Program Design

1. Architecture :

a. server.py –

i. This file contains the commander code and also maintains a centralized state of the battlefield and the alive/dead status of each soldier.

ii. The commander generates M - 1 separate soldier processes on the client side via RPC call (M being the no. of soldiers).

iii. The commander broadcasts the incoming missile info to all the soldiers via an RPC call to the client.

iv. The commander **periodically polls the soldiers** to check if they all (those who are alive) have executed taken shelter. Unless all of them have done so we do not fire the next missile.

v. The commander keeps track of the game time. To mitigate the time delay wasted in RPC calls, we use a **logical timestamp instead of a physical timestamp** to keep track.

vi. The commander also helps the soldiers to avoid conflict while taking shelter (two soldiers on the same position) via an RPC call from the client.

vii. After each missile is fired and all soldiers have taken shelter, a STATUS QUERY is sent to the client to know the status of each soldier (is dead/alive).

viii. If the commander is hit by the missile, we **elect a new commander** from the list of alive soldiers maintained by himself. We send the details of the new commander to the client via RPC and the corresponding soldier who is elected commander terminates their own process (as they are now running on the server side as commander).

ix. After the game time is over, we display a “Game Won” message if more than 50% of the soldiers remain alive.

b. client.py –

i. This file contains the code to be executed by each soldier.

ii. The soldiers are notified of the incoming missile via an RPC call from the commander.

iii. The incoming missile is stored in a shared multiprocessing queue which can be accessed by all the soldier processes.

iv. To access shared queues from the soldier processes we have used shared multiprocessing locks.

v. Each soldier executes take_shelter() procedure locally which consists of the following steps –

- The missile area is marked on an N X N grid.
- The same grid is again traversed but only on the area the soldier can jump to based on his capacity.
- If any missile impact point falls within this possible jump area of the soldier, that location is marked unsafe.
- Thus a list of safe jumping locations is constructed.
- Each soldier still needs to query the commander for a conflict free safe location (no 2 soldiers on the same location) as **the soldiers have no knowledge about each others' location.**
- After getting a conflict free safe location, the soldier jumps there.
- If no such location is available, the soldier dies and that corresponding process terminates.

vi. For the RPC query to get valid jump positions, each soldier pushes his RPC request onto a REQUEST QUEUE, which is constantly polled by the main client process and RPC requests are sent to the server in a FIFO manner. The response from the server is captured in a RESPONSE QUEUE.

vii. The soldiers who are alive continue to run perpetually unless an RPC call is made from the server to the client indicating Game Over.

2. RPC Descriptions :

a. ***get_params_client()*** **–** client.py -> server.py to get hyperparameters N,M which will be useful when each client executes their take_shelter().

b. ***create_soldiers()*** **–** server.py -> client.py to start M – 1 soldier processes on the client.

c. ***send_commander_index()*** **–** server.py -> client.py After commander re-election, the server sends the soldier number who is elected as the commander. Subsequently that particular soldier process terminates himself (as he is already running as commander).

d. ***missile_approaching()*** **–** server.py -> client.py Broadcasts the incoming missile details (missile coordinates, missile type) to all the soldier processes via the client.

e. ***get_valid_positions()*** **–** client.py -> server.py Each soldier sends a list of safe positions to be checked by the client for non-conflict (no other soldier on the same position). Server returns a single non-conflicting position to the client.

f. ***all_taken_shelter()*** **–** server.py -> client.py Server calls this RPC function to check if every soldier finished executing its local take_shelter().

g. ***status()*** **–** server.py -> client.py Server checks the status of each soldier (if hit or not) after firing each missile.

h. ***game_over()*** **–** server.py -> client.py When the Game Time is over, the server informs the client to terminate the execution of the remaining soldiers.

