

Assignment 1

CS G623 - Advanced Operating Systems

Airstrike

https://github.com/coscotuff/Airstrike_RPC

-By

Shashank Shreedhar Bhatt - 2020A7PS0078P

Rohan Srinivasan - 2020A7PS0081P

Assignment 1	1
Introduction	3
About Airstrike	4
Dependencies	4
Design Choices	4
Gamification	4
Agents	5
Soldiers (and Commander):	5
Player Movement:	5
Lives:	5
Selecting a commander: Promotion	5
Connector	5
Player Speed and Missile Type	6
Connection Initialisation	6
Time	7
Turns	7
Total Game Duration	7
Duration between attacks	7
Movement	7
Extra role of commander	8
UI Choices	8
Golden For Commander	8
Red for zone	8
Inputs	8
Messages	8
Result Convey	8
Initialisation and Usage	9
Running	9
1) Single device version:	9
2) Two device version:	9
3) Three device version:	9
4) $2 \cdot M + 1$ device version:	10
Components	11
Player Connector	11
Overview	11
Code Structure	11
Key Components	11
1) Server Class	11
2) RPC Functions	12
Commander and foot soldier connectors	13

Overview	13
Key Components	13
1) Server Class	13
2) Server Initialization	14
3) gRPC Services	14
4) GUI Integration	14
Protocol Buffers	15
soldier.proto	15
Message Definitions:	15
1) SoldierData	15
2) Position	15
3) RedZone	15
4) AttackStatus	15
5) SoldierStatus	15
6) Battalion	16
7) WarResult	16
8) void	16
Service Definitions:	16
1) Alert	16
connector.proto	16
Message Definitions:	16
1) Coordinate	16
2) MissileStrike	16
3) Hit	17
4) Points	17
5) Timestamp	17
Service Definitions:	17
1) PassAlert	17

Introduction

Welcome to the technical documentation for "Airstrike," an exhilarating multiplayer game that brings together strategic warfare and real-time combat.

Airstrike utilizes a robust network of communication facilitated by four key files: `player_connector.py`, `commander_foot_connector.py`, `soldier.proto`, and `connector.proto`. This documentation serves as your gateway to understanding the intricacies of these protocols, empowering you to dive into the heart of the game's mechanics and architecture.

About Airstrike

"Airstrike" is a dynamic multiplayer game that immerses players in a high-stakes battlefield where tactical decision-making and coordinated teamwork reign supreme.

As a player, you assume one of two pivotal roles: that of a Commander or a Soldier, each with distinct responsibilities and objectives. Your mission is to engage in strategic warfare, coordinate missile strikes, protect your team, and ultimately lead your side to victory.

Dependencies

Before delving into the code, it's important to note the dependencies required to run this script successfully. The code relies on the following libraries and technologies:

- Python 3.x
- gRPC (Google Remote Procedure Call): gRPC tools and gRPC Python Library
- TKinter package (sudo apt-get install python3-tk)
- Protobuf 'protoc' compiler (for .proto files)
- Threading Python Library
- Random Python Library
- OS Python Library
- Socket Python Library
- Logging Python Library
- Time Python Library
- Socket Python Library
- Protobuf Python Library

Please ensure that you have these dependencies installed before running the script.

Design Choices

Gamification

Reading the original design specifications, we decided to make it into a Game with two teams where each team (the commander) can initiate attacks on the opposite team. We were inspired by the classic game **Battleships**. Each Team can be on a single system or on multiple systems. We call a team a **Player** and each member of the team is a **Soldier**.

Agents

Soldiers (and Commander):

We have designed a battalion of soldiers, each using a unique port number. One of the soldiers is also the Commander. The Connector of a team launches a missile via the RPC **SendAlert()** function of the opposing team's Connector which in turn calls an RPC called **SendZone()**. The commander gets to know of the incoming missile strike through the Connector via the RPC **SendZone()**. Then the commander conveys this information to itself via a local function **move()**. It conveys it to the remaining soldiers of the battalion via an RPC **UpdateStatus()**. **UpdateStatus()** in turn calls **move()**. This is responsible for the movement logic.

