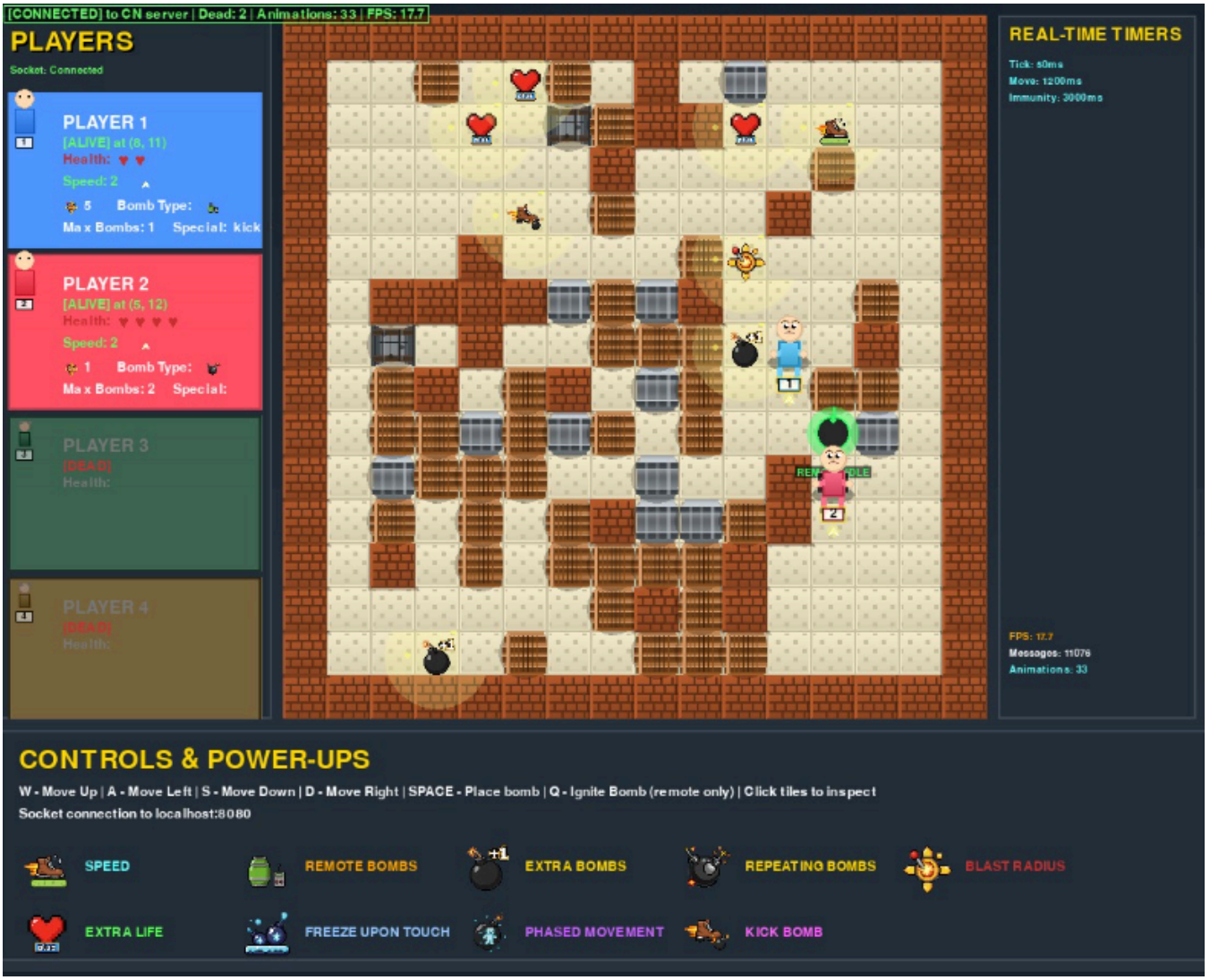
Playing with Fire



Dolev Eisenberg
Roi Alus

Relevant Links:

Git

Youtube

Project Description:

Playing with Fire Erlang is a multiplayer maze-based combat game designed for 4 players across separate computers. Each participant can choose to either play manually or deploy an AI bot to compete on their behalf. The game is fully distributed, running across 5 nodes - 4 of which are player nodes (dubbed “GNs”) while the last is the central node (dubbed “CN”).

The project utilizes Mnesia for distributed database management, QLC for querying from the database in an efficient manner, GUI using Python via port communication and PyGame for generating the visuals.

Core Gameplay

4 players navigate through a 16x16 maze, dynamically generated using a Steiner tree-based algorithm. The map is composed of both destructible and indestructible tiles, which upon destruction may drop a powerup the players can pick up to their benefit (these powerups include: faster movespeed, different bomb types, ability to walk through bombs, increased bomb explosion radius, increased maximal bomb threshold and more). Each player is able to move around the map and place bombs, which detonate (either by a set delay or user input, depending on the bomb’s type) and damage nearby elements. The players’ goal is to kill the other players while simultaneously avoiding blast damage themselves.

Winning condition: Last player standing - survive longer than all opponents through strategic movement, tactical bomb placement, and effective power-up utilization.

Movement System

Each player, whether human-controlled or bot, is able to send requests to move on the map via the WASD keys (up,down,left & right), while also being able to place bombs (space or e key) and ignite remote bombs via the q key (when this is the

equipped bomb type). When bot-controlled, the bot is able to request the same commands via a standalone bot request process.

