Distributed game - Project documentation

Tadej Vozlič
UP FAMNIT, RIN
Koper, Slovenia

Tilen Jesenko
UP FAMNIT, RIN
Koper, Slovenia

Domen Vake
UP FAMNIT, RIN
Koper, Slovenia

1 INTRODUCTION

The document includes the documentation for the project, developed for the RIN course "Project seminar". The project that was implemented is a distributed version of the Atomic Bomberman game, that was introduced to the market in 1997. The main difference in the implementation concepts of the original game and the game presented in this document, is the client-server architecture.

The project does not require a central server to function. The game is completely distributed and every player presents a node in a distributed network. Every node is both a server and the client. Players that play the game are all equal and communicate through a distributed message passing algorithm to exchange information.

The game includes a move verify agent that checks for validity of every move of every player to see weather any of the players are cheating.

2 ARCHITECTURE

The project is structured from three main components. The game, the communication layer(web socket API) and the validator as seen on the figure 1. All the events a client initiates start in the game component and get validated by the validator component and then by using the web socket API, get distributed to other nodes.

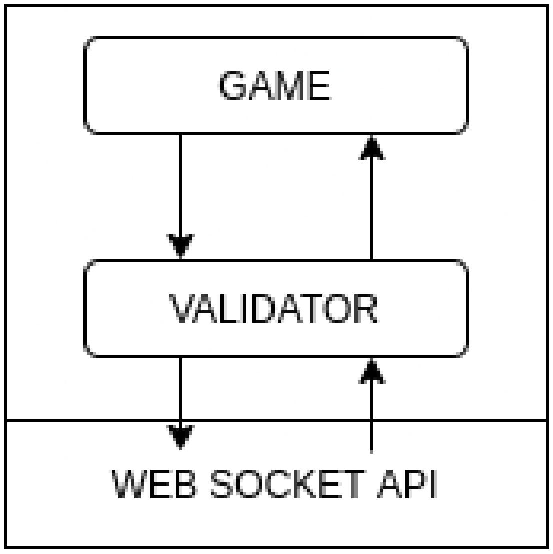

Figure 1: Client architecture

As previously stated every player is represented by it's own node and to play a multiplayer game, players need to communicate. Every node is connected to each other at the communication layer. Every node has a web socket sever connection established with all the other nodes, to exchange messages as seen on the image 3.

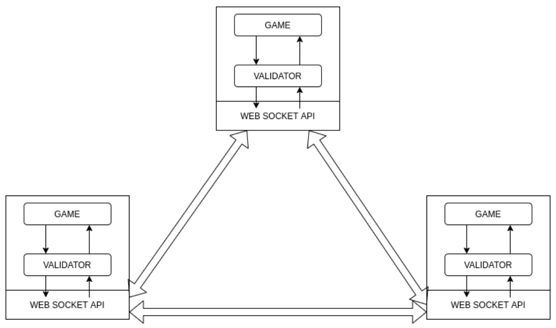

Figure 2: Distributed network architecture

3 GAME

Original game

The game Bomberman was firstly introduced in 1983 on Nintendo entertainment system. It is an arcade maze based game, where player takes control of atomic bomberman - a robot, who has to navigate his way trough the maze while avoiding enemies. He can deploy bombs to destroy enemies or breakable blocks on his way. Bomb can however be deployed on current bomberman's position. When bomb explodes it creates fire in shape of cross like it is shown in image below. Fire is as big as it is player's current fire level. At the beginning it only explodes 1 tile up, down, right and left but can become bigger later on. If any breakable block gets in a way of a fire, it gets destroyed. These blocks can drop power ups on destruction for example: speed boost, more bombs or bigger fire. The goal is to destroy all enemies and reach portal to next room, which is hidden behind random block. He must do so before time runs out, otherwise he loses. Game had total of 50 levels, each with increased difficulty.

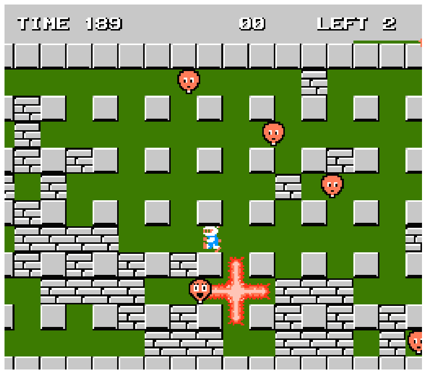

Figure 3: Screenshot from original Bomberman game NES

Peer to peer implementation of bomberman

The game has many remakes since 1983 with newest edition 2021. We implemented our version of bomberman in Java using game library called LibGDX. We used open open source textures to draw animation of objects like bomberman, bomb, fire and blocks. For map creation we used open source tool called Tiled. That tool helps developers draw maps using tiles and specify every tile's properties. Using these properties we specified what is background, what are indestructable blocks and which are breakable. Breakable ones can then also drop power ups as in original game. Unlike original game, which had multiple biomes we only used one. It can be played in multiplayer mode only for up to 4 players and no enemies are spawn unlike in original game. Rules in multiplayer are somewhat different than in original single player game. In multiplayer game, players are in either in teams or each player for himself. Every player spawns in each corner and break walls on the way to each other. They can also pick up power ups to make themselves stronger. The goal is to kill every other player or team and be the last player or team standing.

In our game players communicate with each other using peer to peer technology where every player is equal. Since there is no server involved, every peer - player has to watch other players if anyone is cheating or doing invalid moves like placing bombs on location on the other side of map where enemy is. Players join the game via menu where user chooses if he wants to host the game or join it. If he chooses to Host the game, new lobby is created which is seen by other players who wish to play the game.

4 COMMUNICATION LAYER

The communication layer is responsible for transmitting the data from client to client. The meaning of the data has little importance to this layer, since it's evaluated in the next (validator) step. This layer is in charge of monitoring the existence and changes of a game room we are in. The layer stores a RoomServer object, that stores all the players in the game room. All players communicate on the principle of protocols. A protocol is *a system of rules that explain the correct conduct and procedures to be followed.* In this layer we follow three main and two additional support protocols. Main protocols:

- Handshake protocol
- Peer discovery protocol
- Broadcast protocol

Support protocols:

- Host game protocol
- Fetch rooms protocol

Each of the protocols accomplishes a specific task in the system. The main three protocols enable us to play the game. They connect the player's clients and transmit messages among them. On the other hand we have support protocols, that are not mandatory for the game to be running, but are there to ease the process of getting a specific client into a game room, to play the game.

Protocol structure

Each protocol works on a similar principle. All the protocols have the two same methods. The methods being *initiate* and *digest*. The initiate method is the first method to be called when trying to initiate the protocol. On the other hand, when a client receives the message, the message stores the name of the protocol it is a part of. The protocol gets called by the digest method, taking the message as the input argument.

All the protocols work in three steps. In the initiate stage the client sends a REQUEST typed message to the recipient. Upon receiving the message, the recipient responds with a RESPONSE message and when the initiator receives this message it sends a ACKNOWLEDGE message, that denotes the end of the protocol.

Handshake protocol

This protocol is in charge on setting up a socket connection between the initiator and the receiver of the protocol communications. In the REQUEST part of the message it sends the receiver the relevant information about itself. That is the name of the player, public IP of the client, private IP of the client and the port on which the socket server is running on the client. Upon receiving the REQUEST the recipient sends the same type of information back to the initiator in the RESPONSE message. When the initiator receives the RESPONSE message it sends back the ACKNOWLEDGE message and the recipient establishes a socket connection to the initiator based on the exchanged information.

Peer discovery protocol

After establishing connection with a client in the game room, the initiator client initiates the Peer discovery protocol. The main task of the protocol is to exchange the list of the clients in the room, so all the clients establish a connection with one another. The initiator packs the data of all the clients it knows in the room and sends it to the recipient. The recipient than responds with its own list of clients. Both initiator and the recipient then check if they already have a connection established with the received clients and if they find an unknown client they initiate a Handshake request with the client.

Broadcast protocol

Broadcast protocol is in charge of distributing the data through the network. Once an event in the game has been generated the initiator sends the event to every known client in the network. Every client than receives the message and handles the event in the game, once it was validated by the validator. But also upon getting the message it also sends it to all clients it knows. All clients do the same as long as they did not already forward the message once. This ensures that there are multiple ways of distributing the message to any client, in case a connection between any two clients would drop.

Host game protocol

Host game protocol is not an essential protocol for running the game. This protocol is used to inform the lobby server, that we as a client are open to play the game and have opened a game web socket server, for other people to connect to. The initiator is always a client and the recipient is always the lobby server. The initiator in the REQUEST message sends the credentials (public IP, private IP, server port and game room id) to the lobby server. The lobby server then stores this data so it is available to other clients. Upon receiving it sends back the ACKNOWLEDGE message.

Fetch rooms protocol

The fetch rooms protocol is the last protocol in out message layer. It is also optional and not essential for playing the game. The protocol is used for the client to acquire the open rooms data from the lobby server. The initiator is always a client and the recipient is always the lobby server. The initiator sends a REQUEST message with an empty body to the lobby server. The server then sends a RESPONSE message to the client with all the server information of all the stored rooms. The client may use this data to initiate the handshake protocol with any of the hosts to join their room.