We considered the alternative of all soldiers being informed directly by the connector, but we wished to keep the commander involved as much as possible. Also, the connector does not have access to which particular soldiers are alive at a given point in time. This information lies solely with the current commander thus ensuring loose coupling.

Player Movement:

For player movement, initially, we had a fixed logic where the player would move away from the centre of the attack if it was in the RedZone. However, in order to make it interactive, we now allow the user to input the location where it should move. This also makes it less predictable for other systems present and more game-like.

Lives:

We have added 3 lives to each soldier in order to not make it too boring for each individual player.

Selecting a commander: Promotion

The initial commander is chosen by the Connector (now randomly). If the commander dies, it appoints a new commander and returns the new commander to the Connector. The RPC **Promote** is used to promote a soldier. Each call to **Promote** informs the promoted **Commander** about the details of the Battalion. This battalion initially consisted of only the node numbers, as the port of each player was deterministic. However,

we later realised that in the case of multi-systems, different soldiers would have different IP addresses. Thus, we added the IP and Port into the Soldier message in the soldier proto.

When a commander dies, it randomly selects one of the living soldiers and promotes it to be the new commander, sending it the updated battalion. This models a field promotion that takes place in real armies.

Connector

We have modelled the missile system as a connector (or server). An external agent would communicate (like conducting a missile strike) via the connector. The connector is connected to the entire battalion of soldiers. However, for most of the game, it only communicates with the Commander and the Opposition Connector. We have designed the two Connectors to be running on the same system (i.e. the Server) and the clients could connect to either of them as they are on different port numbers.

While we considered keeping the Connector and Commander in a single system, we believe that having the same file to be run for all the soldiers with conditional blocks for the commander is more representative of the actual game scenario where the team would communicate with the server (connector) via the Commander.

Player Speed and Missile Type

As per the specifications, Speed and Radius both are in the range $[0, 4]$. However, we realised that this meant that a soldier with speed 4 could always escape from the missile, even when it was targeted at its exact coordinates. This would have made it invincible. In order to counter this, we have added another type of missile Radius (Type) 5 which can hit even the fastest soldier.

For player speed, we have chosen to randomly initialise it since the player would always choose the best speed, i.e., 4. This element of randomisation would result in more interesting gameplay. In case the invigilator wishes to customize the player's speed this can be done by editing the line number 64 in the file

`commander_foot_connector.py`.

```
61 # Initiliasie soldier position and speed
62 self.x = random.randint(0, self.N - 1)
63 self.y = random.randint(0, self.N - 1)
64 self.speed = random.randint(0, 4)
65 # self.speed = 1
```

Connection Initialisation

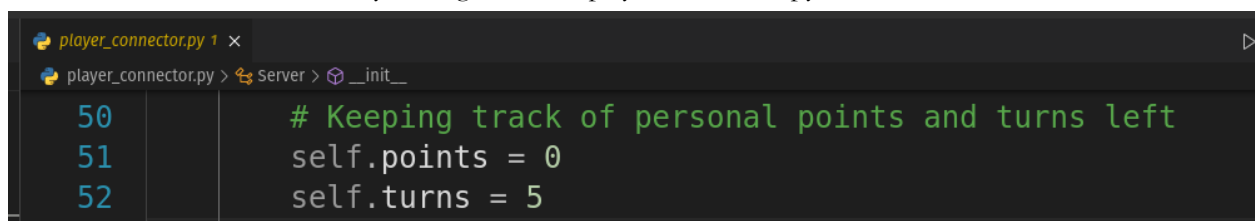
Initially, the connectors are running. Each soldier registers itself with the connector. Once all the soldiers of the team have registered with their respective connector, the connector informs the opponent connector that all of its players are here. The connector also sends its local timestamp. When received by the other connector, if at this point it has not achieved full quota, it does nothing. When it achieves quota, it will communicate to the opposition port with a greater timestamp (set statically to opposition timestamp plus 1)

allowing the one that happened before to go first. If the two `RegisterEnemy` calls were concurrent, then we simply perform a timestamp comparison and allow the one with a lower timestamp to go first. While this may not be accurate, the function is to simply be as fair as possible without undergoing a significant overhead of synchronisation. Only in the case of concurrent events do we allow the deterministic choice to be made as initiation by the player with the smaller timestamp.

