Program Design Document

GitHub - https://github.com/shubhs1306/Battlefield-Simulation

Table of Contents :

Team Members:

Shubham Shrivastav (2022H1120283P)
Sparsh Kumar (2022H1120293P)

Overview:

This program simulates a battlefield scenario where a commander communicates with soldiers on the battlefield. Missiles are launched at the battlefield at specified intervals. The commander alerts soldiers about the incoming missiles, retrieves the status of all soldiers, and displays the current battlefield layout.

Components :

The system has three components namely battle.proto, battle_soldier.py and battle_commander.py.

1) battle.proto

- It uses proto3 syntax.
- The Soldier service has multiple RPCs (Remote Procedure Calls) like setting the battlefield size, setting soldier numbers, getting soldier positions (both X and Y coordinates), actions like taking shelter from incoming missiles, and checking if the soldier was hit.
- Several messages are used to pass information:
    - Request: Used for sending an ID or numeric data.
    - IncomingMissile: Used for sending missile details (its position and type).
    - StatusReply: Used for returning a status (e.g., whether a soldier was hit).
    - PositionReply: Used for returning a position (either X or Y coordinate).

2) battle_soldier.py

The battle_soldier.py file is responsible for defining and managing soldier entities within a simulated battlefield. Using gRPC, the script allows operations to be conducted related to individual soldier entities, such as initialization, movement in response to missiles, and checking statuses.

Modules and Libraries:

- concurrent.futures: Facilitates the creation of a thread pool, useful for concurrent operations but we only allow one client to connect to the server at a time.
- random: Used for generating random numbers, essential for attributes like the soldier's speed or initial position.
- battle_pb2_grpc & battle_pb2: Essential gRPC components that define the Soldier services and the protocol buffers respectively.
- grpc: The core gRPC module that allows the server's creation and operation.
- google.protobuf.empty_pb2: Represents an empty message for those RPC methods which do not return data.

Classes:

Soldier (inherits from battle_pb2_grpc.SoldierServicer):

Attributes:

- id: A class-level attribute for generating unique soldier IDs.
- all: A list that holds references to all instantiated soldier objects.
- N: Stores the battlefield's width/height.
- M: Keeps a count of the total soldiers.

Methods:

- __init__: Responsible for initializing soldier attributes.
- SetBattleFieldSize: Takes a request input to define the battlefield's size and randomly position the soldiers within it.
- SetSoldierNum: Sets the required number of soldiers on the field.
- take_shelter: Based on missile threats, it determines safe zones and directs soldiers accordingly. Soldiers with no safe moves are marked dead.
- was_hit: Checks whether a particular soldier was hit by a missile.
- GetPositionX & GetPositionY: Methods to retrieve the X or Y coordinate of a soldier.
- getSoldier: A static method that fetches a soldier instance based on an ID.
- __repr__: Gives a readable representation of a soldier's object, showcasing its status and position.

Functions:

- serve( ): Initiates the gRPC server, sets the address and port (eq. localhost:50051), adds the Soldier service, and listens to requests and routes them appropriately. It can be configured to listen to requests either locally or from a defined IP address.

Execution:

- When the script is executed, the serve function is triggered, initiating the gRPC server.

Special Notes:

- The terminal output shows all the created soldiers with their ID, position and speed.
- Soldiers are given a 'speed' attribute, determining how many grid spaces they can move in a single turn.
- The missile's area of impact is calculated, and soldiers are moved out of harm's way if possible using the take_shelter method.
- Before missile threats are simulated, it's assumed that the battlefield dimensions and soldier counts are predefined.
- The gRPC server's listening address can be toggled between localhost and a specific IP, depending on the requirement.

3) battle_commander.py:

This file is responsible for simulating a battlefield environment where commanders and soldiers are dynamically engaged with incoming missile threats. Using gRPC for communication, it monitors incoming missiles and soldiers, and keeps logs of battlefield events.