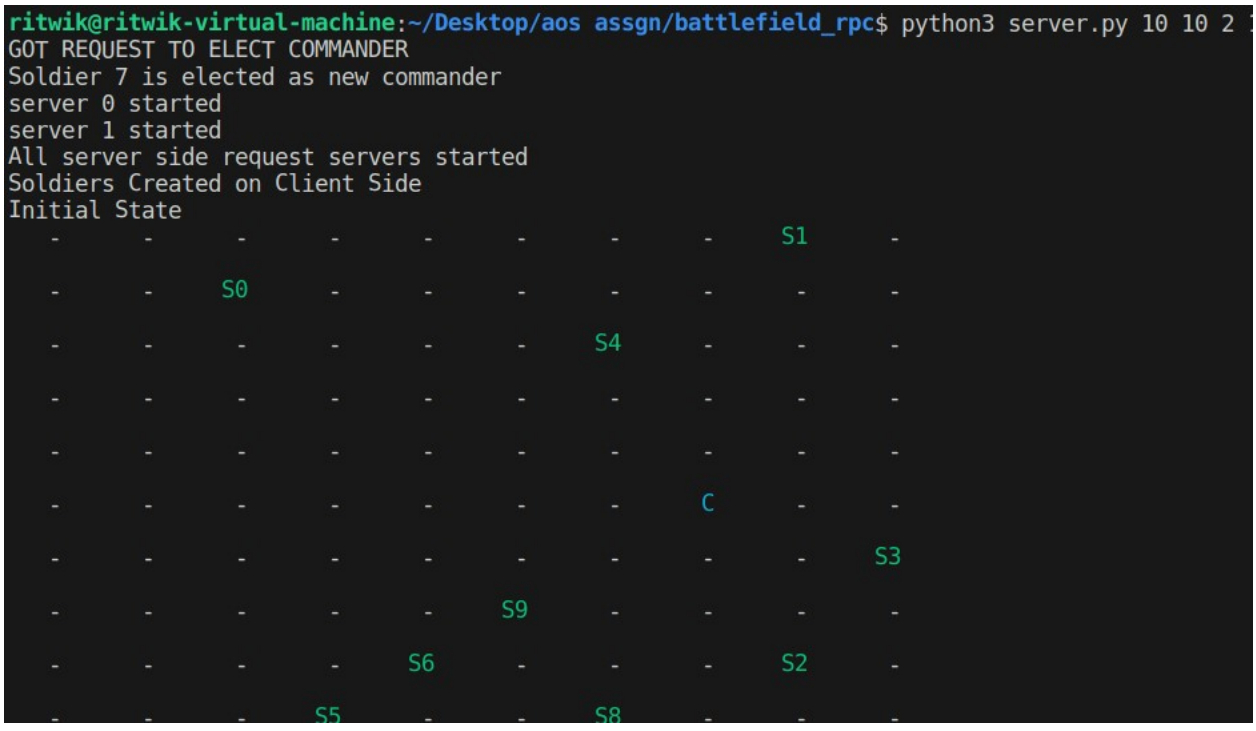
3. Design Tradeoffs Considered :

- We perpetually run the commander on the server side and is not “killed”; when the commander is about to be killed we elect a new commander from the list of available soldiers, start running that soldier on the commander and that soldier is killed on the client side.
- All the soldiers are on the same machine; the architecture of the game could have been implemented so that each soldier could be run on a different machine.
- Each soldier queries the commander for non-conflicting safe positions every time it executes take_shelter(). Drawback is a large number of extra messages exchanged (2 * no. of alive soldiers) each time a missile is fired. Advantage is that it ensures the constraint that no 2 soldiers occupy the same grid position.

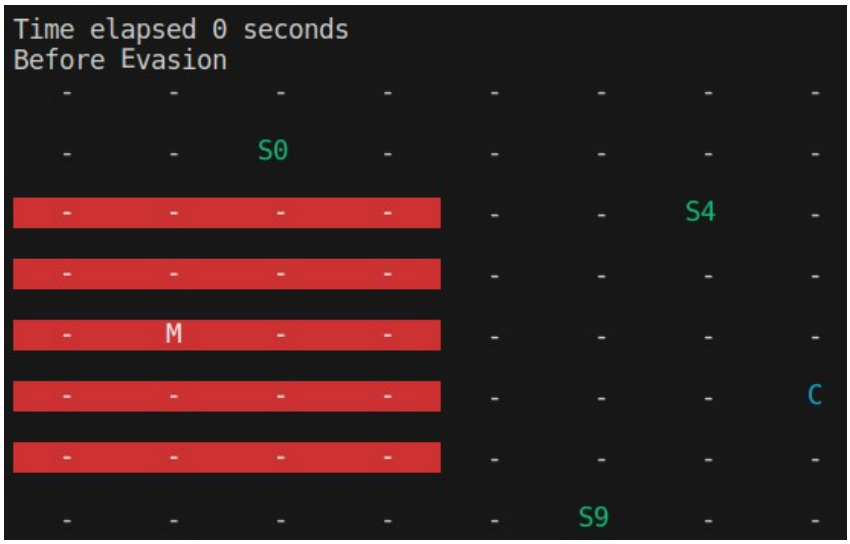
4. Test Cases :

