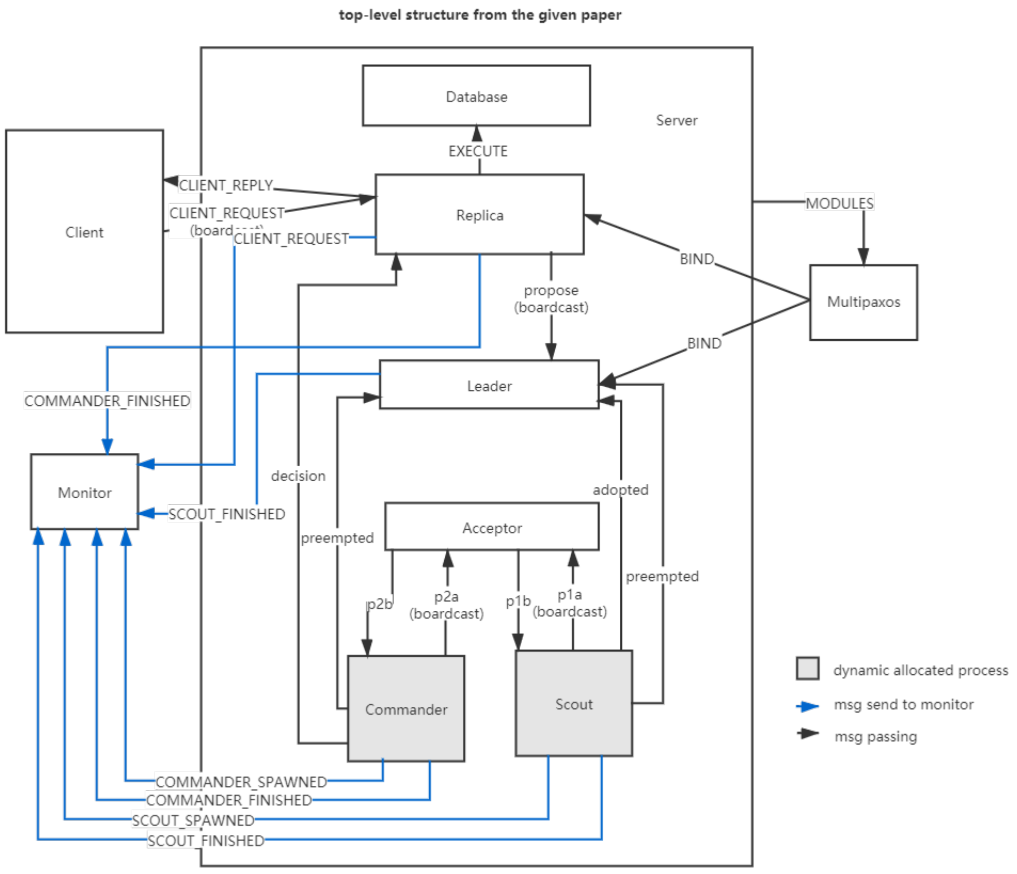


Figure 1: Top-level Structure

2 Facility and Environment

Environment	Status
Original OS	Windows 10
Virtual Machine	virtualbox
OS	Linux + Ubuntu(64-bit)
Processor	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz(Linshan) AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz(Zhenghui)
RAM	16G
Core	10(Linshan)/8(Zhenghui)

3 Design and Implementation

3.1 Basic design

Our design is based on the Pseudocode of the given paper (Paxo Made Moderately Complex). the major implementation divided into three parts: Acceptor, Replicas and Leader.

- Acceptor response to requests from Leaders (we count Scout and Commander as part of the Leader), receive msg p1a/p2a and returns msg p2a/p2b. The returned messages contain current ballot number and whether the value been accepted.
- Replicas receive requests from clients, assign to specific slots, and create commands, and the replica also checks whether the current slot already in the configuration and send slot and command (propose) to all leaders. Replicas also deal with the message received from Commander.
- Leader will spawn new Scout when it is created and spawn Commander/Scout when receiving specific msg. Leaders got two helper handler which can be further divided into Commander & Scout.
- Commander is dynamically spawned by Leaders. It focuses on one command each time, sends msg p2a to all acceptors, and receives msg p2b from acceptors. It checks whether it has been accepted by most of the acceptors and sends the result :decision/:preempted back to Replica or Leader, then the process destroyed.
- Scout is dynamically spawned by Leaders. It sends msg p1a to all acceptors, acceptors sends back msg p1b, by considering whether the ballot number changes and send :adpoted/:preempted msg back to Leader, then the process destroyed.