Modules and Libraries:

- random: For random number generation.
- grpc: To facilitate RPC communication.
- battle_pb2 and battle_pb2_grpc: gRPC generated client and server classes from the protocol buffers definition.
- logging: To maintain logs of battlefield activities.

- Program Components:

Global Variables:

- flag: Monitors the battlefield's status.

Classes:

- Commander

Attributes:

- alive_soldiers: A dictionary maintaining the positions and ID of alive soldiers.
- missile: Stores the coordinate details of incoming missiles.
- stub: The gRPC stub to interact with the server.

Methods:

- __init__( ): Initializes the gRPC stub and positions of all soldiers. Selects a random commander.
- missile_approaching( ): Alerts soldiers of an incoming missile and logs the event.
- status_all( ): Updates the status of all soldiers post-missile strike and possibly selects a new commander if the previous one was hit.
- printLayout( ): Renders the current state of the battlefield, showing soldiers' and missile positions.

Execution:

- When the script is executed, the run function is triggered which first connects to the gRPC server. Gets battlefield parameters like size, number of soldiers, missile launch intervals, and total simulation time from the user via command line input. Initiates a Commander and finally runs a loop for the simulation time provided by the user in which it launches missiles at defined intervals, checks soldier status, and prints the battlefield layout.

Special Note:

- Every activity and status is recorded onto the output_log.txt log file using the logging library in python.

How it works

1. A battlefield size (N), the number of soldiers (M), the time interval at which missiles are launched (t) and the total simulation time (T) are determined from user inputs.
2. Battlefield size and number of soldiers are communicated to the server.
3. A battlefield of size NxN is created and M soldiers are placed randomly on the battlefield each with random speed.
4. A commander is created from a randomly selected soldier.
5. Every t interval, a missile is launched at a random location. The missile has a certain type, which might denote its blast radius or damage capability.
6. The commander communicates this information to the soldiers
7. For each soldier take_shelter is called and soldiers within the missile's range move to avoid being hit.
8. The commander checks the status of each soldier, removes dead soldiers from the battlefield, and updates the positions of surviving soldiers.
9. If at any point, the commander dies, a random soldier becomes the new commander.
10. If all soldiers die, the game ends with a loss. Otherwise, the game continues until the simulation time T expires.
11. When the timer expires, if more than 50% of soldiers are alive then the game is won otherwise lost.
12. After every missile launch, the game prints a detailed information of all alive soldier's positions, and prints them nicely on a 2D Matrix where each alive soldier is represented by its ID and the spot where missile hit is marked with '**X**'

Design Tradeoff Considered

1. All the soldier related information is maintained only on the server including the commander's, to satisfy the Single Responsibility principle, as all the logic for taking shelter is present in the server only.
2. The commander has only a soldier's ID, signifying which soldier is currently a commander. The commander communicates all the missile information to all the soldiers and also query their statuses.

3. We only allow 2 machines to communicate at a time. One which acts as a server running the soldier program and the other as a client running the commander program.

Test Cases and validation

Test case 1 :

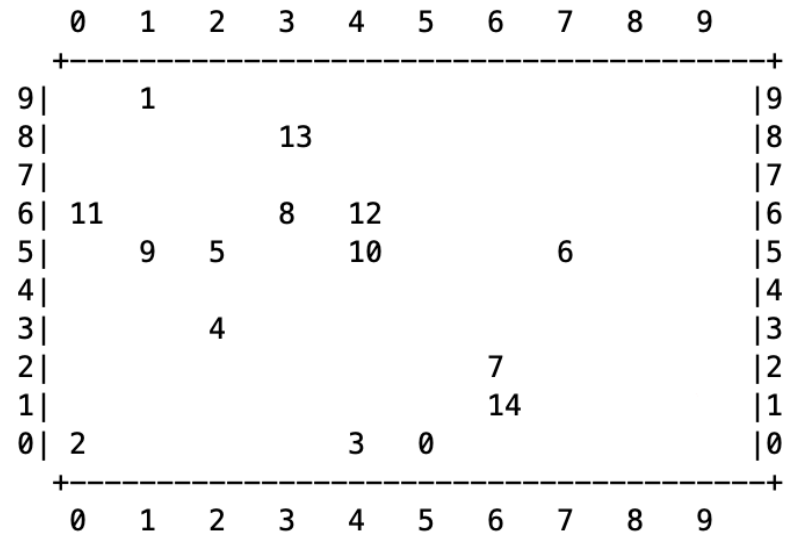
Inputs:

Size of Battlefield : 10
Number of Soldiers : 15
Missile launch interval : 2
Simulation time : 10

Outputs:

Activity at soldier (server side):
Created a new soldier id - 0 with position 5,0 and with speed 1.
Created a new soldier id - 1 with position 1,9 and with speed 2.
Created a new soldier id - 2 with position 0,0 and with speed 2.
Created a new soldier id - 3 with position 4,0 and with speed 3.
Created a new soldier id - 4 with position 2,3 and with speed 1.
Created a new soldier id - 5 with position 2,5 and with speed 2.
Created a new soldier id - 6 with position 7,5 and with speed 2.
Created a new soldier id - 7 with position 6,2 and with speed 3.
Created a new soldier id - 8 with position 7,2 and with speed 4.
Created a new soldier id - 9 with position 1,5 and with speed 4.
Created a new soldier id - 10 with position 4,5 and with speed 2.
Created a new soldier id - 11 with position 0,6 and with speed 2.
Created a new soldier id - 12 with position 4,6 and with speed 3.
Created a new soldier id - 13 with position 3,8 and with speed 0.
Created a new soldier id - 14 with position 6,1 and with speed 1.
Total number of soldiers is 15

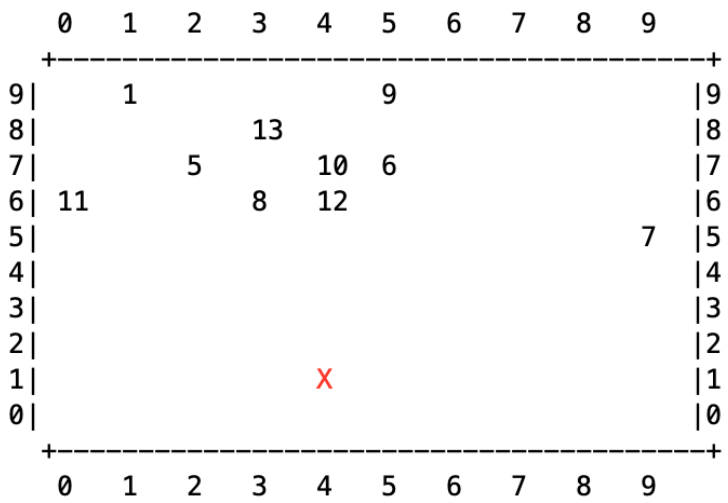
Representation in matrix form :



Activity at commander(client side):

New Commander's ID is 8
A Missile to hit at location (4,1) which is of type - 4
Status of soldiers after missile hit :
Soldier 0 got Hit!
Soldier 1 is still at (1,9)
Soldier 2 got Hit!
Soldier 3 got Hit!
Soldier 4 got Hit!
Soldier 5 has moved from (2,5) to (2,7)
Soldier 6 has moved from (7,5) to (5,7)
Soldier 7 has moved from (6,2) to (9,5)
Soldier 8 is still at (3,6)
Soldier 9 has moved from (1,5) to (5,9)
Soldier 10 has moved from (4,5) to (4,7)
Soldier 11 is still at (0,6)
Soldier 12 is still at (4,6)
Soldier 13 is still at (3,8)
Soldier 14 got Hit!