Power-Up System

The game features power-ups, which can be dropped by tiles upon breaking. These power-ups can be picked-up by the players by entering their coordinate on the map, and provide one of the following:

- Speed Boost: Increases player movement speed, allowing them to move faster through the map.
- Phased Movement: Allows the player to walk through placed bombs, bombs no longer block the player from moving.
- Plus Bomb: Increases the maximum number of bombs that can be simultaneously placed by the player by 1.
- Remote Bomb: Changes the bomb type to a remote-controlled bomb that explodes when a player presses the *ignite* key (Q). These bombs do not explode automatically on-timer like the default bombs.
- Repeating Bomb: Changes the bomb type to repeating, meaning once the player's bomb explodes, a 2nd bomb spawns in its wake.
- Extra life: Grants the player an extra life point.
- Freeze: Upon touching a bomb (i.e. moving into one), freezes the bomb, causing it to explode on an extended delay.
- Explosion Radius: Increases the radius of explosions generated by the player's bomb by 1 (at each direction).

Power-ups spawn with predetermined probability rates when players destroy breakables tiles.

Health & Survival System

Each player begins with 3 life points. When damaged by a bomb's explosion, players lose one life point and gain temporary immunity from further damage for several seconds. Once a player's life points reach zero, they are eliminated from the match. Players are able to replenish life points by picking up an *extra life* powerup.

Fault Tolerance

The CN monitors all GN PIDs. When a process exits:

- If the PID is not a GN, CN ignores it.
- If the PID belongs to one of the four GNs, CN identifies which GN (GN1–GN4) has failed and begins the recovery process.

To determine where to restart the crashed GN, CN uses a load-balance strategy:

- Counts the number of active GN processes on each node.
- Selects the node with the smallest number of GNs currently running.

This ensures that workload is balanced dynamically across available nodes.

Once the target host node is selected, CN executes a recovery procedure:

1. Replicates State - CN copies the crashed GN's Mnesia tables (tiles, players, power-ups, bombs) to the new node.
2. Restarts GN - A new GN process is spawned remotely via rcp:call.
3. Reconstruction - The restarted GN reinitializes its tiles, players, bombs, and bot processes based on the recovered Mnesia tables.
4. Updates CN State - CN updates its internal gn data list, replacing the old PID with the new GN PID.

Project Architecture:

The game runs across a network of 5 separate computers in a distributed architecture. 4 of these computers serve as end-point computers from which a player can play, and the 5th computer being a control node, serving as a central communication hub, responsible for fault-tolerance and synchronizing the game to all participants. As such, the map is split into 4 distinct quarters - each quarter is being controlled by one of the four Game Nodes (end-point computers), controlling and overseeing all interactions and objects within their region of the map.

During gameplay, each of the nodes is composed of two main processes - the CN/GN Server, and a Graphics Server.

Central Node (CN) Server

The CN Server serves multiple roles in the system. Namely, as the primary orchestrator, data management hub and fault-tolerance controller:

1. Database Management
    - Constructs and distributes all Mnesia database tables across the system - all tables are shared to the CN and an updated copy of the data is stored in it.
    - Utilizes QLC for complex, cross-node & table data queries and real-time game state analysis.
    - Retains an updated, accurate game state that is used to generate uniform GUI to all participating nodes.
2. Network Synchronization and Fault Tolerance
    - Discovers and establishes connections with all Game Node servers during the start-up phase.
    - Implements fault tolerance mechanisms for server crashes and disconnections, giving over the responsibilities of the failed nodes to another active node until the original node regains connectivity.
    - Sends a signal to all GNs when everything is loaded up and ready for the game to start, assuring synchronization and preventing unwanted behavior before the game starts.

3. Map Generation
    - Generates the complete game map structure using a Steiner tree-based algorithm, making sure all players are able to reach one another, and creating a unique map for every match.
    - Loads the map to the Mnesia tables and splits the map into quarters, to be handled by each GN server.

CN Graphics Server

An Erlang process responsible for graphics coordination and generation.

Responsibilities:

- Retrieves real-time game data from Mnesia tables using table subscriptions and queries for efficient data filtering and aggregation.
- Processes game events, player positions, and state changes using *mnesia:select/2* pattern matching.
- Forwards processed data for visualization to the Python GUI using port communication.
- Distributes graphical updates to all GN Graphics Servers, making sure the graphics seen by all players is uniform, consistent and up to date.

Game Node (GN) Server

Each GN Server manages a dedicated game quarter with all its sub-processes, as well as a single player- bot or human-controlled.

Responsibilities:

- Controls one quarter of the game map (clockwise order, with top-left quarter controlled by GN1).
- Manages one player (human player or AI bot) - consisting of several processes *(player_fsm, bot_handler/ io_handler & user input interface)*.
- Processes player action requests and validates moves.
- Handles all map changes and interaction within his quarter - tile breaking, bomb exploding, power-up dropping and pick-up etc.
- Communicates bidirectionally with CN server (for forwarding requests to other GNs & cross-node query request).
- Uses queries to access local game state and player-specific data from Mnesia tables.

Communication Flow:

- Receives player input requests.
- Exchanges data and confirmations with CN server.
- Returns responses to both players and CN server.

GN Graphics Server

Localized graphics management component that:

- Receives graphics data from CN Graphics Server.
- Processes visualization updates using game state information.
- Sends updates about the map state to the Python GUI as JSON data.
- Maintains player-specific view synchronization.