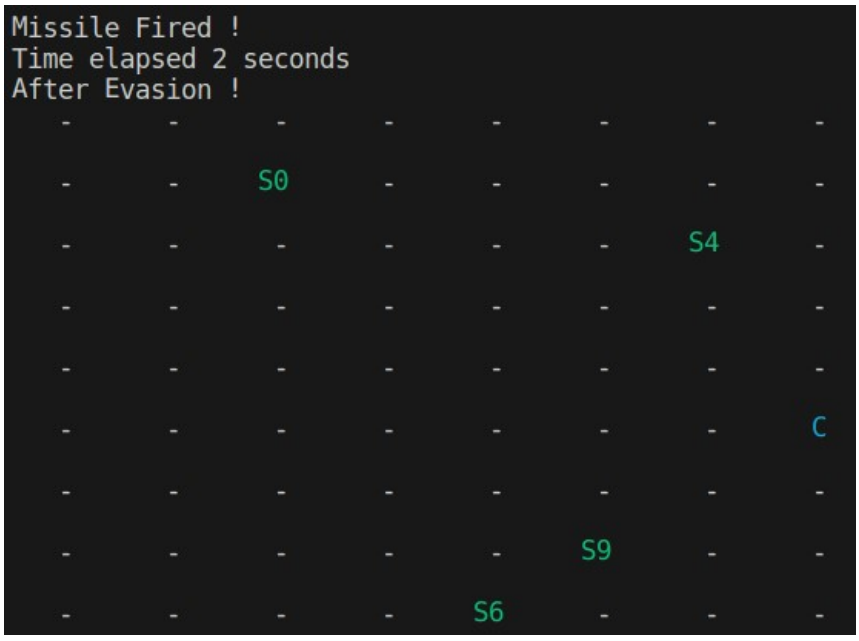
a. Grid = 10 X 10 , 10 Soldiers

Snapshot at beginning of the game :