A Ping-Pong may happen when a leader X preempted by another leader Y. leader X will spawn a new Scout and then finally preempt leader Y. It could keep repeat.

We add a random process sleep time to avoid the Ping-Pong situation and livelock. However, it works well for a certain amount of time initially, and then the process slow drops down.

Another possible way to solve this issue is to pick a replica_leader by their ids. The leader is the one with the highest id. This replica_leaderis the only one who can send prepose. Replicas send pings messages to each other every T ms. Suppose a replica doesn't receive any msg with a higher id than its own for a 2*T period. Then itself become a new replica_leader.
Under the support of the multi-paxos algorithm, all servers will achieve consistency eventually.

3.2 Sequence diagram

The detailed sequence is shown below, including the five parts mentioned above as well as client:

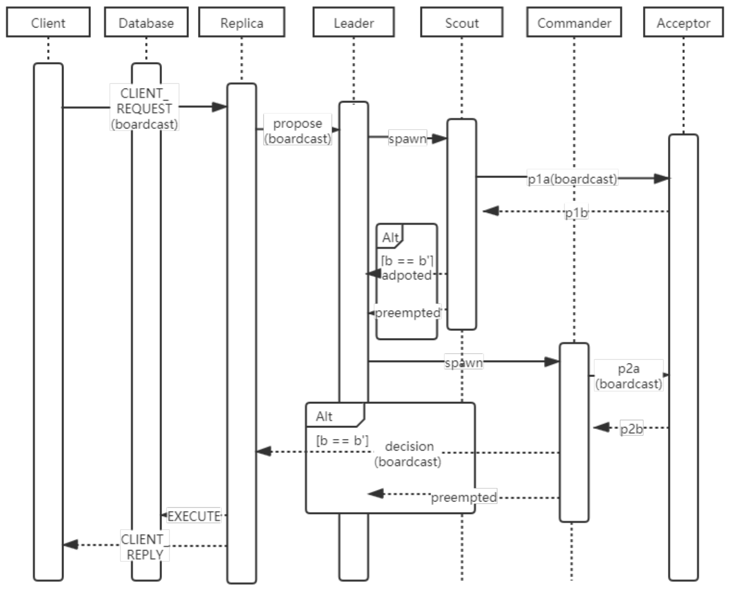


Figure 2: Sequence Diagram

4 Debug and Test

We mainly got three ways to debug and test our system. First one is the Error Message Detection, which will throw out an IO.puts when an unexpected messaged received. The second one is Working Flow Inspection, which threw out an IO.puts when a new state has happened. For instance, a leader received a preempted/ leader to create a new Scout. The final method is Parameter Controller, which will be described as below

- Error Message Detection, if the message that any component in the server receives does not match the program's cases set in the program, the result will output "Module_name-funcion_name received unexpected msg".
- Working Flow Inspection, set the parameters config. debug as 1 or CONFIG = custom_debug. Which display the transmission of messages intuitively and helpful to find bugs. It quite powerful when tracing the message flow during running. it will be better to use it with a low request amount case. Otherwise, it might look not very pleasant.
- Parameter Controller, adjust the number of clients and servers, as well as other configuration parameters, such as the max_request, window size. For instance, we can set max_request to 1/5, client = 1, server = 1 to test our system's performance and increase the number of client/server or max_request to find out what situation the bug occurred.

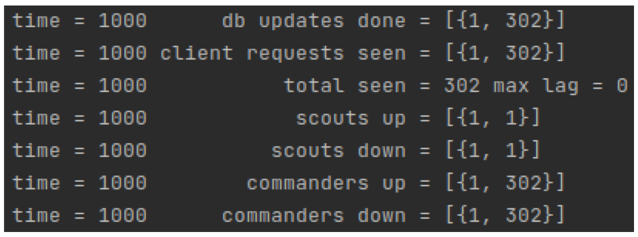
5 Evaluation

The group set different situations to test and evaluate the performance of the built multipaxos system, and the part below displays some of the test cases and their running result.