5 VALIDATOR

Validator is layer between socket and game's graphical interface. It serves all players participating in game as anti cheat program, which validates their moves so only valid ones are acknowledged. In case one player makes invalid moves intentionally or unintentionally, validation on other player's end rejects this move and it is ignored. Since our game is based on peer to peer technology instead of server based, validation must be done on all players end.

Validator receives a message from communication layer and first checks if the event structure is valid. Each event has action type, unique identifier, data and local game tick. Content of data depends on the action type. If event is structured correctly and has necessary data associated with it, then it is pass to the game layer.

Actin type is used to define the action that happened on local game and needs to be updated on all the other remote clients. Unique identifier or initiator or id, as it is named in the game, tells the other nodes from with node the message was send. For example, there are 4 players in the game, and validator receives a message that the player moved to the left. Validator needs to know from which node that message came, so the game layer can move correct player to the left. Data is a structure that holds any $key - value$ pair of data. Mostly holds the location there the event happened. In case on placing bomb data holds the x and y location. The same goes for break a wall. This data structure could also hold any data that might be added in the future. Local game tick tells, when the event happened. Although game ticks in nodes are not synchronized, the tick can still be used to check is player is moving at the correct speed or if a player can place another bomb.

Action types are declared in *Action* class and below list represents the valid once:

- NEW_PLAYER
- PLAYER_MOVED_RIGHT
- PLAYER_MOVED_LEFT
- PLAYER_MOVED_UP
- PLAYER_MOVED_DOWN
- PLAYER_NOT_MOVING
- PLAYER_PLACED_BOMB
- BOMB_EXPLOSION
- PLAYER_UPGRADE_PICKUP
- WALL_DESTROYED
- UPGRADE_SPAWN
- PLAYER_DEAD
- NOTHING

Any other action will be rejected by validator and therefor not processed and pass further to the game layer.

Action types, their data and effects

When *NEW_PLAYER* is passed from the communication layer to validator, a new player has joined the room. Validator evaluates the event and adds the event to event queue for game layer to handle it.

Actions *PLAYER_MOVED_RIGHT*, *PLAYER_MOVED_LEFT*, *PLAYER_MOVED_UP*, *PLAYER_MOVED_DOWN*, *PLAYER_NOT_MOVING*, as their name implies, are player movement actions.If a player moves right, the *PLAYER_MOVED_RIGHT* actions gets propagated to the other nodes, if a player moves up, the *PLAYER_MOVED_UP* action gets propagated to the nodes in the room and so on. Initial idea was to only propagate the event for player's movement direction and *PLAYER_NOT_MOVING*, when player stops. The actual movement would be handled by the game's player movement function. But it was quickly discovered, that this way players gets out of synchronization, meaning their location throughout the nodes is not the same. Therefor player's local x and y location is added to the event's data.

Similar to player's movement actions, is action called *PLAYER_PLACED_BOMB* and *BOMB_EXPLODED*. Each time x and y location is added to event's data.

PLAYER_DEAD action is propagated when a flame touches a player. When this event is received from the communication layer, validator checks for the player id and adds the event to event queue. When player dies, the node is also disconnected from the game.

Other actions are not implemented in the current game. All of them, requires x and y location, to identify which object is the target.

Security

The idea around security in peer-to-peer networks is guarantee that data cannot be fakes (but this is almost impossible) or to check each and every message to be authentic and valid. In a peer-to-peer network game, such as our Bomberman, events needs to be checked before they are presented in the game. Validators task is to make sure game state is the same on all nodes.

Player movement

One of most noticeable "cheats" are faking player movement. For example, one could forge the location of a player to avoid being killed by the bomb blast or to pick an upgrade. To prevent this from happening, event objects contains local game tick. Since the players speed is known, validator would have to check players old location and game tick at that time, and compare it to players new location and game tick. If validator determines the player could not move to new location in amount of time passed since last location event, the players position should be restored to the old location.

Placing bombs

Game rules are, player has only 1 bomb at the beginning of the game and can pick upgrades to increase the available number of bombs. When validator receives a *PLAYER_PLACED_BOMB* event, would have to check is this player has another bomb available before passing event to event queue. If a player has placed all his bombs already, validator should reject this event and not pass it to the game.

Picking up upgrades

The a wall in broken, there is a change an upgrade can spawn. When receiving this event, validator checks if this upgrade is still available and if it actually exists. If both are true, the event is confirmed and pass to the event queue, otherwise message gets rejected.