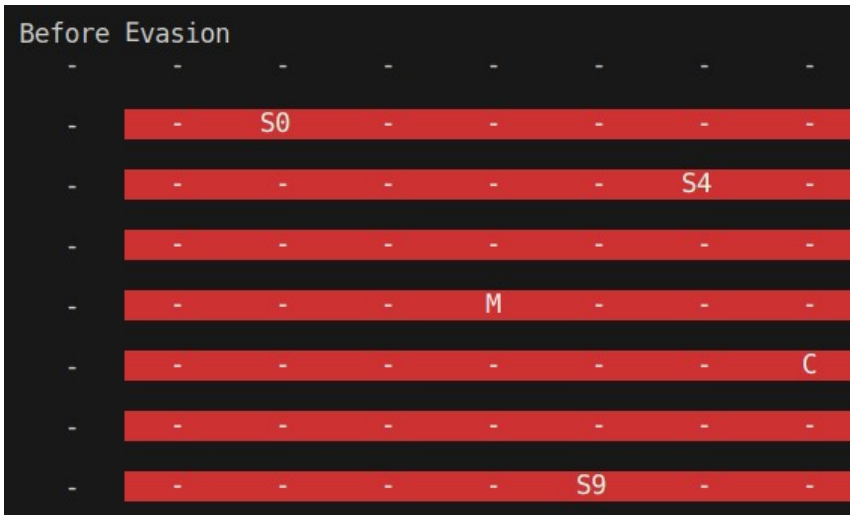


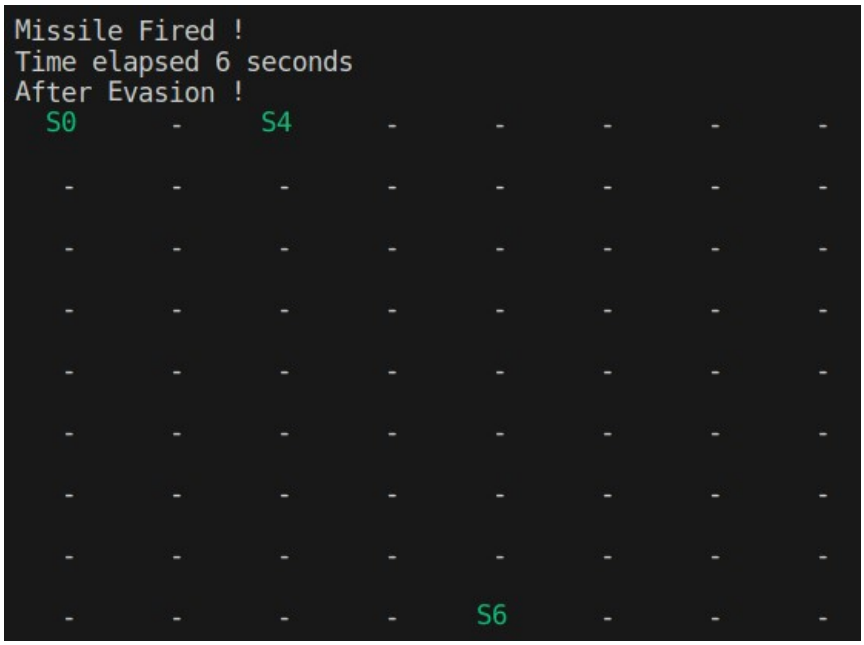
Snapshot after 1st missile fired :