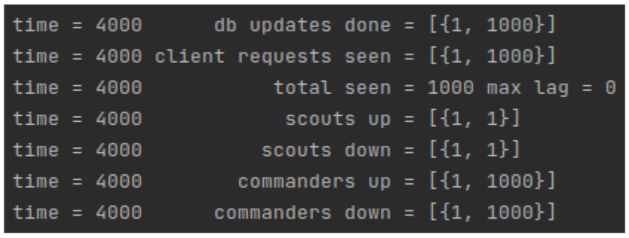
5.1 Test Case 1: Single server and client

SERVERS = 1	CLIENTS = 1	CONFIG = default	max_request = 1000	Window = 5

Result:



```
time = 1000      db updates done = [{1, 302}]
time = 1000 client requests seen = [{1, 302}]
time = 1000           total seen = 302 max lag = 0
time = 1000            scouts up = [{1, 1}]
time = 1000          scouts down = [{1, 1}]
time = 1000        commanders up = [{1, 302}]
time = 1000      commanders down = [{1, 302}]
```



```
time = 4000      db updates done = [{1, 1000}]
time = 4000 client requests seen = [{1, 1000}]
time = 4000           total seen = 1000 max lag = 0
time = 4000            scouts up = [{1, 1}]
time = 4000          scouts down = [{1, 1}]
time = 4000        commanders up = [{1, 1000}]
time = 4000      commanders down = [{1, 1000}]
```

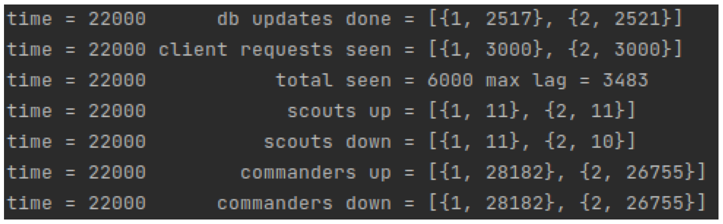
Figure 3: Test Case 1 Result

The parameter setting and the screenshot of the output have been shown above. The complete output file for this testcase is in "TC1_1s_1c.txt". There is no livelock exists and can initially show whether the system works successfully. Since only one client sends 1000 requests, the server finishes the work after 4 seconds. We can see that the average process speed is around 200-300 requests per second. The process speed is effect by the computation ability of our VM machine and is also effects the process.sleep() we added into our program.

5.2 Test Case 2: Compare performance with different window

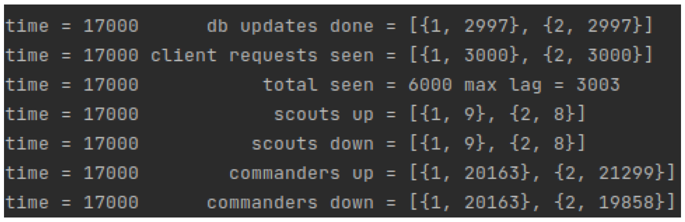
SERVERS = 2	CLIENTS = 3	CONFIG = default	max_request = 1000	window = 5
SERVERS = 2	CLIENTS = 3	CONFIG = default	max_request = 1000	window = 100

Result:



```
time = 22000      db updates done = [{1, 2517}, {2, 2521}]
time = 22000 client requests seen = [{1, 3000}, {2, 3000}]
time = 22000           total seen = 6000 max lag = 3483
time = 22000            scouts up = [{1, 11}, {2, 11}]
time = 22000          scouts down = [{1, 11}, {2, 10}]
time = 22000        commanders up = [{1, 28182}, {2, 26755}]
time = 22000      commanders down = [{1, 28182}, {2, 26755}]
```

Figure 4: Test Case 2: window = 5



```
time = 17000      db updates done = [{1, 2997}, {2, 2997}]
time = 17000 client requests seen = [{1, 3000}, {2, 3000}]
time = 17000           total seen = 6000 max lag = 3003
time = 17000            scouts up = [{1, 9}, {2, 8}]
time = 17000          scouts down = [{1, 9}, {2, 8}]
time = 17000        commanders up = [{1, 20163}, {2, 21299}]
time = 17000      commanders down = [{1, 20163}, {2, 19858}]
```

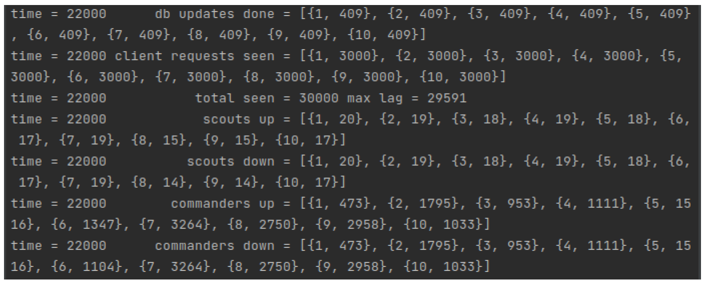
Figure 5: Test Case 2: window = 100

Since the larger window will enable more proposals simultaneously, the system can process more operations simultaneity, leading to the reduction of the response time. In Figure 4, since the MAX_TIME is set as 25 seconds. There are still around 500 requests that are not handled until at 22 seconds; however, when the window size is set to 100, the updates almost finish in the 17 seconds, and a livelock occurs (shown in Appendix). The complete output file for this testcase is in "TC2_1_2s_3c.txt" and "TC2_2_2s_3c.txt" respectively.

