Program Design File

Content Table:

| SL No. | Content |
| --- | --- |
| 1. | Overview |
| 2. | Description Of Files and Functions Implemented |
| 3. | Program Flow |
| 4. | Flowchart of Program |
| 5. | Description Of Test Cases |
| 6. | Design TradeOffs |
| 7. | Steps to Run the Code |

**Overview**

The program essentially functions as a game with a set number of soldiers and one commander. There is also a specific number of grids, with the assumption that one soldier occupies each grid. The game operates over a series of iterations, which can be referred to as steps or time.

During each iteration, a missile with a defined radius approaches the battlefield, marking its coordinates as danger zones. The soldiers must make strategic moves based on their individual speeds, as provided by the user. If a soldier can successfully move and evade the incoming missile, they remain unharmed, particularly if the missile's proximity is beyond their zone of operation or if their speed exceeds the missile's radius. However, soldiers who fail to move in time, either due to a speed of zero or inadequate speed to escape, are considered 'hit.'

In summary, the game involves soldiers and a commander, grid-based positioning, and a series of iterations. The objective is for soldiers to avoid incoming missiles by utilizing their specified speeds. Those who manage to escape remain unHit, while others unable to move are marked as casualties."

Note: **At Every Iteration If any Commander is Hit then New Soldier is ELected as Commander at Random**

Description Of Files and Functions Implemented

In our program we have three files war.proto, War\_server.py and War\_client.py.

1. **war.proto**

This file contains the protocol buffer definition for gRPC service. It is used to define the services and message types used by the sever and client.

It defines:

* Messages: (1) SimulationRequest and (2) SimulationResponse
* Service : WarzoneSimulator which accepts SimaulationRequest and returns Simulationresponse.

1. **War\_client.py**

This is the Python snippet which implements the **gRPC client**.It is responsible for establishing connection with the server and taking required inputs from the user and prints out the results.

1. **War\_server.py**

It implements the gRPC server, it has 3 classes as Soldier, Warzone and WarzoneSimulatorServicer.

The Soldier class is used for creating soldiers which has the following attributes:

* ***soldier\_id***which uniquely identifies a soldier.
* ***x****:* x-axis coordinate of the soldier in the grid.
* ***y****:* y-axis coordinate of the soldier in the grid.
* ***N****(size of warzone)* for restricting the movements od soldier within bounds.
* ***speed****:* The speed of a soldier ranging from 0-4
* ***is\_alive***is a Boolean variable which is set to false if the soldier is within the radius of a missile. It is used for implementing the was\_hit interface mentioned in the problem.
* ***is\_commander***is also a Boolean which tells if the soldier is a commander or not.
* ***commander*** variable stores the reference of commander in each soldier.

The soldier class has following methods:

* **move()** : For changing the soldiers location in the warzone.
* **take\_shelter()** : Handles the selection of direction and movement of soldier if it is within the radius of a missile.

The warzone class is used for initializing the N\*N grid, i.e., warzone it accepts the values N = size of warzone and M = number of soldiers. It also keeps track of the casualties in a list. It has following methods:

* **initialize\_soldiers()** : creates soldiers
* **missile\_approaching()** : simulates a missile approaching the warzone
* **commander\_election()** : for the election of a new commander
* **check\_battle\_result()** : checks the outcome of whole war, i.e., if casualties are more than 50% the war is lost otherwise won.
* **print\_layout()** : prints an N\*N grid for showing the warzone at each time t.

The WarzoneSimulaterServicer class is responsible for communicating with the client side and has thee overall flow of the whole program. It has 2 methods: SimulateWar() and serve().

**Program flow:**

The client calls the run\_simulation() method and asks for the host and port in which the server is running. After that it asks the user for input of N, M, time and soldier\_speeds and passes this value by creating a gRPC request to the server and stores the response given by the server in a variable *response.*

