**Dragon Arena Game Design Document**

This document covers the project requirements, our analysis of them, as well as proposed implementation solutions. Additionally it also contains an evaluation plan (test cases) for the project and specific features. Our approach on creating a distributed version of the DAS game is focused on Total-ordered multicasting [insert textbook ref]. Specifically utilizing Skeen’s algorithm for server-to-server communication based on Lamport’s logical clock[insert paper ref]. To address the client receiving updates on the game state, we opted for a publish(s)-subscribe(c) model.

**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 have to modify the existing AI to make it so that a ‘player’ scans adjacent cells for healing targets first and attack targets or moves otherwise. Additionally the attack range (=2) and the heal range (=5) are not coded.

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. (As an optimization we can check for nearby Dragons as well during this step).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | 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 towards the direction of the closest dragon.

Recursively check nearby cells (expanding outwards) to locate a Dragon and ‘pathfind’ to it.

* Implementing TOM queues 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.
  + Accept the received client message and append the local LC time-stamp.
  + Multicast the message to the rest of the servers and move it to the local un-deliverables queue.
  + Every other server stores the message to its local un-deliverables queue. If needed they adjust their local clock.
  + The servers then send back their proposed timestamps for that specific message.
  + The initiator of the process gathers the proposed timestamps, and selects the maximum between them as the final time-stamp for the message.
  + It proceeds to multicast the elected time stamps to the rest of the servers and moves the specific message from the local un-deliverables to the local execution queue.
  + The rest of the servers do the same for their copy of the message as soon as they receive the final time-stamp.

These actions constitute what we have dubbed as the TOM procedure. This procedure is decoupled from its input and output stages, meaning that a call to the procedure should not be blocking for the server. The decoupling is achieved by using queues both for the input and the output. When a client message is received by the server, a ‘client-receiver’ thread should take the message, insert it to the input queue, and keep on listening for other incoming messages.

Considering the output of the TOM procedure, there should be a separate thread continuously checking on the TOM output queue for available executable actions to be applied to the battlefield. Since message ordering is guaranteed by the TOM process, the output queue will have the same messages on each server, ensuring consistency among the replicated game states.

Essentially the TOM procedure should exist as a software layer between a server’s receiving socket and the battlefield. This implementation induces overhead for each message that needs to be processed by the servers but in return it ensures that messages are totally ordered within the system. This design choice is based on what players (in this case the final customers) would expect from an on-line multi-player game. Given that rollbacks and inconsistency is generally not nice to experience as a player and due to the fact that the DAS game is rather simple (no 3D representation or complicate game actions), we decided that consistency should be a focus of our project.

* 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 the players to receive updates on the game, we decided that the map and the units it contains should be published on a set rate to the clients (every x ticks). This means that when a player issues a move (besides the get method), the servers should respond with an acknowledgement that the move was received and a ‘promise’ that the move will be executed sometime in the near future.
* “*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) spawn rate. If trace is enabled add players to the game relative to the timestamps specific by the trace file.
* Fault tolerance and 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 events are also deduced from failed communication(server does not send ack or server doesnt publish game state to clients, or no response is received within the specified timeout).

The expected behaviour for crashes is as follows:

When 1 or more players crash they are simply removed from the battlefield with no option to reconnect to a state they left, meaning that they will have to re-join as a new player.

When 1 or more game servers crash, all the players connected to it are also disconnected from the game. It is up to the players to re-establish a connection and join the game as a new entity.

Essentially we also 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).

* 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**

Additional features

Our plans for extending the project with additional features revolve around Advanced fault-tolerance and Benchmarking. Ideally we would like to improve fault tolerance by implementing current state saving and auto-reconnect options. For instance if a player crashes during the game, his in-game entity would enter a ‘stasis’ phase (no actions performed, just sitting idly) for a period of time, giving the player the chance to reconnect and pick up where he left off before disconnecting.

We would also implement a similar feature for server crashes. When a server would crash, all the players connected to it will also enter a ‘stasis’ phase instead of disconnecting entirely, allowing time for the client software to search for other game servers to connect to, so that the players can continue playing.

**Test cases**

While we implement the features, we should also figure out how to test them and refine them based on the results.

**Basic testing**

* 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. (dc the server by killing process)
* Run a simulation where 1 server who is also a main logger, crashes at random.

**Consistency Testing (TOM)**

To effectively test our implementation of Total-ordered Multicasting, we will have to do it on a gradual basis, starting from the point of origin and incrementally work our way up to testing the complete solution. We can achieve this by hard-coding test scenarios in our solution.

For instance we will begin with having 5 servers running and only 1 player connect to the game. This will allow us to test whether if the communication between the servers is working as expected.

For the next step we will allow 2 players to connect (on different servers) and have them issue game commands at fixed interval instead of random. This way we can test if our implementation orders the messages correctly.

From that point onwards, we can incrementally add more players to each execution test, so as to further ensure that the ordering works and the system is consistent.