5.3 Test Case 3: Impact of the number of server/client on the results

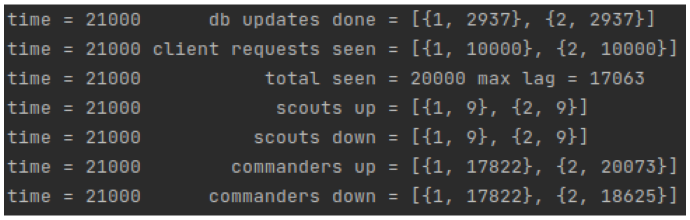
SERVERS = 10	CLIENTS = 3	CONFIG = default	max_request = 1000	window = 5
SERVERS = 2	CLIENTS = 10	CONFIG = default	max_request = 1000	window = 5

Result:



```
time = 22000      db updates done = [{1, 409}, {2, 409}, {3, 409}, {4, 409}, {5, 409}
, {6, 409}, {7, 409}, {8, 409}, {9, 409}, {10, 409}]
time = 22000 client requests seen = [{1, 3000}, {2, 3000}, {3, 3000}, {4, 3000}, {5,
3000}, {6, 3000}, {7, 3000}, {8, 3000}, {9, 3000}, {10, 3000}]
time = 22000           total seen = 30000 max lag = 29591
time = 22000            scouts up = [{1, 20}, {2, 19}, {3, 18}, {4, 19}, {5, 18}, {6,
 17}, {7, 19}, {8, 15}, {9, 15}, {10, 17}]
time = 22000          scouts down = [{1, 20}, {2, 19}, {3, 18}, {4, 19}, {5, 18}, {6,
 17}, {7, 19}, {8, 14}, {9, 14}, {10, 17}]
time = 22000        commanders up = [{1, 473}, {2, 1795}, {3, 953}, {4, 1111}, {5, 15
16}, {6, 1347}, {7, 3264}, {8, 2750}, {9, 2958}, {10, 1033}]
time = 22000      commanders down = [{1, 473}, {2, 1795}, {3, 953}, {4, 1111}, {5, 15
16}, {6, 1104}, {7, 3264}, {8, 2750}, {9, 2958}, {10, 1033}]
```

Figure 6: SERVERS = 10, CLIENTS = 3



```
time = 21000      db updates done = [{1, 2937}, {2, 2937}]
time = 21000 client requests seen = [{1, 10000}, {2, 10000}]
time = 21000           total seen = 20000 max lag = 17063
time = 21000            scouts up = [{1, 9}, {2, 9}]
time = 21000          scouts down = [{1, 9}, {2, 9}]
time = 21000        commanders up = [{1, 17822}, {2, 20073}]
time = 21000      commanders down = [{1, 17822}, {2, 18625}]
```

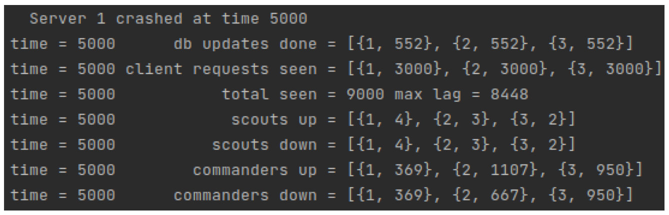
Figure 7: SERVERS = 2, CLIENTS = 10

More servers it has, harder to reach consensus, more clients it has, faster response time it gets. The complete output file for this testcase is in "TC3_1_10s_3c.txt" and "TC3_2_2s_10c.txt" respectively.