Time

Turns

We have modelled the game as a series of turns. This is initiated to be 5 in order to have 'quick' games. However, this can be modified by editing line 52 in `player_connector.py`

A screenshot of a code editor window titled 'player_connector.py 1'. The editor shows the following code:

```
50 # Keeping track of personal points and turns left
51 self.points = 0
52 self.turns = 5
```

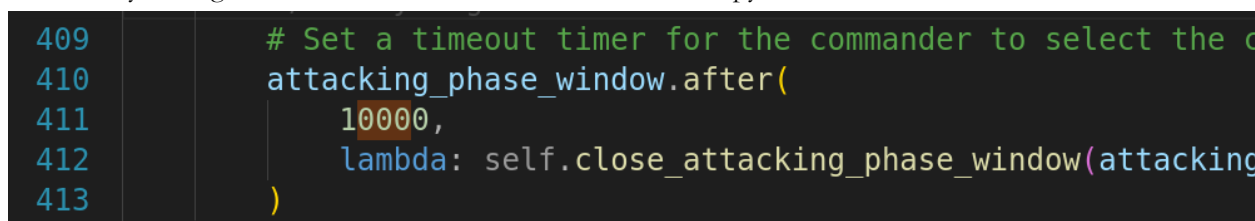
Also, continuing our inspiration from Battleships, we increment the number of turns by 1 for each hit.

Total Game Duration

Total game duration is the time taken for all turns of both teams to be exhausted or for either team to lose all its members.

Duration between attacks

For each attack the initiating commander is given some time to decide the attack, after which it is chosen randomly if the commander did not make a choice. Currently this is set to 10 seconds, however this can be modified by editing line 411 in file `commander_foot_connector.py`.

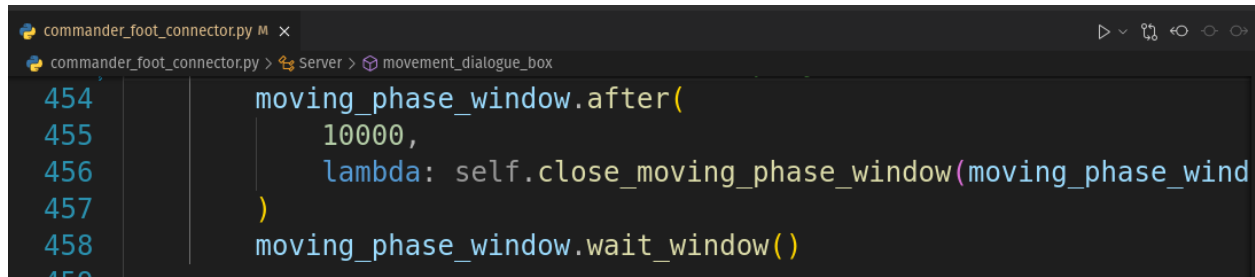
A screenshot of a code editor window showing the following code:

```
409 # Set a timeout timer for the commander to select the c
410 attacking_phase_window.after(
411     10000,
412     lambda: self.close_attacking_phase_window(attacking
413 )
```

Note, the time is to be specified in ms here.

Movement

Similarly, each player gets up to 10 seconds to select where it wants to move, failing which the AI kicks in to try and self-preserve. This time can also be modified by modifying line 455 of file `commander_foot_connector.py`.



```
commander_foot_connector.py M x
commander_foot_connector.py > Server > movement_dialogue_box
454     moving_phase_window.after(
455         10000,
456         lambda: self.close_moving_phase_window(moving_phase_wind
457     )
458     moving_phase_window.wait_window()
459
```

Extra role of commander

We decided to give the decision of the location where the missile will be launched to the commander. The commander is prompted to choose a location on a grid where they wish to launch the attack. The actual type of the missile has been randomised, else the commander would always be incentivised to choose the largest radius missile type (type 5). We felt that this makes the game more interesting.

UI Choices

Golden For Commander

The commander has an extra golden outline around it in the grid window. This is to distinguish between the commander and a regular soldier in the UI beyond the added functionalities that it gets.