Representation in matrix form :



No of alive soldiers: 10
No of dead soldiers: 5

Test case 2:

Inputs:
Size of Battlefield : 8
Number of Soldiers : 16
Missile launch interval : 2
Simulation time : 10

Outputs:

Activity at soldier (server side):

Created a new soldier id - 0 with position 0,6 and with speed 2.
Created a new soldier id - 1 with position 7,6 and with speed 2.
Created a new soldier id - 2 with position 1,2 and with speed 0.
Created a new soldier id - 3 with position 1,2 and with speed 4.
Created a new soldier id - 4 with position 0,0 and with speed 3.
Created a new soldier id - 5 with position 0,6 and with speed 3.
Created a new soldier id - 6 with position 1,6 and with speed 4.
Created a new soldier id - 7 with position 4,4 and with speed 3.
Created a new soldier id - 8 with position 4,4 and with speed 3.

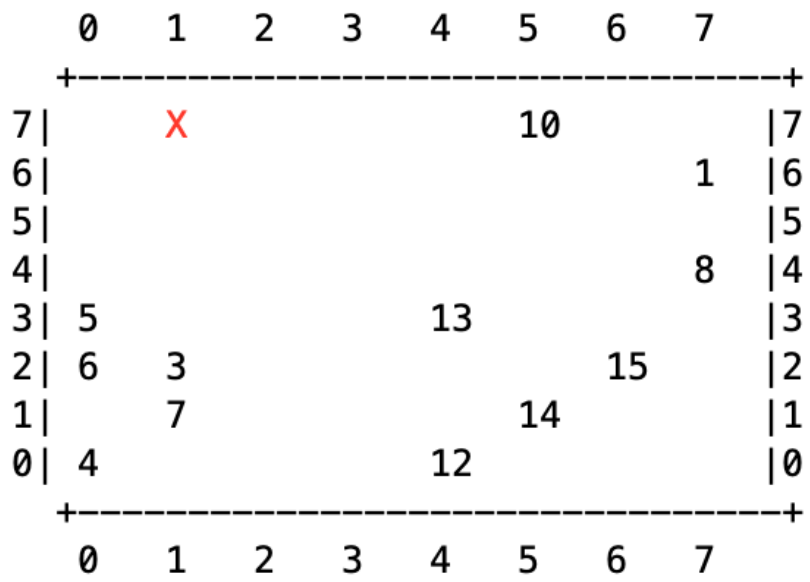
Created a new soldier id - 9 with position 5,1 and with speed 2.
Created a new soldier id - 10 with position 5,7 and with speed 0.
Created a new soldier id - 11 with position 3,0 and with speed 1.
Created a new soldier id - 12 with position 4,4 and with speed 4.
Created a new soldier id - 13 with position 4,7 and with speed 4.
Created a new soldier id - 14 with position 5,1 and with speed 4.
Created a new soldier id - 15 with position 6,4 and with speed 2.
Total number of soldiers is 16

Activity at commander(client side) :

New Commander's ID is 0
A Missile to hit at location (1,7) which is of type - 3
Getting status of all alive soldiers -
Soldier 0 got Hit!
Soldier 1 is still at (7,6)
Soldier 2 is still at (1,2)
Soldier 3 is still at (1,2)
Soldier 4 is still at (0,0)
Soldier 5 has moved from (0,6) to (0,3)
Soldier 6 has moved from (1,6) to (0,2)
Soldier 7 has moved from (4,4) to (1,1)
Soldier 8 has moved from (4,4) to (7,4)
Soldier 9 is still at (5,1)
Soldier 10 is still at (5,7)
Soldier 11 is still at (4,0)
Soldier 12 has moved from (4,4) to (4,0)
Soldier 13 has moved from (4,7) to (4,3)
Soldier 14 is still at (5,1)
Soldier 15 has moved from (4,4) to (6,2)

Commander is dead!
New Commander's ID is 14

Representation in matrix form :



From the above test cases, we can validate that:

1) The messages are being communicated from client i.e commander to the server i.e soldier and vice versa.
2) The soldiers are being created with random positions and speed.
3) Missile alerts are being sent by the commander at regular intervals and received by the soldiers.
4) The soldiers act on the alert and try to move to all the possible coordinates with respect to their speed if they are within the blast zone.
5) Dead soldiers are removed from the battlefield.
6) If the commander dies a random commander is chosen from the alive soldiers.