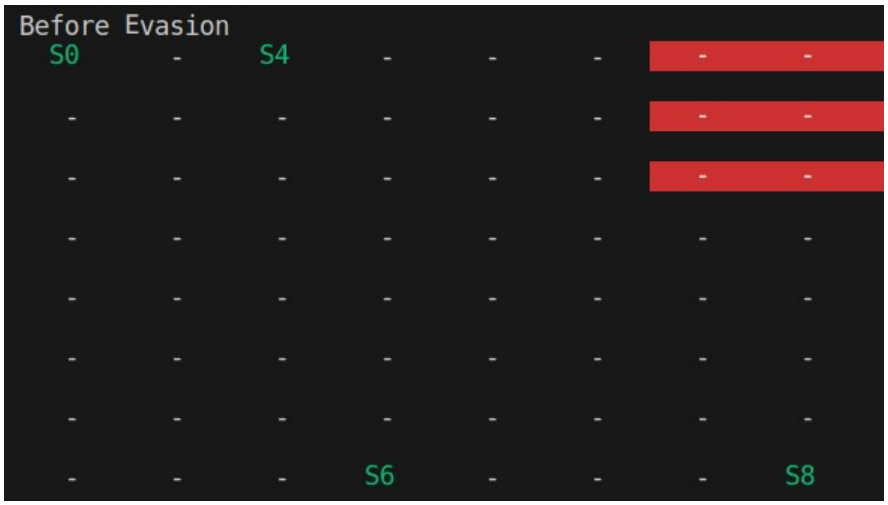


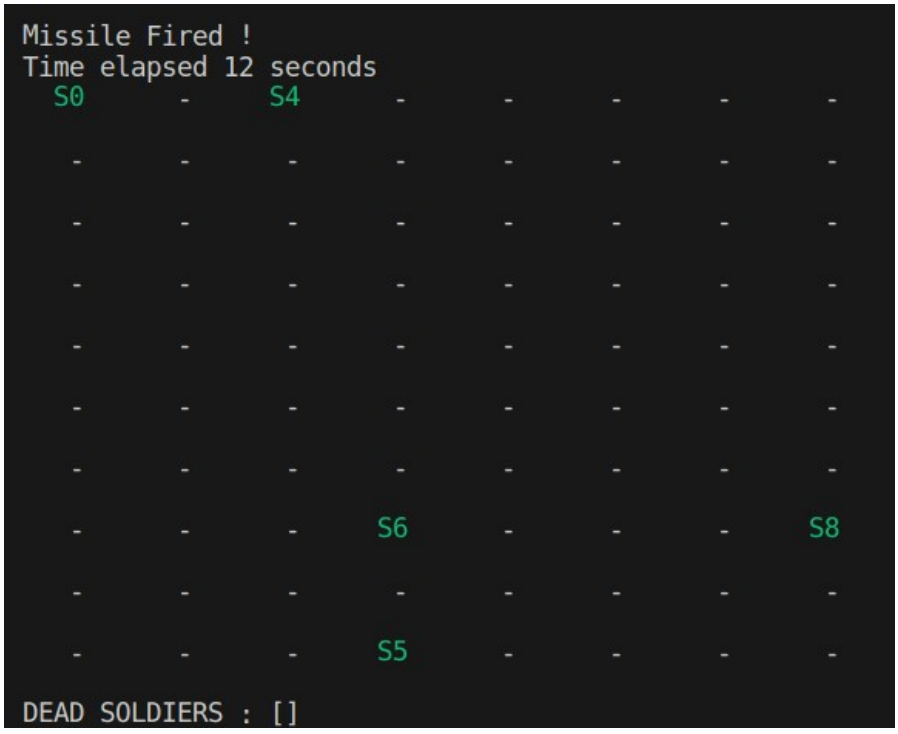
Snapshot at t = 6 secs :





Snapshot at t = 12 secs :