5.4 Test Case 4: Server1 crash

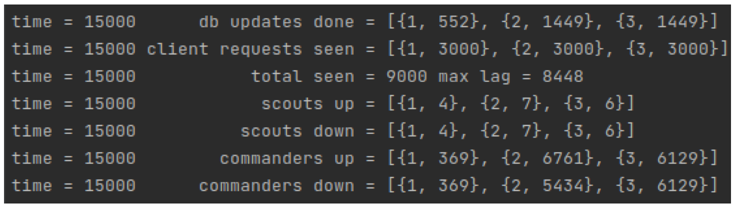
SERVERS = 3	CLIENTS = 3	CONFIG = default	max_request = 1000	Crash = Server1	Time = 5000

Result:



```
  Server 1 crashed at time 5000
time = 5000      db updates done = [{1, 552}, {2, 552}, {3, 552}]
time = 5000 client requests seen = [{1, 3000}, {2, 3000}, {3, 3000}]
time = 5000           total seen = 9000 max lag = 8448
time = 5000            scouts up = [{1, 4}, {2, 3}, {3, 2}]
time = 5000          scouts down = [{1, 4}, {2, 3}, {3, 2}]
time = 5000        commanders up = [{1, 369}, {2, 1107}, {3, 950}]
time = 5000      commanders down = [{1, 369}, {2, 667}, {3, 950}]
```

Figure 8: Crash server 1 at 5s (t = 5)



```
time = 15000      db updates done = [{1, 552}, {2, 1449}, {3, 1449}]
time = 15000 client requests seen = [{1, 3000}, {2, 3000}, {3, 3000}]
time = 15000           total seen = 9000 max lag = 8448
time = 15000            scouts up = [{1, 4}, {2, 7}, {3, 6}]
time = 15000          scouts down = [{1, 4}, {2, 7}, {3, 6}]
time = 15000        commanders up = [{1, 369}, {2, 6761}, {3, 6129}]
time = 15000      commanders down = [{1, 369}, {2, 5434}, {3, 6129}]
```

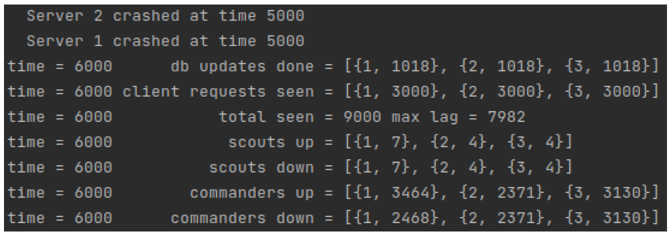
Figure 9: Crash server 1 at 5s (t = 15)

When set the total server number as 3, and crash one server at the fifth second, since the 2 * number of crashed server + 1 ≤ number of total server, the system still can work, so that the first server stops working and remains the same db updated, commander and scout created, while the others still working. This is because there is no bound on timing for delivering and processing messages, it is impossible for other processes to know for certain that the process has failed. The complete output file for this testcase is in "TC4_serve1_clash.txt" respectively.

5.5 Test Case 5: Server1 and Server2 crash

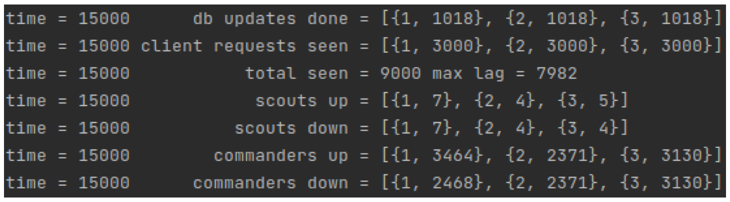
SERVERS = 3	CLIENTS = 3	CONFIG = default	max_request = 1000	Crash = Server1,2	Time = 5000

Result:



```
  Server 2 crashed at time 5000
  Server 1 crashed at time 5000
time = 6000      db updates done = [{1, 1018}, {2, 1018}, {3, 1018}]
time = 6000 client requests seen = [{1, 3000}, {2, 3000}, {3, 3000}]
time = 6000           total seen = 9000 max lag = 7982
time = 6000            scouts up = [{1, 7}, {2, 4}, {3, 4}]
time = 6000          scouts down = [{1, 7}, {2, 4}, {3, 4}]
time = 6000        commanders up = [{1, 3464}, {2, 2371}, {3, 3130}]
time = 6000      commanders down = [{1, 2468}, {2, 2371}, {3, 3130}]
```