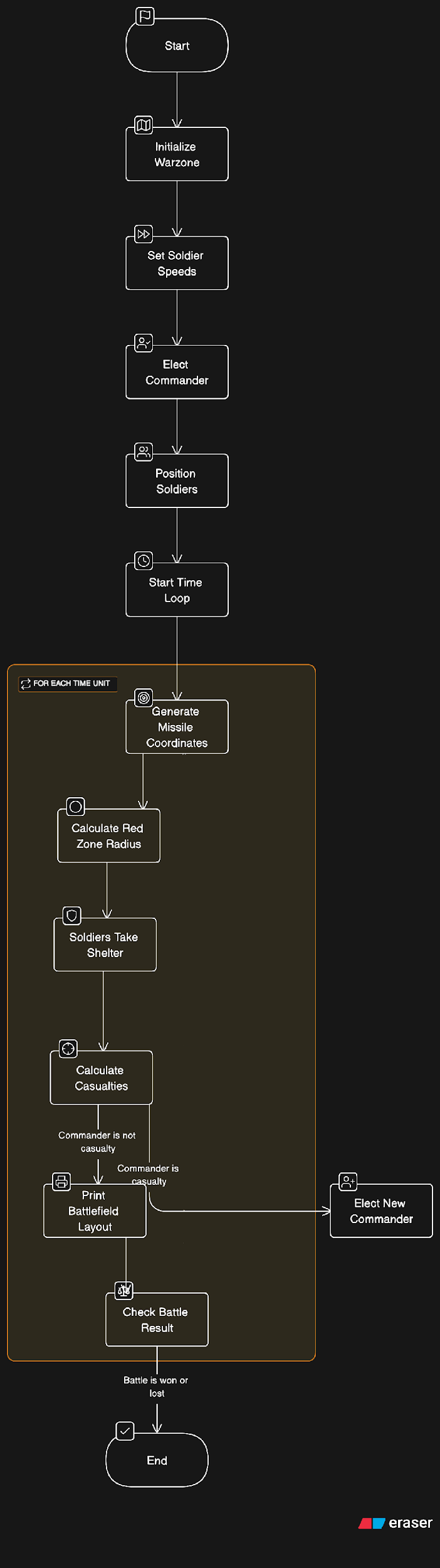
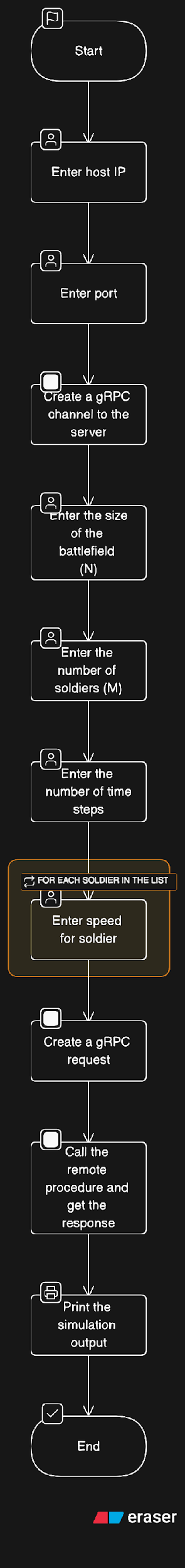
On the server side the SimulateWar() method accesses the values of N, M, time and soldier\_speeds and initializes the *warzone* class by passing the values of N and M to it, the constructor of warzone class calls initialize\_soldiers() method which creates the soldiers by assigning each soldier random *(x,y)* co-ordinates and initially sets all their speed to zero. After that we assign the value of speed of each soldier provided by the client to the soldiers. A commander is randomly elected at first.

Now we loop through the time, i.e., the total duration of war, provided by client to simulate each battle at time = t. For each t we initialize a missile which has *(missile\_x, missile\_y)* co-ordinates and a *red\_zone\_radius* which has a value in range 1-4. We call the missile\_approaching() method defined in the warzone class. The missile\_approaching() triggers the take\_shelter() method if any soldier is within the *red\_zone\_radius* of the missile. The take\_shelter() method checks for the commander reference in the soldier for mimicking the communication between commander and soldiers. Now the commander has passed the values of missile coordinates and radius to all the soldiers the soldier takes appropriate steps according to their respective speeds to safeguard them from the incoming missile. If the soldier failed to escape the missile it’s *is\_alive* value is set to False indicating it was hit by the missile and are added to the casualties list. Now if the soldier which died during this battle included the commander a new commander is elected by calling commander\_election() method of warzone class.

We initialize a list named *simulation\_output* for storing all the results which are to be printed on the client side and pass it as a response. The ouput generated by the print\_layout() method is also appended in this list. After the total duration of war is finished i.e., t = time, we check the result of war by calling the check\_battle\_result() method which checks if the number of casualties are more than 50% of the original number of soldiers.

Finally the *simulation\_output* is passed to the client and the client displays the ouput.

**Flow diagrams:**

Fig**: Flowachart of Program** 

**Test case description :**

Client will provide the following input :

Enter the size of the battlefield (N): 10

Enter the number of soldiers (M): 10

Enter the number of time steps: 5

Enter speed for soldier 1: 2

Enter speed for soldier 2: 1

Enter speed for soldier 3: 0

Enter speed for soldier 4: 4

Enter speed for soldier 5: 3

Enter speed for soldier 6: 1

Enter speed for soldier 7: 1

Enter speed for soldier 8: 0

Enter speed for soldier 9: 4

Enter speed for soldier 10: 4

Once the client provides the input, all computation will be done at the server end and the result will be printed on the client-side. For more insight on the movement of soldiers in each iteration we have printed the movements on the server side.