Red for zone

The area of effect of the missile is depicted by red squares in the grid window, as well as by squares with coordinates in red in the movement window that pops up when the player needs to move. This was in order to provide the users with the opportunity to choose where to go in a better manner

Inputs

The soldiers are given a window to move. The commander is also given an attack window to choose where to attack.

Messages

The soldiers are given alerts when the game is over on who won (or if there is a tie). The commander is also given an alert as to what the effect of the missile launched was (hits, kills and points).

Result Convey

At the end of the game when either team/player has won, all the soldiers get a notification for the result of war (win, lose or tie). Once they close the popup using the button, the process ends.

Initialisation and Usage

Note:

- 1) *The operating systems of the machines on which the files are to be run must be Linux-based.*
- 2) *Ensure that all the required software and libraries are installed on your system.*
- 3) *Since this is a game, with multiple random initialisations and we have made use of threads, it was not possible to provide deterministic behaviour even with same inputs, our testcases are in the form of different game modes provided in shell scripts. By following the instructions documented here, you can expect to achieve the desired behaviour. We have performed rigorous testing on the same by running it multiple times as can be witnessed by the logs in our Github tree.*

Running

First, raise the privilege of the corresponding shell script files that are to be run using `chmod +x filename`. Then you will be able to execute the required files.

1) Single device version:

The single device version of the game is where the server and players (both teams) are on a single device. To run it change your present working directory to that of where this file exists and run:

```
sudo chmod +x game_single_device.sh
./game_single_device.sh
```

2) Two device version:

The two-device version of the game is where the server is on one machine and the players (both teams) are on another machine. To run it:

- Find out the IP address (ipv4) of the machine running the server using a command like `ip a`.
- Set the shell script variable `IP` in the soldier `.sh` file to be that IP address.
- On the machine that is running the server, change your present working directory to that where this file exists and run:

```
sudo chmod +x game_multi_devices_server.sh
./game_multi_devices_server.sh
```

- On the machine that is running the players, change your present working directory to that of where this file exists and run:

```
sudo chmod +x game_two_devices_soldier.sh
./game_two_devices_soldier.sh
```

3) Three device version:

The three device version of the game is where the server is on one machine and the players of each team are on their own respective machines. To run it:

- Find out the ip address (ipv4) of the machine running the server using a command like `ip a`. Set the shell script variable `IP` in the player `.sh` files to be that IP address.

- On the machine that is running the server, change your present working directory to that of where this file exists and run:

```
sudo chmod +x game_multi_devices_server.sh
./game_multi_devices_server.sh
```

- On the machine that is running player 0, change your present working directory to that of where this file exists and run:

```
sudo chmod +x game_three_devices_player0.sh
./game_three_devices_player0.sh
```

- On the machine that is running player 0, change your present working directory to that of where this file exists and run:

```
sudo chmod +x game_three_devices_player1.sh
./game_three_devices_player1.sh
```

4) $2M + 1$ device version:

The $2M + 1$ device version of the game is where the server is on one machine, and each node participating in the game is on its own device. To run it:

- Find out the ip address (ipv4) of the machine running the server using a command like `ip a`. Set the shell script variable `IP` in the player `.sh` files to be that IP address.

- On the machine that is running the server, change your present working directory to that of where this file exists and run:

```
sudo chmod +x game_multi_devices_server.sh
./game_multi_devices_server.sh
```

- On the machine that is running a player, change your present working directory to that of where this file exists and run:

```
sudo chmod +x game_2m_plus_1_devices_player.sh
./game_three_devices_player0.sh
```

When prompted, the user will need to enter their player and soldier numbers as well. Care needs to be taken that the correct player and soldier numbers are entered, else there will be overwriting and the game will not start.

Note: In the case that the user makes an error in running the scripts, they will need to explicitly kill those processes and start anew. They can use commands like `kill -9 $(pgrep python3 | awk ' $1>=INSERT-LOW-PID')`.

Components

Player Connector

Overview

The 'player_connector.py' file serves as a connector for coordinating communication between opposing teams in a game scenario. It acts as both a server and a client, facilitating interactions between the player's team and the opposing team's commander.