Player FSM:

Main Flow Transitions

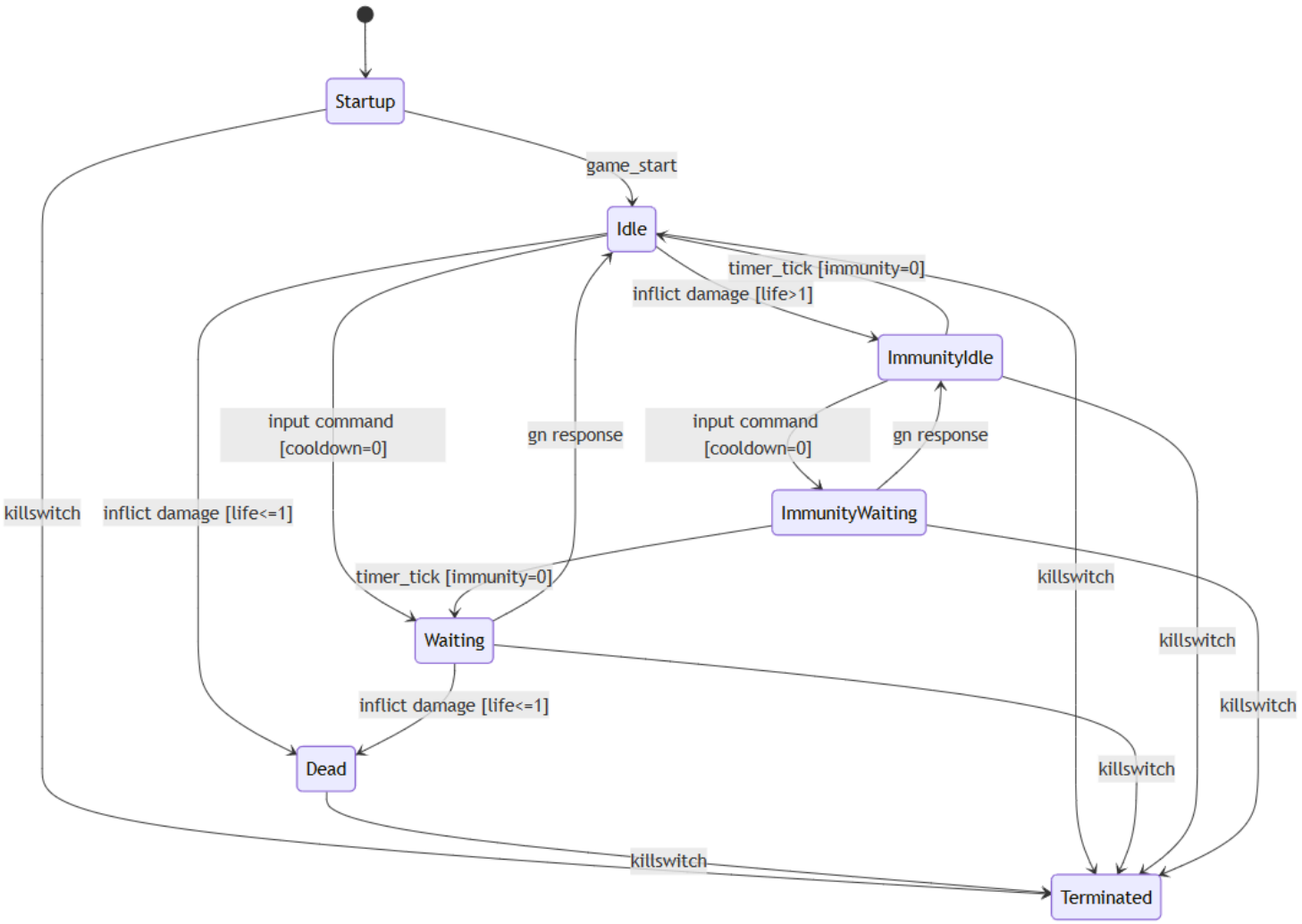
Current State	Logical Event	Condition	Action	Next State
startup	game start	-	Notify I/O handler, start timers	idle
idle	input command	cooldown = 0	Process command, send to GN	waiting for response
waiting for response	gn response	-	Handle response, update state	idle
idle	inflict damage	life > 1	Decrement life, set immunity	immunity idle
immunity idle	input command	cooldown = 0	Process command, send to GN	immunity waiting for response
immunity waiting for response	gn response	-	Handle response, update state	immunity idle
immunity idle	timer_tick	immunity = 0	Clear immunity timer	idle
immunity waiting for response	timer_tick	immunity = 0	Clear immunity timer	waiting for response
any state	inflict damage	life ≤ 1	Notify death, stop I/O	dead
any state	killswitch	-	Clean termination	terminated

Enumeration of Events

Event	Description	Interrupt Type
game_start	Signal to start the game. Transitions player from startup to active state and notifies I/O handler that game has begun.	Cast
timer_tick	Regular timer event for updating cooldowns (immunity, request, movement). Manages state transitions based on timer expirations.	Info
input_command	Player input command from I/O handler (move, drop_bomb, ignite_remote). Processed based on current state and cooldowns.	Cast
gn_response	Response from Game Node to a previous player request. Contains result of move, bomb placement, or ignition attempts.	Cast
inflict_damage	Player takes damage from explosion. Reduces life and may trigger immunity period or death.	Cast
bomb_exploded	Updates the target Game Node due to coordinate changes or network topology changes.	Cast
update_target_gn	Updates the target Game Node due to coordinate changes or network topology changes.	Cast
notify_power_up	Notification of power-up collection. Updates player stats (speed, bomb capacity, life).	Cast
killswitch	Emergency termination signal. Stops the player process and associated I/O handler.	Cast

State-Level Decomposition

State	Description
startup	Initial state. Awaiting game start signal before allowing any player inputs. The player FSM is initialized but not yet active in the game.
idle	Normal active state. Ready to receive and process input commands from the I/O handler. Player can move, drop bombs, and perform actions. Not waiting for any GN response.
waiting_for_response	Player has sent a request to the Game Node (GN) and is waiting for a response. No new input commands are processed until response is received.
immunity_idle	Player is in immunity period after taking damage. Cannot take additional damage but can still receive and process input commands. Ready for actions.
immunity_waiting_for_response	Player is in immunity period and also waiting for a GN response. Cannot take damage and cannot process new input commands until response is received.
dead	Player has died (life = 0). Cannot perform any actions except responding to killswitch. Process remains alive until explicitly terminated.