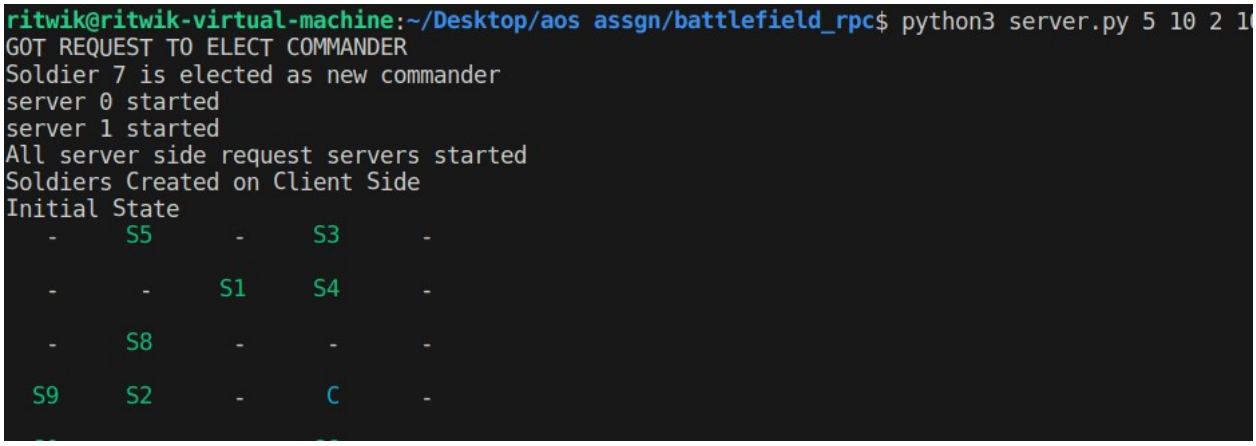




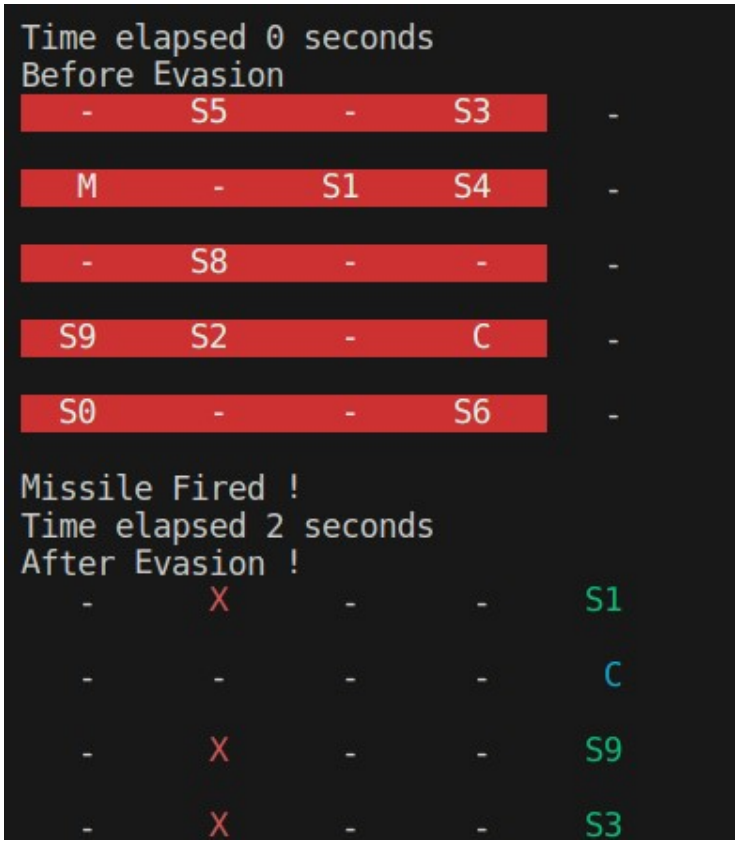
TEST CASE PASSED AS WE WERE ABLE TO SIMULATE AT LEAST 9 SOLDIERS AND ONE COMMANDER.
GAME WON AS EVERY SOLDIER WAS ABLE TO TAKE EVASIVE ACTION AFTER EACH MISSILE AND 100% SOLDIERS REMAINED ALIVE AT THE END.

b. Grid = 5 X 5 , 10 Soldiers

Snapshot at the beginning of the game :



Snapshot after 1st missile fired :

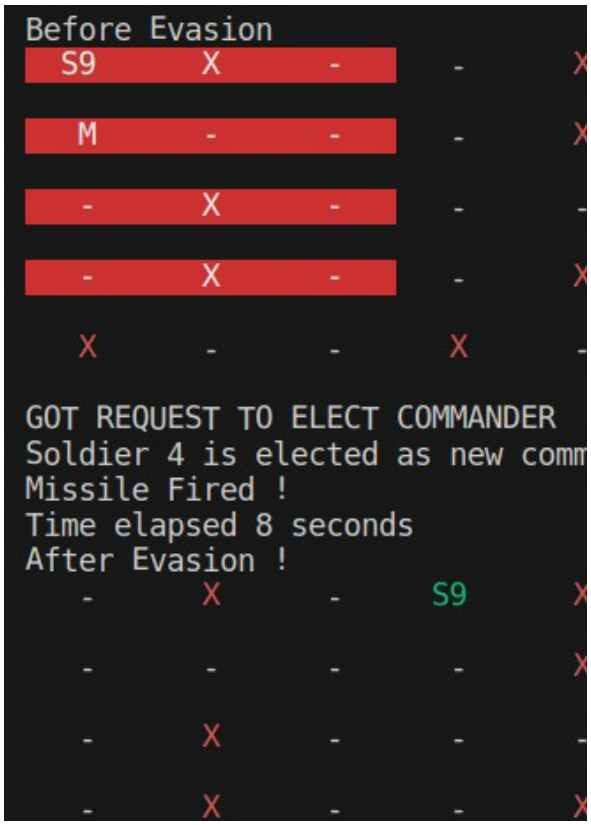


Here, we can see the 1st missile killed 5 soldiers and the same is reflected in the global state.

Snapshot at t = 4 secs :

Snapshot at t = 6 secs :

Snapshot at t = 8 secs :



Snapshot at t = 10 secs :

Snapshot at t = 12 secs :

TEST CASE PASSED AS WE DEMONSTRATED THE CASE WHEN A MISSILE HITS A SOLDIER AND HE IS MARKED DEAD IN THE GLOBAL STATE.
GAME LOST AS 10% SOLDIERS WERE REMAINING AT THE END OF THE BATTLE.