Client side output:

**Here the commander is soldier 9. (The indexing starts from 0)**

Commander is soldier 9

Soldier 1 is at (4,4)

Soldier 2 is at (8,2)

Soldier 3 is at (5,5)

Soldier 4 is at (6,8)

Soldier 5 is at (7,5)

Soldier 6 is at (7,4)

Soldier 7 is at (4,2)

Soldier 8 is at (1,5)

Soldier 9 is at (3,4)

Soldier 10 is at (0,3)

Time 1: Missile approaching (4, 5) with radius 1

. . . 10 . . . . . .

. . . . . 8 . . . .

. . . . . . . . . .

. . . . 9 . . . . .

. . 7 . 1 X . . . .

. . . . . 3 . . . .

. . . . . . . . 4 .

. . . . 6 5 . . . .

. . 2 . . . . . . .

. . . . . . . . . .

Casualties: []

**As we can see above in iteration-1 the missile lands at (4,5) and soldier 1 is at (4,4) just besides the missile but it doesn’t die as the missile had radius = 1.**

------------------------------------------------------------------

Time 2: Missile approaching (1, 2) with radius 2

. . . . . . . . . .

. . X . . 8 . . . .

. . . . . . . . . .

. . . . 9 . . . . .

. . 7 . 1 . . . . .

. . . . . 3 . . . .

. . . . . . . . 4 .

. . . . 6 5 . . . .

. . 2 . . . . . . .

. . . . . . . . . .

Casualties: [10]

**This time the soldier 10 was in radius of missile so it moved to (4,0) as printed on the server side but could not avoid the missile and died, hence was added to the casualties list**

------------------------------------------------------------------

Time 3: Missile approaching (3, 2) with radius 3

. . . . . . . . . .

. . . . . 8 . . . .

. . . . . . . . . .

. . X . 9 . . . . .

. . . . . . . . . .

. . . . . 3 . . . .

. . . . . . . . 4 .

. . . . 6 5 . . . .

. . 2 . . . . . . .

. . . . . . . . . .

Casualties: [10, 1, 7,9]

New commander is soldier 4

**Here soldier 9 which was a commander died so new commander was elected which came out to be soldier 4**

------------------------------------------------------------------

Time 4: Missile approaching (0, 0) with radius 2

X . . . . . . . . .

. . . . . 8 . . . .

. . . . . . . . . .

. . . . . . . . . .

. . . . . . . . . .

. . . . . 3 . . . .

. . . . . . . . 4 .

. . . . 6 5 . . . .

. . 2 . . . . . . .

. . . . . . . . . .

Casualties: [10, 1, 7, 9]

------------------------------------------------------------------

Time 5: Missile approaching (0, 2) with radius 4

. . X . . . . . . .

. . . . . . . . . .

. . . . . . . . . .

. . . . . . . . . .

. . . . . . . . . .

. . . . . 3 . . . .

. . . . . . . . 4 .

. . . . 6 5 . . . .

. . 2 . . . . . . .

. . . . . . . . . .

Casualties: [10, 1, 7, 8, 9]

**There are 5 casualties which is exactly 50% of the original soldiers hence the battle was declared as won.**

------------------------------------------------------------------

The battle was won!

Server Side Output:

War Started!

Current time: 1 **(No soldier movement)**

Current time: 2

Soldier 10 moved to (4,0) **(As mentioned previously the soldier 10 was in radius of missile so it moved)**

Current time: 3

Soldier 1 moved to (2,2)

Soldier 7 moved to (3,3)

Current time: 4

Current time: 5

War ended!

**Design Tradeoffs**

In the above solution the implementation of the information exchange between soldiers and the commander is not implemented exactly as commander giving the missile information to the soldiers the program just behaves like it does that.

Another tradeoff is that we are printing the grid in which we are only showing the aftermath of the battle, meaning the positions of soldier after the missile landed which does not capture the movement of soldiers but to compensate that we print the soldier’s movements on the server-side.

**Steps to run the code**

To run this code on the same machine, add all the 4 files, i.e., War\_client.py, War\_server.py, war\_pb2.py and war\_pb2\_grpc.py in the same directory. Install all the necessary modules mentioned in the **readme.txt** file and run the client and server code on separate terminals. Both will ask for **host** and **port**, enter “localhost” as host and any port of your choice which must be free, e.g., 8000.

For running the code on different machines follow these steps:

**Step 1:** Install all modules and dependencies mentioned in the **readme.txt** file on both the machines.

**Step 2:** Make sure both machines are on the same wifi network.

**Step 3:** On the server machine run War\_sever.py file and enter IP address of this machine in **host** as input and give port number (e.g. 8000). For ending the server give a keyboard interrupt (usually **ctrl+C**)

**Step 4:** On the client machine run War\_server.py file and enter the same IP address and port. Give all necessary inputs and the code should give result.