Startup
game_start
Idle
timer_tick [immunity=0]
inflict damage [life>1]
ImmunityIdle
input command
[cooldown=0]
gn response
input command
[cooldown=0]
gn response
ImmunityWaiting
killswitch
inflict damage [life<=1]
timer_tick [immunity=0]
killswitch
Waiting
killswitch
inflict damage [life<=1]
killswitch
Dead
killswitch
killswitch
Terminated

I/O-Handler:

Main Flow Transitions

Current State	Logical Event	Condition	Action	Next State
uninitialized	set player pid	-	Store player PID	ready
ready	external input or keyboard input	valid input	Convert to command, send to player	waiting for ack
ready	external input or keyboard input	invalid input	Ignore input	ready
waiting for ack	external input or keyboard input	-	Buffer input (or drop)	waiting for ack
waiting for ack	player ack	buffer empty	Display response to user	ready
waiting for ack	player ack	buffer not empty	Display response, process next buffered input	waiting for ack

Enumeration of Events

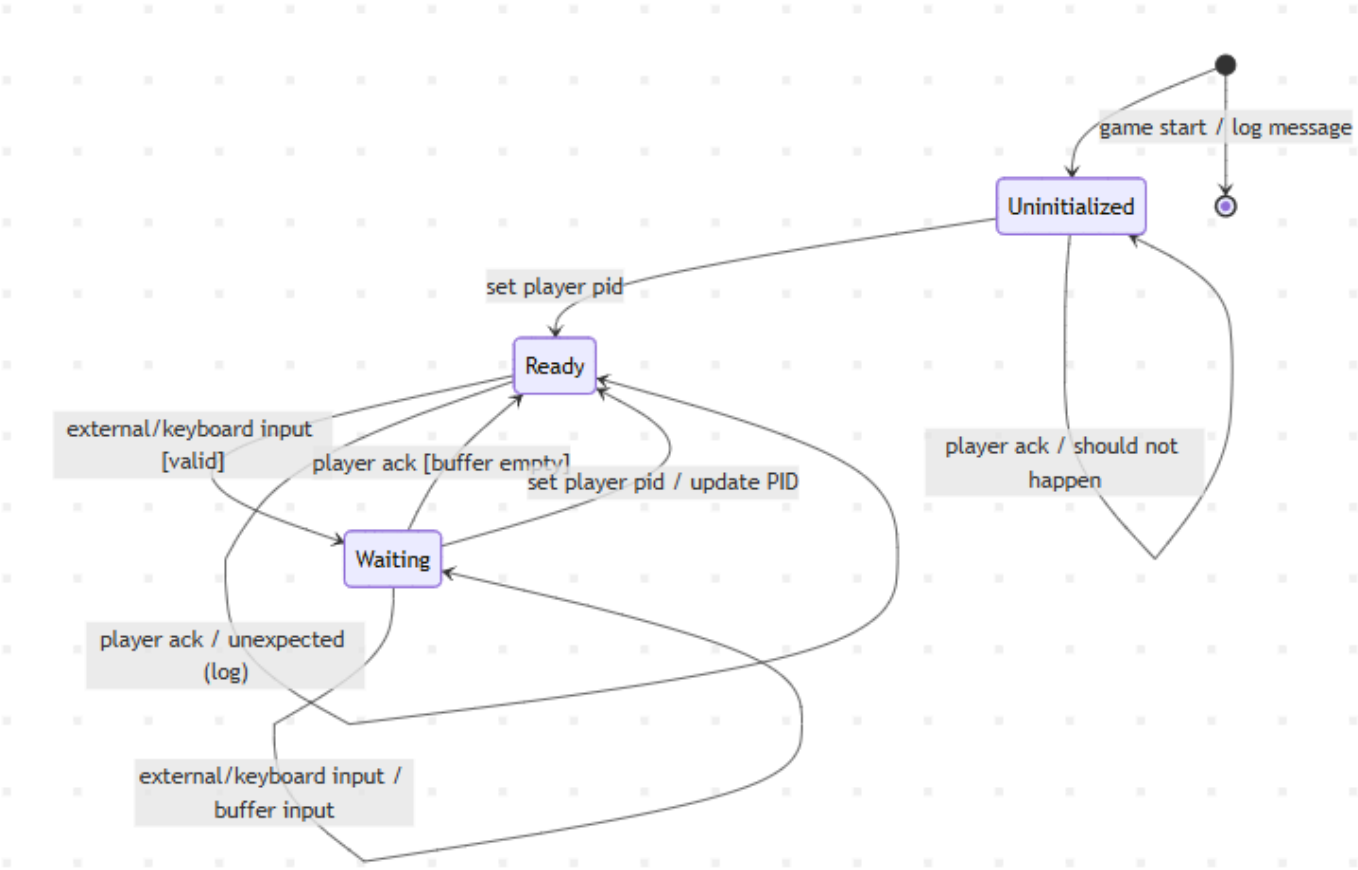
Event	Description	Interrupt Type
set_player_pid	Sets the player FSM PID after the player process has been created. Required before the handler can communicate with the player.	Call
game_start	Notification that the game has started. Enables the handler to begin processing input and communicating with the player FSM.	Cast
external_input	Input command received from external source (not keyboard). Processed same as keyboard input but comes via cast message.	Cast
player_ack	Acknowledgment from player FSM that a command has been processed. Allows handler to process buffered inputs or accept new commands.	Cast
keyboard_input	Raw keyboard input from the Python port process. Converted to game commands (move, bomb, ignite) before forwarding to player FSM.	Info

