**Dragon Arena Game Implementation requirements**

**Mandatory**

* Improve the game’s AI for players base on “simple strategy: heal a nearby player as soon as there is one that has below 50% of its initial hp, and go towards the closest dragon and strike otherwise”. Right now in the provided code, players randomly pick a direction and perform an action based on what lies in that direction(Adjacent cell). We should make it so that a ‘player’ scans adjacent cells for healing targets first. Additionally the attack range (=2) and the heal range (=5) are not coded.

Brief overview of what was discussed in the previous meeting: The AI should be re-written (according to the requirements) based on the following 3 step decision making process.

1) Check nearby area(5) for valid healing targets. If a target is found, heal them.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | range |  |  |  |  |  |
|  |  |  |  | range |  | range |  |  |  |  |
|  |  |  | range |  |  |  | range |  |  |  |
|  |  | range |  |  |  |  |  | range |  |  |
|  | range |  |  |  |  |  |  |  | range |  |
| range |  |  |  |  | Player |  |  |  |  | range |
|  | range |  |  |  |  |  |  |  | range |  |
|  |  | range |  |  |  |  |  | range |  |  |
|  |  |  | range |  |  |  | range |  |  |  |
|  |  |  |  | range |  | range |  |  |  |  |
|  |  |  |  |  | range |  |  |  |  |  |

2) If no valid healing target exists, check the immediate area (2) for valid attack targets (Dragons). If a target is found, attack it.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | range |  |  |
|  | range |  | range |  |
| range |  | Unit (both player and dragon) |  | range |
|  | range |  | range |  |
|  |  | range |  |  |

3) If no attack target is found, move in a random direction.

(Be mindful of OutOfBounds exceptions when coding this).

* Implement TOM queue for consistency between the game servers. Based on what discussed, when a client message arrives at any server, the following actions will be performed by the server.
  + Read the received message and get the client timestamp.
  + Use the timestamp to insert message into the TOM queue.
  + Broadcast the message to the rest of the servers so they add it to their local queues.

* Implement publish-subscribe model for pushing game updates to clients. When discussing the testing setup and looking at the code we came to the conclusion that the clients (“players”) are part of the server code. There is no client software provided(only 1 main method for the server code ). Essentially clients are threads generated at runtime, on the game server and they communicate with the ‘server’ thread.
* “For connecting to and disconnecting from the system, WantGame BV wants to use data from a real workload trace taken from the Game Trace Archive [5]” Available at: <http://gta.st.ewi.tudelft.nl/datasets/> .Initially a number of players are generated(above the min player cap) and the game begins. The server then adds a player every X seconds (5 sec \* Game Speed)until the player cap is reached. Based on the requirements we also have to implement trace driven simulation. Proposed solution: command line argument( if true, read trace, if false use default (provided) spawnrate. If trace is enabled add players to the game relative to the timestamps specific by the trace file. We still need to find a suitable trace that include connect/disconnect events with timestamps.
* Logging: “all game and system events (e.g., player moves, strikes, client and server node restarts) must be logged in the order in which they occur, on at least two server nodes.”

Every server is aware of **all** the game events due to using a common queue(TOM). Every server is also aware of if any clients/servers have crashed since those event are also deduced from failed communication(server does not send ack or server doesnt publish game state to clients while player disconnect event are also put in the queue).

Essentially we need a logger class, that will write all game and system events to a file. Initially get it to work on 2 servers(hardcoded). At a later stage create a logging group where these servers belong to and handle cases such as 1 of the logging server crashes and needs to be replaced(removed from group and another is added). Identifying which node is which, might be problematic since the DAS4 changes the nodes on each job submission.

* Scalability: Remove/change hard coded player/dragon caps. Although the test cases include the baseline benchmark, we can also experiment with different caps so as to report on scalability. We could have these caps/variables stored in a configuration file so we don't need to recompile for different test cases.

**Optional**

…..

**Bugs in the provided code**

Because the TA’s slipped some bugs in the provided code on purpose, and because we should all be on the same page for testing, list all observed bugs below:

* Player.java lines 29 and 30 should be switched around
* Player.java line 82 values() method undefined
* BattleField.java line 14 (Even if we should create the missing classes/packages, the import distributed.systems.example.LocalSocket seems to not belong there.

**Test cases**

While we implement the features, we should also figure out on what machine(s) we will test our code(DAS4 vs other options).

* Run a baseline benchmark (100 players, 20 dragons, and 5 server nodes,).
* Run a both a lighter and a heavier load (more/less players, dragons, nodes). This could be useful for reporting the scalability of our system.
* Run a simulation where 1 or more players disconnect from the game. (code a dc function)
* Run a simulation where 1 server crashes at random. (unaware of how to do this at runtime)
* Run a simulation where 1 server who is also a main logger, crashes at random.