Code Structure

The 'player_connector.py' script is structured as follows:

1. Import Statements: The script begins with import statements to bring in the necessary libraries and gRPC-generated modules.
2. Server Class: The core functionality is encapsulated within the `Server` class, which handles server operations and RPC methods.
3. Initialization: The server is initialized with parameters such as the game field size (`N` and `M`) and the player's team number (`player`).
4. RPC Functions: Several RPC functions facilitate communication between the player's team and the opposing team's commander.
5. Main Function: The script's main function parses command-line arguments (if provided) and initializes the server.

Key Components

1) Server Class

The `Server` class defines the core functionality of the connector. It acts as a missile detection system and facilitates communication between teams. Key attributes and methods include:

- **__init__(self, N, M, player):** The constructor initializes the server with game parameters, such as field size (`N` and `M`) and the player's team number (`player`).
- **SendAlert(self, request, context):** This function is the main RPC method used to receive missile strikes from the opposing team, alert the commander, and return the consequences of the attack.
- **AttackRPCCall(self):** This function initiates an attack RPC call to the commander in response to an enemy missile launch. It handles turn-based gameplay.

- **Attack(self, request, context):** This RPC method is called by the commander to launch a missile at the opposing team. It also handles turn-based gameplay and damage calculation.
- **RelayResults(self, is_predecided, is_winner):** This function calculates and relays game results, determining the winner based on points and turns. It communicates the results to all the nodes.
- **RegisterEnemyPoints(self):** Sends points to the enemy team and receives their points in response. Used to compare scores and determine the winner.
- **GetPoints(self, request, context):** Receives points from the enemy when they are out of turns and returns the current points. Handles the end of the game and results calculation.
- **RegisterEnemyRPCCall(self):** Initiates registration to the enemy team and sets timestamps. Determines which team goes first based on timestamps.
- **CompareTimestamps(self):** Compares timestamps of both teams and initiates the game based on the turn-by-turn protocol.
- **RegisterNode(self, request, context):** Registers a node (team member) with the connector and checks if all required nodes have connected. Notifies the enemy team when all nodes are registered.
- **RegisterEnemy(self, request, context):** Registers the enemy team's timestamp and initiates the game when both teams are ready.
- **TerminateProgram(self):** Termination function to exit the program with a delay. Used to ensure all server program instances are closed.

2) RPC Functions

The following RPC functions facilitate communication between the connector and other game components:

- **SendAlert(self, request, context):** Handles incoming missile strikes, alerts the commander, and returns attack consequences.
- **Attack(self, request, context):** Allows the commander to launch a missile at the opposing team.
- **GetPoints(self, request, context):** Receives and returns points from the enemy when their turns are over.
- **RegisterNode(self, request, context):** Registers a node (team member) with the connector.
- **RegisterEnemy(self, request, context):** Registers the enemy team's timestamp and initiates the game.

Commander and foot soldier connectors

Overview

`commander_foot_connector.py` is a Python script that serves as a connector between a team's commander and its foot soldiers in a grid-based game scenario. It enables communication between the commander and the soldiers, allowing them to coordinate actions, share information, and engage in battles within a grid-based environment. The script uses gRPC for communication between different game components and provides a graphical interface for visualizing the game's progress.

Key Components

1) Server Class

The `Server` class represents each soldier object in the game. It has several attributes and methods to manage the soldier's state and interactions within the game. Key attributes and methods of the `Server` class include:

- **node_number**: The unique identifier for each soldier.
- **lives**: The number of lives a soldier has.
- **player**: The team (player) to which the soldier belongs.
- **N**: The size of the grid.
- **x** and **y**: The current position of the soldier within the grid.
- **speed**: The speed of the soldier.
- **connector_ip**: The IP address of the connector server.
- **connector_port**: The port number for communication with the connector.
- **is_commander**: A flag indicating whether the soldier is the commander.
- **ip_address** and **port_number**: The IP address and port number of the soldier.
- **battalion**: A list of soldiers and their IP addresses within the same team.
- **canvas**: A reference to the GUI canvas for rendering the soldier's position.
- **SendResult(self, request, context)**: An RPC method that receives battle results (hits, kills, points) from soldiers and displays the results in a pop-up message.
- **RegisterNodeRPCCall(self)**: Method for making an RPC call to register the soldier node with the connector server.
- **SendZone(self, request, context)**: An RPC method used by the connector to send a "red zone" to the commander for further action. It calculates hits, deaths, and points based on the received red zone and responses from battalion members.
- **UpdateStatus(self, request, context)**: An RPC method used by soldiers to respond to a red zone sent by the commander. It calculates whether the soldier is hit, records hits, and updates its status.
- **PromoteSoldier(self, request, context)**: An RPC method used by the commander to promote a soldier to the new commander. It also updates the battalion to include the new commander.
- **InitiateAttack(self, request, context)**: An RPC method used by the commander to initiate an attack on the opposing team. It opens a GUI window for the commander to select attack coordinates and type, or it selects random coordinates if the commander does not respond in time.

- **move(self, hit_x, hit_y, radius):** A method that determines the soldier's movement strategy within the grid. It calculates whether the soldier is within a red zone and moves accordingly to avoid getting hit.
- **RegisterHit(self):** A method that updates the soldier's status when it is hit by an attack. It decreases the number of lives and can result in the soldier being removed from the game.
- **open_attacking_phase(self, window):** A method that opens a GUI window for the commander to select attack coordinates during the attacking phase.
- **display_attack_status(self, window, response):** A method that displays a pop-up message with information about hits, kills, and points earned during an attack.
- **display_result(self, window, result):** A method that displays a pop-up message with the final result of the game, such as whether the team won, lost, or tied.
- **TacticalNuclearStrike(self, box):** A method that is called when a soldier dies. It destroys the pop-up message box and terminates the soldier's processes.
- **movement_dialogue_box(self, window, hit_x, hit_y, radius):** A method that is called to take the input of a soldier to decide their movement.
- **move_button_click(self, x, y, moving_phase_window, window):** A method that is called as an effector for changing the position of the soldier, and destroys the move window.
- **close_moving_phase_window(self, moving_phase_window, window):** A method that is called in case a choice is not made in time. It destroys the move window and initiates an optimal movement algorithm.

2) Server Initialization

The script initializes individual soldiers by creating instances of the `Server` class. Each soldier is associated with a unique node number, team (player), and other parameters. Upon initialization, the soldier registers itself with the connector server using an RPC call.

3) gRPC Services

The script defines several gRPC service methods that allow soldiers to communicate with each other and the connector. These include:

- **SendResult:** Receives battle results from soldiers and displays them.
- **RegisterNodeRPC:** Registers the soldier with the connector.
- **SendZone:** Receives a red zone from the connector, and the commander initiates battle.
- **UpdateStatus:** Handles a soldier's response to a red zone.
- **PromoteSoldier:** Allows the commander to promote a soldier to be the new commander.
- **InitiateAttack:** Allows the commander to initiate an attack.

4) GUI Integration

The script uses the `tkinter` library to create a GUI for displaying the game grid and soldier positions. It represents soldiers as colored rectangles on the grid.

Protocol Buffers

This is the documentation for two Protocol Buffer files: `soldier.proto` and `connector.proto`. These files define messages and services for communication between soldiers, their commander, and the connector server in a game scenario. Below are the details for each of the defined messages and services.

soldier.proto

Message Definitions:

1) SoldierData

- Contains information about a soldier, including its unique identifier, IP address, and port number.
- Fields:
 - `id` (int32, optional): Unique identifier for the soldier.
 - `ip_address` (string, optional): IP address of the soldier.
 - `port` (int32, optional): Port number associated with the soldier.

2) Position

- Stores the coordinates (x and y) of a missile strike's position.
- Fields:
 - `x` (int32, optional): X-coordinate of the position.
 - `y` (int32, optional): Y-coordinate of the position.

3) RedZone

- Contains information about a missile strike, including its position and radius.
- Fields:
 - `pos` (Position, optional): Position of the missile strike.
 - `radius` (int32, optional): Radius of the missile strike.