Figure 10: Crash server 1,2 at 5s (At time = 6s)



```
time = 15000      db updates done = [{1, 1018}, {2, 1018}, {3, 1018}]
time = 15000 client requests seen = [{1, 3000}, {2, 3000}, {3, 3000}]
time = 15000           total seen = 9000 max lag = 7982
time = 15000            scouts up = [{1, 7}, {2, 4}, {3, 5}]
time = 15000          scouts down = [{1, 7}, {2, 4}, {3, 4}]
time = 15000        commanders up = [{1, 3464}, {2, 2371}, {3, 3130}]
time = 15000      commanders down = [{1, 2468}, {2, 2371}, {3, 3130}]
```

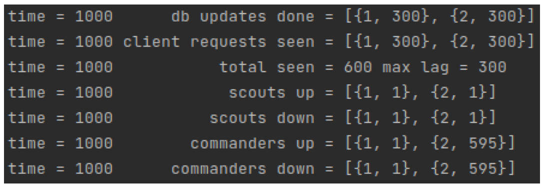
Figure 11: Crash server 1,2 at 5s (At time = 15s)

When set the total server number as 3, and crash two servers at the fifth second, 2 * number of crashed server + 1 >number of total server, the system cannot work, all the servers remain the same status(db updated, commander and scout created). The complete output file for this testcase is in "TC5_ser1_ser2_clash.txt" respectively.

5.6 Test Case 6: Performance with different max requests

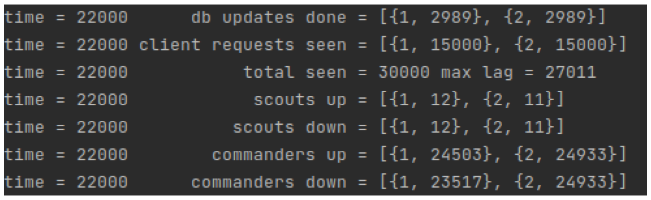
SERVERS = 2	CLIENTS = 3	CONFIG = default	window = 5	max_requests = 100
SERVERS = 2	CLIENTS = 3	CONFIG = default	window = 5	max_requests = 5000

Result:



```
time = 1000      db updates done = [{1, 300}, {2, 300}]
time = 1000 client requests seen = [{1, 300}, {2, 300}]
time = 1000           total seen = 600 max lag = 300
time = 1000            scouts up = [{1, 1}, {2, 1}]
time = 1000          scouts down = [{1, 1}, {2, 1}]
time = 1000        commanders up = [{1, 1}, {2, 595}]
time = 1000      commanders down = [{1, 1}, {2, 595}]
```

Figure 12: Each client sends max 100 requests



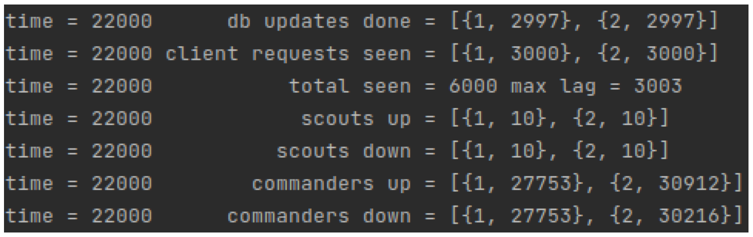
```
time = 22000      db updates done = [{1, 2989}, {2, 2989}]
time = 22000 client requests seen = [{1, 15000}, {2, 15000}]
time = 22000           total seen = 30000 max lag = 27011
time = 22000            scouts up = [{1, 12}, {2, 11}]
time = 22000          scouts down = [{1, 12}, {2, 11}]
time = 22000        commanders up = [{1, 24503}, {2, 24933}]
time = 22000      commanders down = [{1, 23517}, {2, 24933}]
```

Figure 13: Each client sends max 5000 requests

From the two results above, it can be seen that the less the max requests sends by a client, the less time the servers use to finish the work and less likely to reach to livelock.The complete output file for this testcase is in "TC6_2s_3c.txt" and "TC6_2s_3c.txt" respectively.

Appendix A

Livelock occurs for SERVERS = 2, CLIENTS = 3, window = 100



```
time = 22000      db updates done = [{1, 2997}, {2, 2997}]
time = 22000 client requests seen = [{1, 3000}, {2, 3000}]
time = 22000           total seen = 6000 max lag = 3003
time = 22000            scouts up = [{1, 10}, {2, 10}]
time = 22000          scouts down = [{1, 10}, {2, 10}]
time = 22000        commanders up = [{1, 27753}, {2, 30912}]
time = 22000      commanders down = [{1, 27753}, {2, 30216}]
```

Figure 14: Test Case 2: Livelock