State-Level Decomposition

State	Description
initialization	Initial state after startup. The handler has been created but is waiting for the player FSM PID to be set before it can communicate with the player process.
waiting_for_game_start	Player PID has been set, but the game has not yet started. Handler is ready but not actively polling for input or processing commands.
ready_for_input	Normal operational state. Game has started, handler is ready to accept keyboard input and forward commands to the player FSM. Not waiting for any acknowledgment.
waiting_for_ack	Handler has forwarded a command to the player FSM and is waiting for acknowledgment. Additional input is buffered during this time to prevent command flooding.
input_buffering	When waiting for acknowledgment, new inputs are buffered rather than immediately forwarded. This prevents overwhelming the player FSM with rapid keystrokes.

Input Command Mappings

Key Input	Command	Description
w, up, arrow_up	{move, up}	Move player upward
s, down, arrow_down	{move, down}	Move player downward
a, left, arrow_left	{move, left}	Move player left
d, right, arrow_right	{move, right}	Move player right
space, e	drop_bomb	Place a bomb at current location
q	ignite_remote	Ignite remote bombs



Bomb FSM:

Enumeration of Events

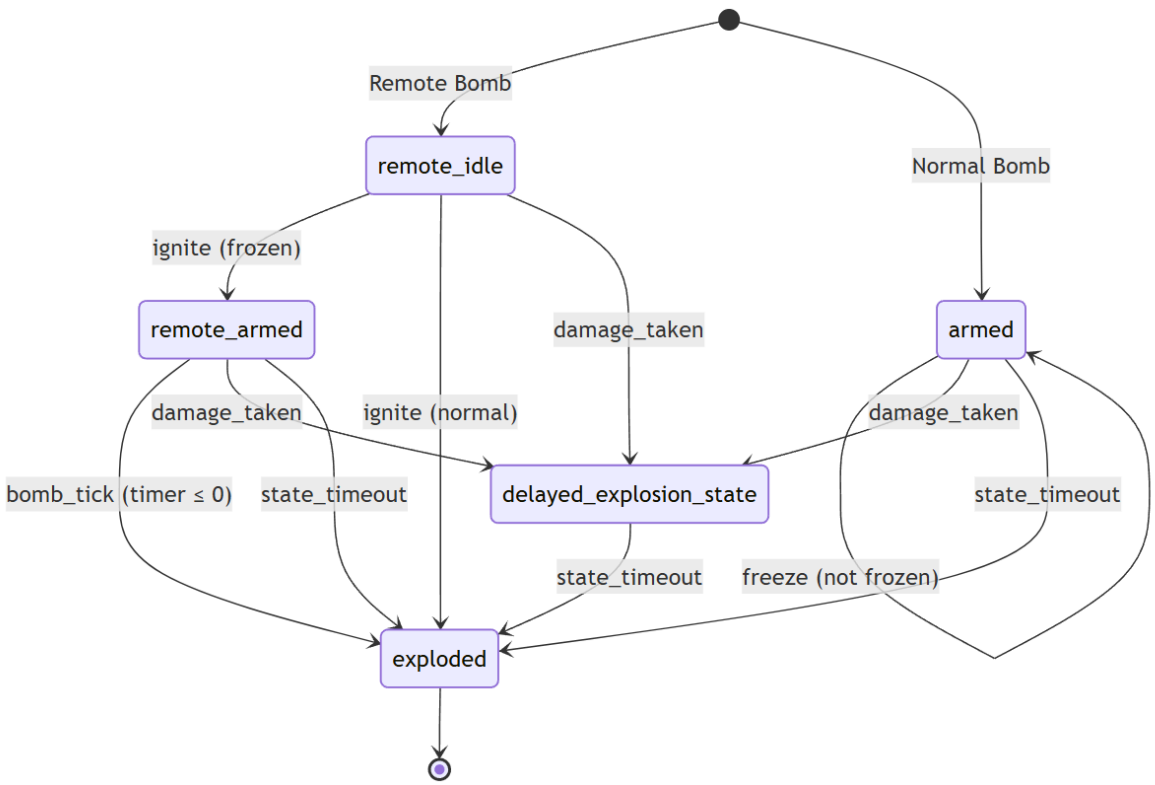
Event	Description	Interrupt Type
freeze	Freeze the bomb, adding delay to explosion timer	Cast
reply_move_req	GN server responds to movement request	Cast
damage_taken	Bomb takes damage and will explode soon	Cast
ignite	Remote ignition signal for remote bombs	Cast
bomb_tick	Internal timer tick (1000ms intervals)	Self
state_timeout	State timeout expired, bomb explodes	Timeout
stop_movement	External signal to stop bomb movement	Cast

State-Level Decomposition

State	Description
remote_idle	Remote bomb waiting for ignition signal. Can be kicked but not exploding yet.
armed	Normal bomb armed and counting down to explosion. Can be kicked, frozen, or take damage.
active_movement	Armed bomb currently moving across the game field while counting down.
delayed_explosion_state	Bomb will explode after one tick delay, ignores most messages to avoid conflicts.
remote_armed	Remote bomb that has been ignited and is now counting down to explosion.
remote_idle_movement	Remote bomb moving while not yet ignited. Can be kicked or stopped.
remote_armed_frozen_movement	Remote bomb that is ignited, frozen, and moving simultaneously.