4) AttackStatus

- Stores information about the consequences of a missile strike on the enemy, including death count, hit count, points earned, and the current commander's ID.
- Fields:
 - `death_count` (int32, optional): Number of enemy soldiers killed.
 - `hit_count` (int32, optional): Number of enemy soldiers hit.
 - `points` (int32, optional): Points earned as a result of the missile strike.
 - `current_commander` (int32, optional): ID of the current commander.

5) SoldierStatus

- Contains information about the effect of a missile strike on a soldier, including whether it was hit, whether it sank (lost all lives), and the points earned.
- Fields:
 - `is_hit` (bool, optional): Indicates if the soldier was hit by the missile.
 - `is_sink` (bool, optional): Indicates if the soldier sank (lost all lives).

- ``points`` (int32, optional): Points earned as a result of the missile strike.

6) Battalion

- Stores information about the presently alive battalion when transferring commander duties. It includes an array of ``SoldierData`` messages.
- Fields:
 - ``soldiers`` (repeated `SoldierData`, optional): List of alive soldiers in the battalion.

7) WarResult

- Stores the result of the war, indicating whether a team won, lost, or tied.
- Fields:
 - ``result`` (int32, optional): Result code, where -1 represents a loss, 0 represents a tie, and 1 represents a win.

8) void

- A special message used when no actual data needs to be returned.

Service Definitions:

1) Alert

- Defines the services for exchanging information between the connector and the commander, as well as between the commander and the soldiers during the game. These services include sending red zones, updating soldier status, promoting soldiers to commanders, initiating attacks, and sending war results.
- RPC Methods:
 - **`SendZone(RedZone)` returns `(AttackStatus)`**: Sends a red zone (missile strike) to the commander and receives an attack status in response.
 - **`UpdateStatus(RedZone)` returns `(SoldierStatus)`**: Allows soldiers to update their status in response to a red zone.
 - **`PromoteSoldier(Battalion)` returns `(void)`**: Promotes a soldier to a new commander and transfers the current battalion.
 - **`InitiateAttack(void)` returns `(void)`**: Initiates an attack by the commander.
 - **`SendResult(WarResult)` returns `(void)`**: Sends the result of the war to the commander.

connector.proto

Message Definitions:

1) Coordinate

- Stores the coordinates (x and y) of a missile strike's position.
- Fields:
 - ``x`` (int32, optional): X-coordinate of the position.
 - ``y`` (int32, optional): Y-coordinate of the position.

2) MissileStrike

- Contains information about a missile strike, including its position (coordinates) and type.

- Fields:
 - ``pos`` (Coordinate, optional): Position of the missile strike.
 - ``type`` (int32, optional): Type of the missile strike.

3) Hit

- Stores information about the consequences of a missile strike on the enemy, including the number of hits, kills, and points earned.
- Fields:
 - ``hits`` (int32, optional): Number of enemy soldiers hit.
 - ``kills`` (int32, optional): Number of enemy soldiers killed.
 - ``points`` (int32, optional): Points earned as a result of the missile strike.

4) Points

- Used to exchange the points each team has between each other.
- Fields:
 - ``points`` (int32, optional): The points to be exchanged.

5) Timestamp

- Stores a timestamp used to decide which team attacks first during the game.
- Fields:
 - ``timestamp`` (float, optional): A timestamp value.

Service Definitions:

1) PassAlert

- Defines the services for exchanging information between teams, such as sending missile alerts, registering nodes, registering enemies, initiating attacks, and exchanging points and timestamps.
- RPC Methods:
 - **SendAlert(MissileStrike) returns (Hit):** Sends a missile alert (missile strike) to the enemy and receives information about the consequences.
 - **RegisterNode(soldier.SoldierData) returns (soldier.void):** Registers a soldier node with the connector server.
 - **RegisterEnemy(Timestamp) returns (soldier.void):** Registers the enemy team with a timestamp.
 - **Attack(MissileStrike) returns (Hit):** Initiates an attack on the enemy team by sending a missile strike and receiving information about the consequences.
 - **GetPoints(Points) returns (Points):** Exchanges points between teams.