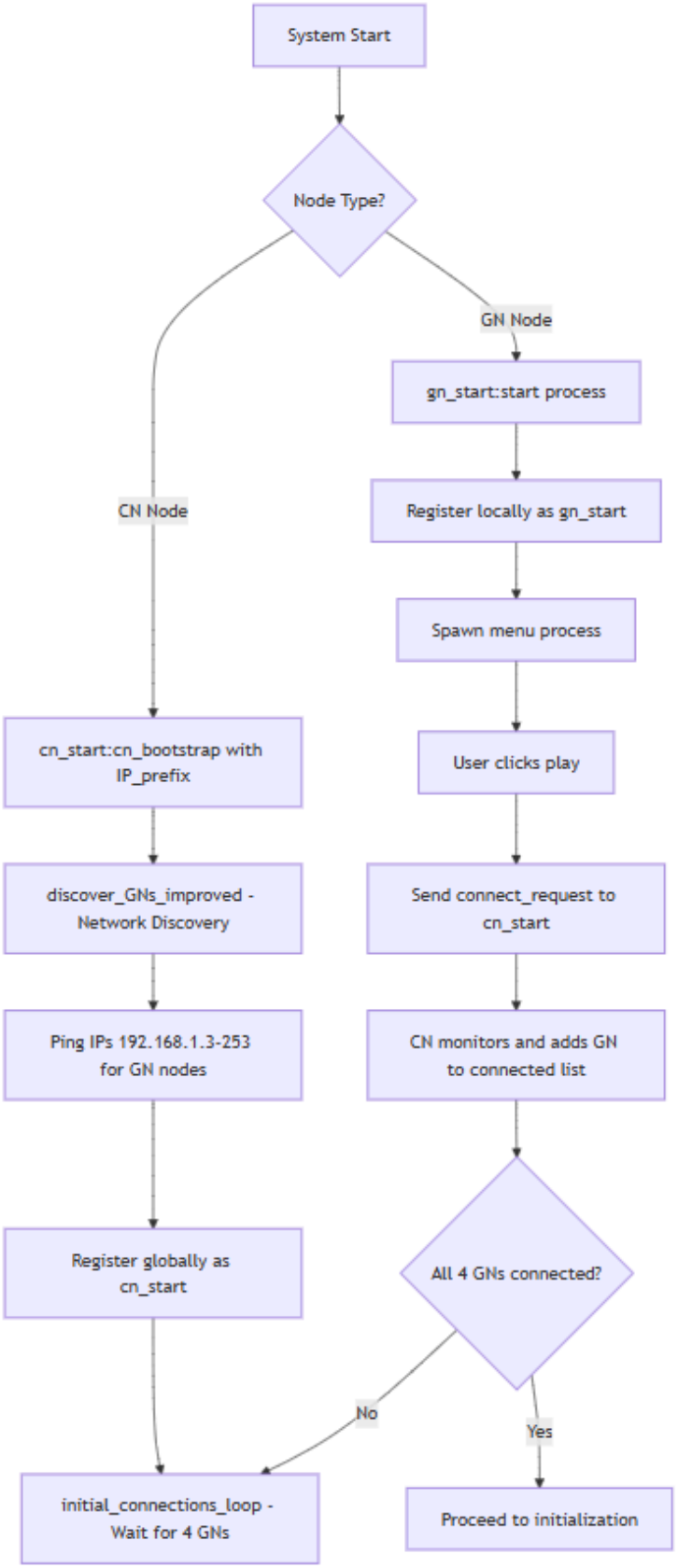
Main Flow Transitions

Current State	Logical Event	Condition	Action	Next State
armed	freeze	Not frozen	Add freeze delay to timer	armed
armed	damage_taken	None	Set explosion delay to one tick	delayed_explosion_state
remote_idle	ignite	Status = frozen	Start freeze delay timer	remote_armed
remote_idle	ignite	Status = normal	Immediate explosion	terminated
remote_armed	bomb_tick	Timer <= 0	Explode bomb	terminated
delayed_explosion_state	state_timeout	None	Explode bomb	terminated

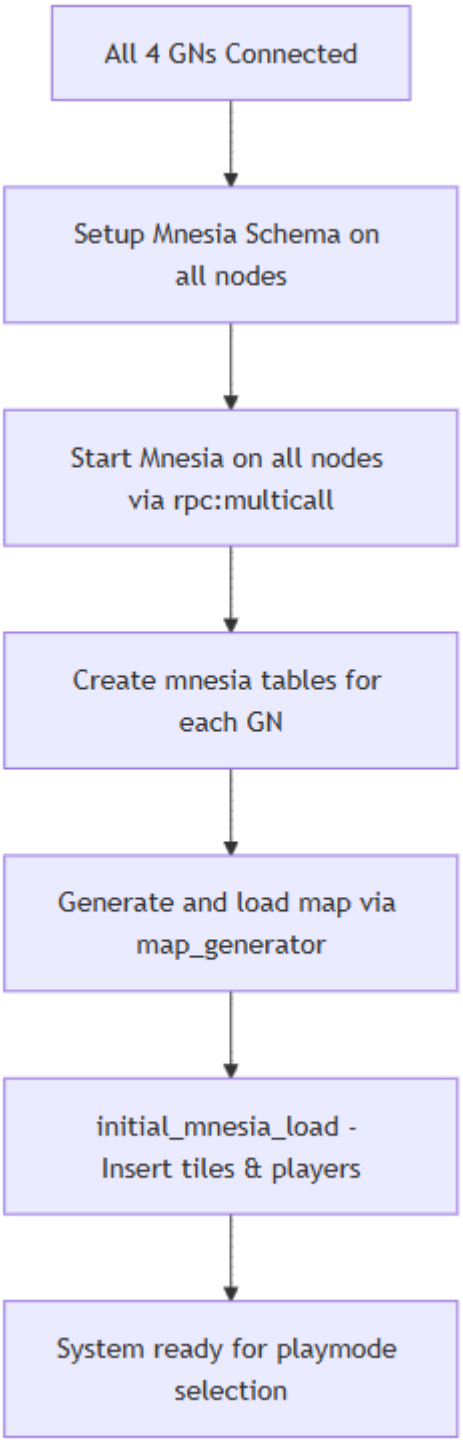


GN And CN Start-Up:

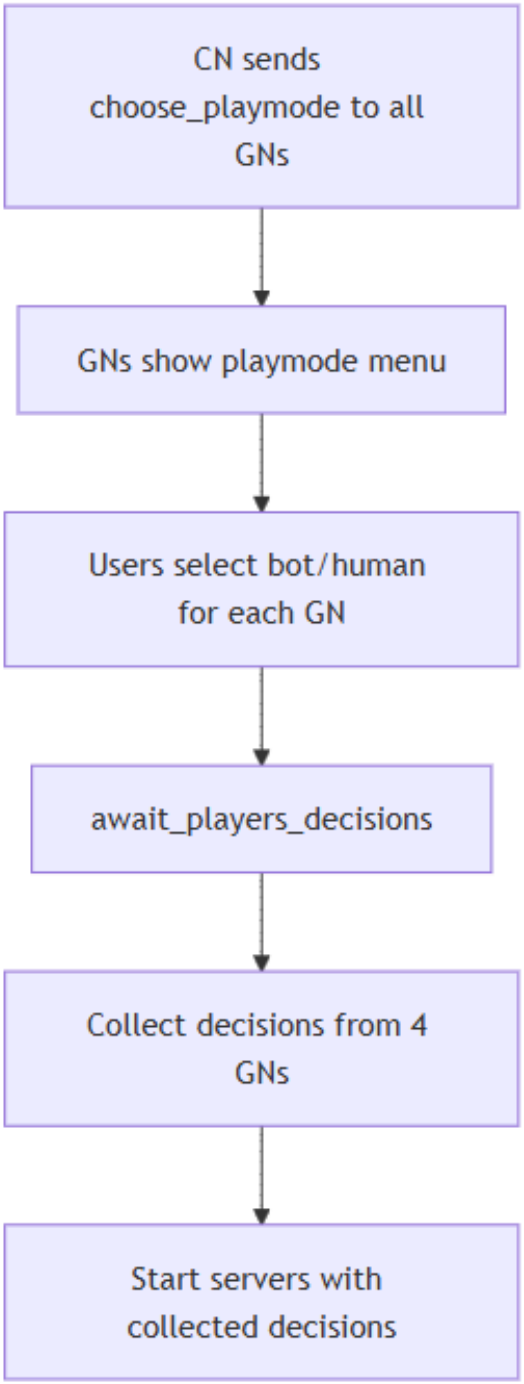
Initial Discovery and Connection:



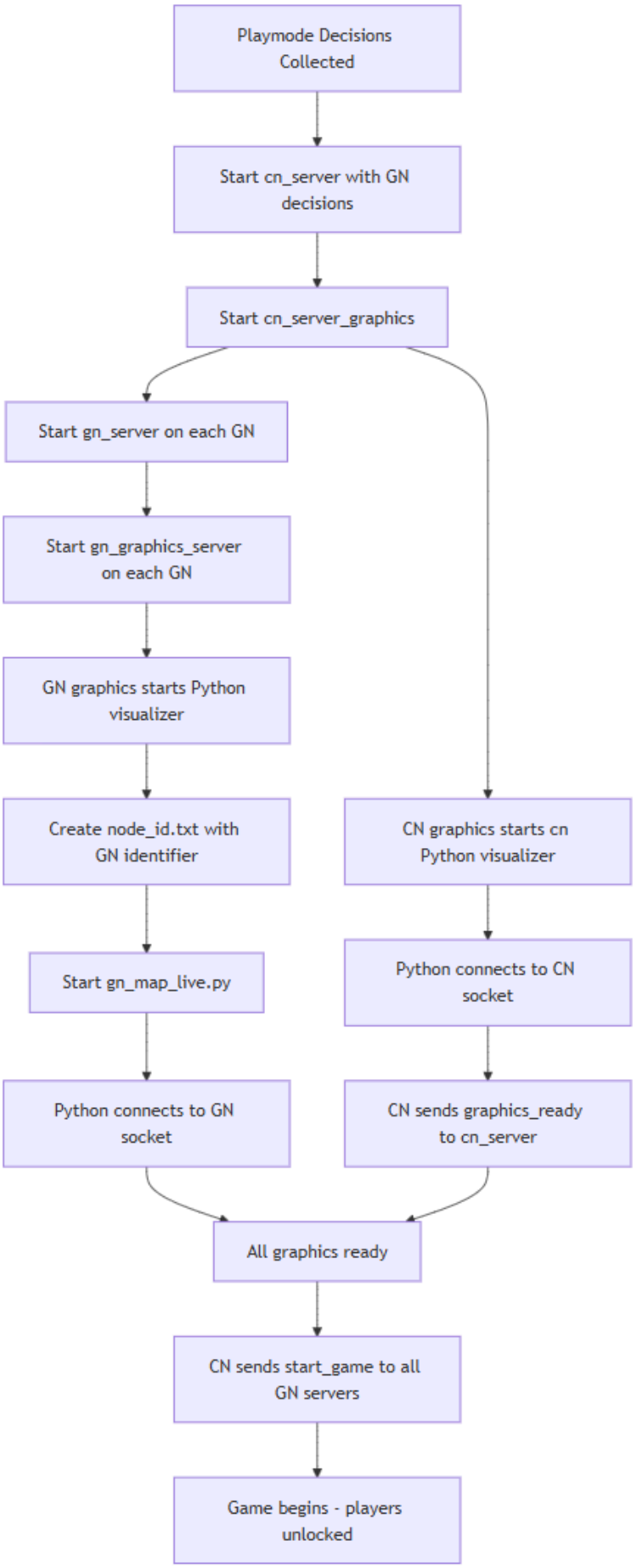
System Initialization Phase:



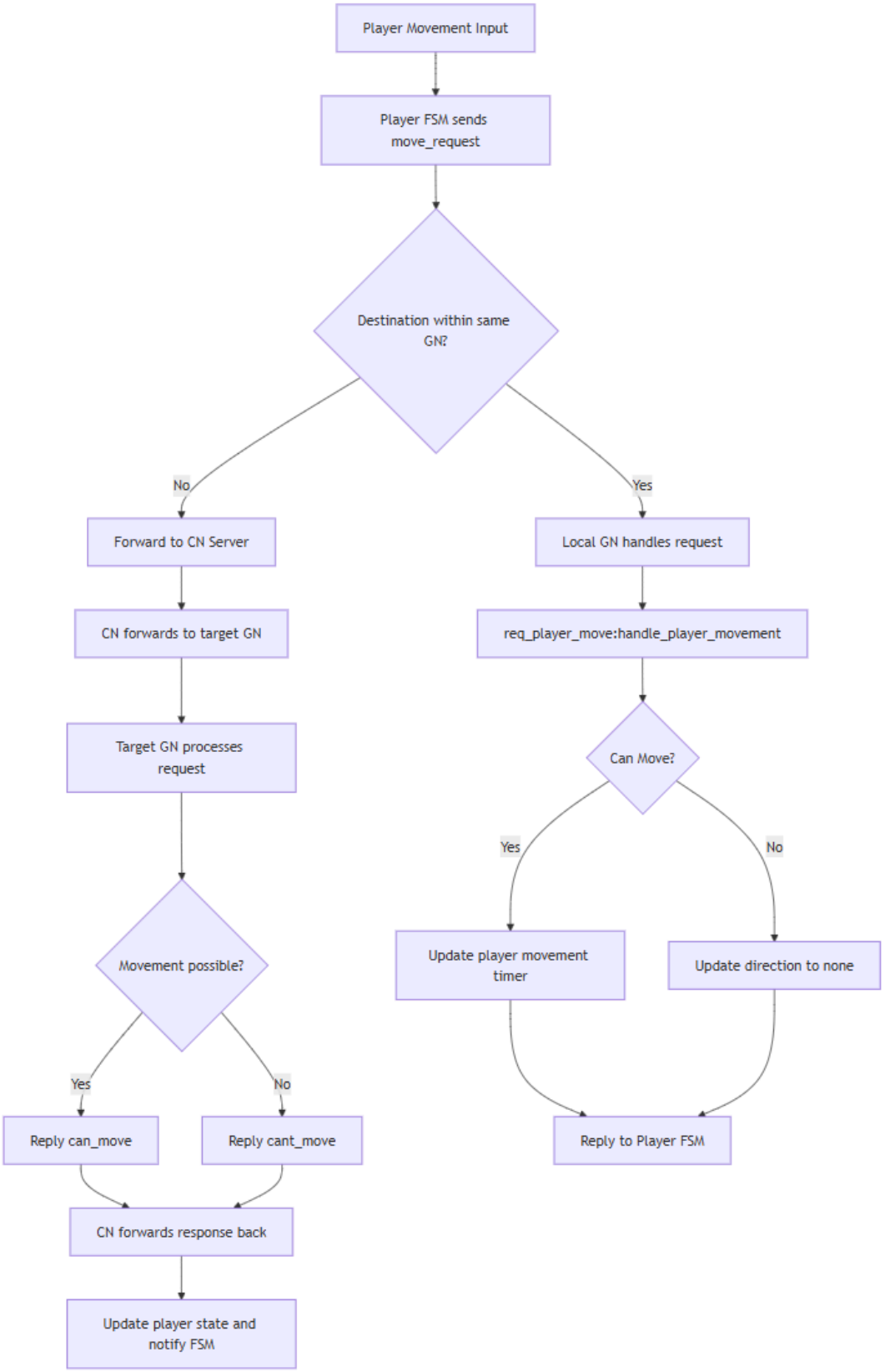
Playmode Selection Process:



Server Startup and Graphics:



Player Request:



Movement Request:

Bomb Request: