Distributed Game of Live

Lukas Faber

January 6, 2019

1 Introduction

Game of Live is a simulation model of living cells in an 2D cellular automaton. Live is modeled in discrete time steps. Based on rules, cells, a single entry in the grid, become alive or die based on the aliveness of its neighbors. In more detail, there are four available transitions for a cell, all of which Figure 1 shows:

Creation A dead cell, which is surrounded with a number of living cells over a threshold α becomes alive for the next timestep.

Isolation Conversely, a living cell dies of isolation if there are no or very few living neighbors around it.

Starvation Similarly, a cell dies if it has too many living neighbors. This can deprive a cell of the necessary resources to survive.

Preservation With a moderate number of neighbors, a cell survives to the next iteration.

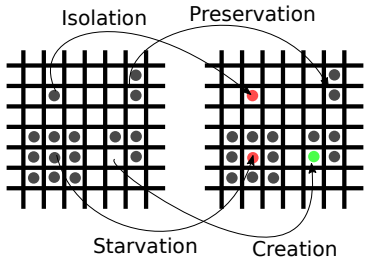

Figure 1: Example depiction of a simulation with the possible state transitions

2 Idea and Architecture

2.1 High-Level Idea

The simulation of this environment is straightforward: we iterate through every element of the grid, count its living neighbors and apply its aliveness for the next time step according to the rules. In the simple case, one computer is responsible for simulating the whole model (see Figure 2).

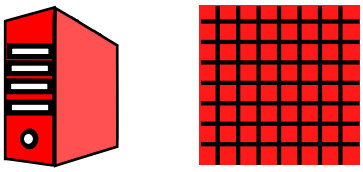

Figure 2: Simulation run through a single computer

If we want to simulate a massive automaton, a nested loop (over all elements in the grid) might become expensive to compute. Additionally, we might hit a storage barrier storing the whole grid. Therefore, we distribute the simulation, splitting the grid up in many smaller subgrids. Each such grid is much more hand-able (since the problem shrinks quadratically). Figure 3 shows this approach. Every subgrid is colored after the computer that simulates it.

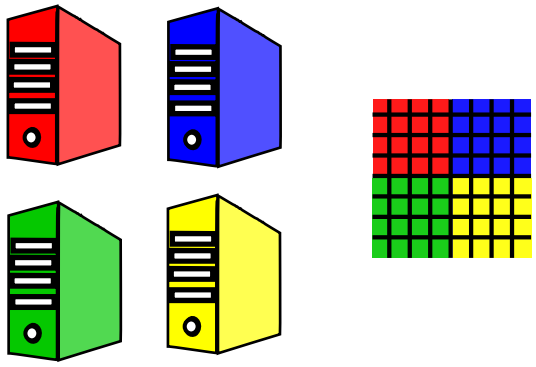

Figure 3: Simulation by multiple coordinating computers

But on the other hand, we also create additional overhead. We have to exchange cell information at the very edges of each subgrid. In Figure 3, cells that lie next to a differently-colored cell must be exchanged since these cells are relevant for the computation of the neighboring grid in the next step.

2.2 System Requirements

1. There should be a user interface showing the simulation.
2. The distribution should be flexible with regards to the number of simulation nodes involved.
3. From an outside persepective, the system should appear as a single system (i.e. the distributed nature should be made transparent).
4. For scalability purposes, no node should be required to materialize the complete automaton (i.e. there should be no complete materialization of the whole grid on a single node).

2.3 Refining the Architecture

Requirement 1

To support this requirement, we create an agent visualizing the current state of the simulation at every timestep. This agent will also act as client in the sense that he starts the simulation and decides on the parameters such as the number of rows/columns.

Requirement 2

We can solve this requirement by adding an additional parameter to the simulation. This parameter specifies across how many nodes we want to distribute the simulation. However, this prevents an A priori creation of distributed networks, as we do not know how many nodes are involved. Therefore, we augment the architecture with a centralized registry service that allows the discovery of available nodes.

2.3.1 Requirement 3 + 4

We add an additional node that coordinates the distributed simulation and surrogates the entire system for the client. For flexibility, we find this node also through the service discovery at the registry. Therefore, every node in the registry is capable of filling both roles (simulation a subgrid or coordinating a simulation) depending on what is requested.
The tasks of the coordinator include finding worker agents for simulation, coordinating their simulation (in particular, exchange the borderline cells), and propagate the model state to the user interfact. At the same time, the coordinator must not reconstruct the overall system state (requirement 4).

2.4 Refined Architecture and High-Level Interaction

Figure 4 shows the refined architecture incorporating the solutions to above requirements. In addition to the working nodes from Figure 3, we add a registry agent (for binding and looking up worker agents), a coordinator agent (for synchronizing the workers) and a client agent (for the user interface). The high-level workflow is as follows: In a first step all worker agents register at the registry. This also includes the coordinator agent. When a client requests a worker, it receives one of the previously registered agents and instructs him to create a game. Only now does this agent know that it becomes a coordinator. It will collect the requested number of workers from the registry and initialize them. Then, the client can send events signalling a new time step. The workers jointly (with the coordinator) simulate this event. The coordinator simultaneoursly returns the new cell states to the client.

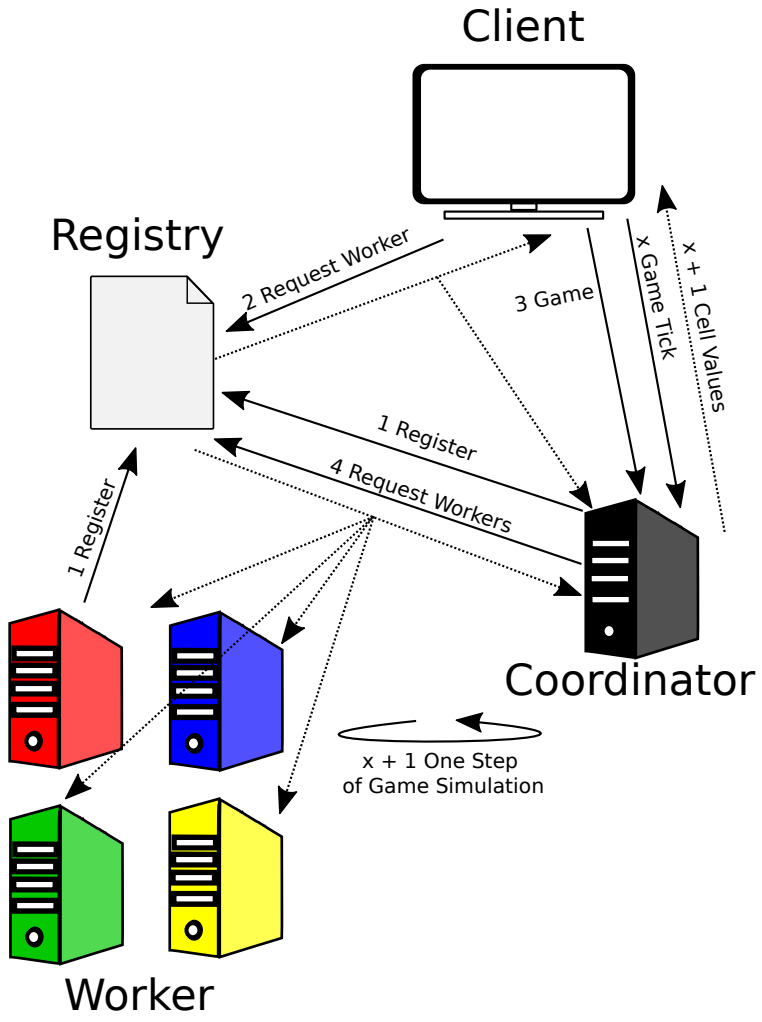

Figure 4: Diagram of the architecture and high-level interaction

3 Coordination Model

In Figure 4 we outlined the high-level interaction schema for the distributed automaton, whereas in this section, we focus on the low-level coordination of the different agents. We use concurrent state machines as the foundational concept for this application, built on socket connections as the technological foundation. One main advantage is its straightforward translation to source code. It also requires only minimal middleware support (see section 4) but could support high heterogeneity in the agents. This is because coordination happens over messages (single-line strings). Thus, we could theoretically have agents of different languages. Nevertheless, all agents will be java based in this example.

3.1 Notation

States are drawn as rectangles with rounded corners. Transitions between states are shown as arrows with three components: a precondition, an inbound message, an outbound message, separated by a semicolon. A transition can only be taken if the agent receives the inbound message while the precondition evaluates to true. If either requirement is empty, it is always considered fulfilled. As part of the transition, the agent also sends, if specified, the outbound message.
Additionally, states can specifiy “ENTER” actions. The agent executes these actions every time it enters this state (either from another state or even the state itself).

3.2 The Registry

Figure 5 shows a state diagramm for the registry. The registry works only in a simple request-reply schema, so a client-server communication would have sufficed. Nevertheless, we choose to implement it in this way, to keep the interaction styles consistent with the other agents. This eases their development components. In the following we briefly glance at the messages the registry sends and receives.

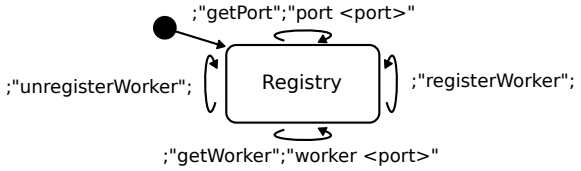

Figure 5: State Machine for the Registry Agent

getPort As we only tested this on a single source machine, we do not have information about valid ports beforehand. Therefore, the registry gives out port numbers to workers and ensures there are no duplicates.

registerWorker/unregisterWorker This message (de)registers a worker for later lookups, where the worker is identified by the port number for its server socket. All agents keep their connection to the registry open. Therefore, they do not need to send their port (as identifier) in each message.

getWorker This message asks for a worker, which is taken from previously registered ones. The worker is also removed from the pool. Therefore, it has to reregister again after completing its previous task. It responds with the port where to contact the worker

3.3 The worker

Figure 6 shows the states a worker agent can occupy. Every worker has to start its lifecycle with requesting a port on which it can run (due to our simulations only being able to run on one machine). Afterwards, it registers on this port and is `Available` to serve requests. Depending on the request coming in, it will serve as a coordinator or continue as a worker.

3.3.1 Coordinator

A worker becomes a coordinator when it receives a "createGame" message. This message additionally entails the number of workers the simulation should use and the grid dimensions. The worker will switch into the `Creating` state.
First, it will look up the specified number of workers from the registry, partition the overall grid, and send each worker agent a command to initialize their part. After all are done (signaled by an "ackInit" message), the coordinator transitions into the `Synchronize` state.
In this state, the coordinator ensures that every worker receives the cell values of cells next to its subgrid. It requests all cell values from all subgrids and redirects them to the corresponding other workers. At the same time, the coordinator pushes all values to the client for display.
When all workers acknowledge having received all necessary neighbors, the coordinator transitions to the `TickAwaiting` state. Upon receiving a "gameTick" event, it signals every worker a tick and waits for their completion in the `Ticked` state. Afterwards, it transitions back to the `Synchronize` state to inform the workers and the user interface about the new cell values.

3.3.2 Worker

A "initField" message decides that a worker does not become a coordinator but instead simulates a part of the grid. It receives the coordinates for the part it is responsible for. The worker randomly initializes

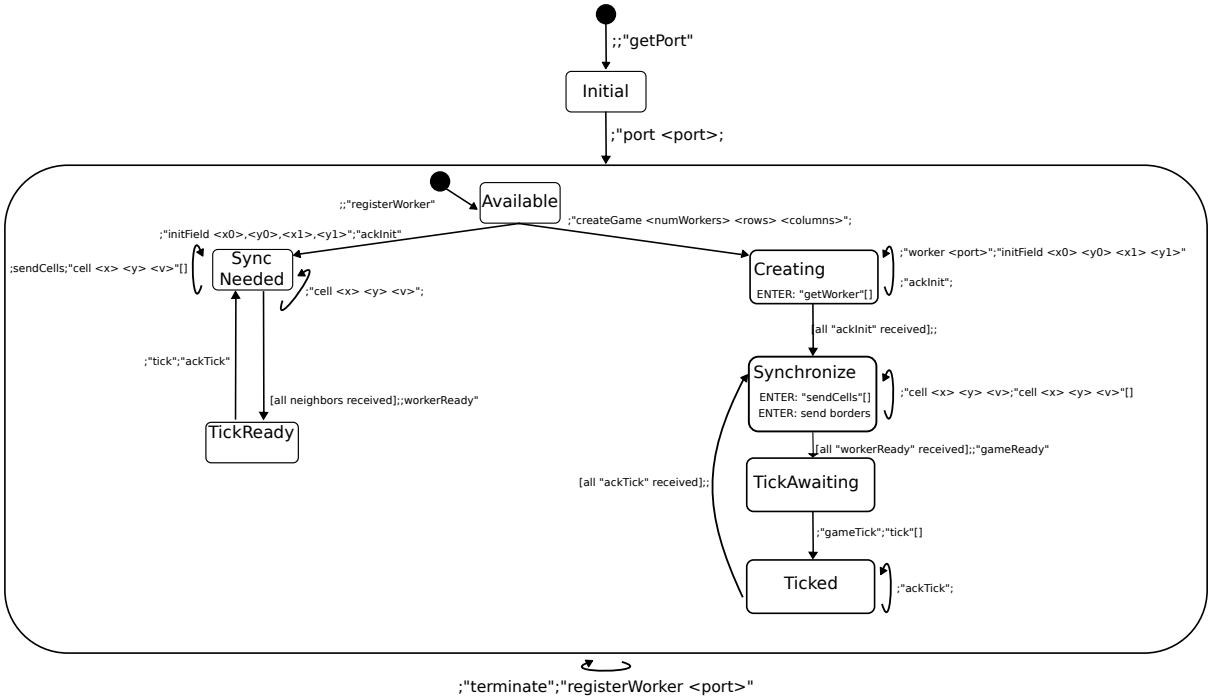

Figure 6: State Machine for the Worker Agents

every cell in its grid and then returns "ackInit" message to signal its completion to the coordinator. Then, a worker transitions to the `SyncNeeded` state. Upon receiving a "sendCells" message, it sends a "cell" message for every cell it simulates, while also waiting for "cell" messages that contain information about its immediate neighbors.
When all neighbors are received, the worker is ready for a tick and transitions to the `TickReady` state, while signaling its readiness to the coordinator. Upon receiving a "tick" message, it evaluates the new state of every cell in its subgrid based on rules like in section 1. Upon completion it sends an "ackTick" message to the coordinator and transitions back to the `SyncNeeded` state.

3.4 The Client

Figure 7 shows the coordination scheme for the client agent. This agent starts by requesting a worker agent that will serve as the coordinator. Upon receiving its address, it tasks the agent with creating a game. Afterwards, it waits to receive "cell" messages which inform the client about the cell states. Only after a "gameReady" message will the client react to user actions which trigger a game tick. Then it will transition back to the `waitingForCells` state to gather the updated cell information.

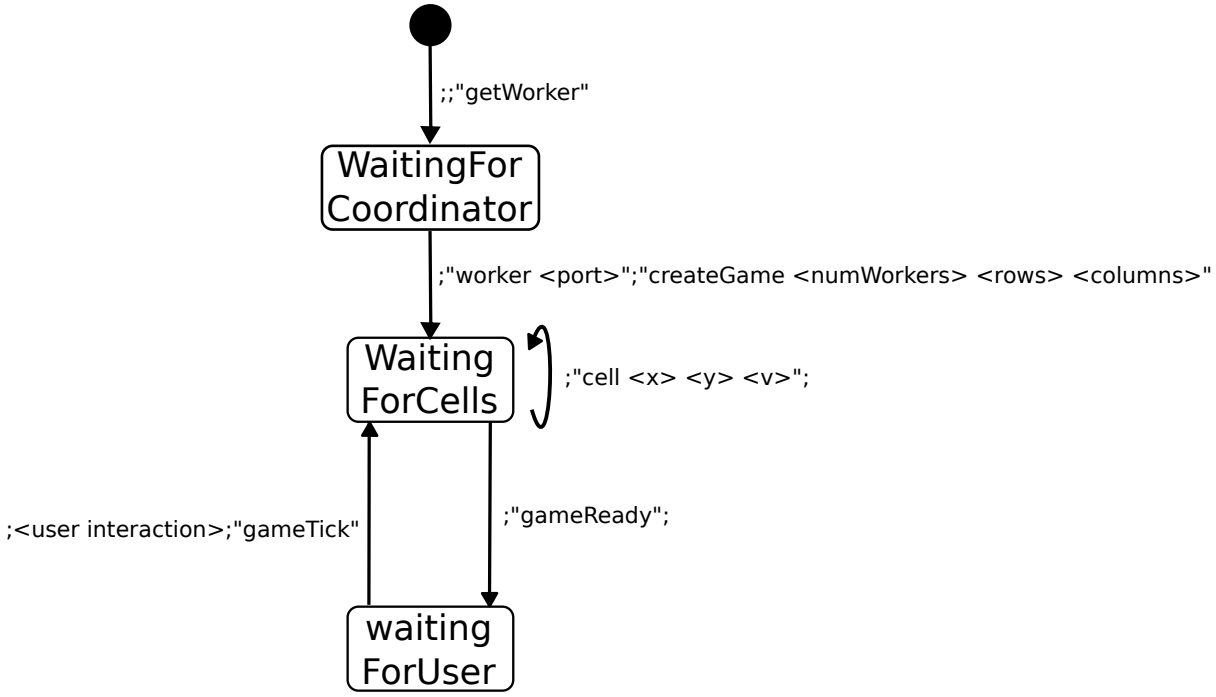

Figure 7: State Machine for the Client Agent

4 Employed Middleware

Our middleware has to support message passing among established connections. We build on top of the existing socket technology from the java socket library. We build our own very basic middleware based on an interface (`CallbackAble`) and a java class (`SocketIO`). The appendix includes their source codes in Appendix A. A class that wants to use our middleware must comply with the interface and implement a `callback method`. In this method the class should specify its reaction on incoming messages.

The `SocketIO` class abstracts a single socket connection for a given socket. It provides an easy method to write a single-line string (our messages) over the connection. At the same, time it listens for incoming messages in a separate thread (since receiving data is a blocking action) and triggers the `callback` method of its parent agent so it can handle the message contents.

A Source Codes

A.1 Callable

```
public interface CallbackAble {

    void callback(String message);
}
```

A.2 SocketIO

```
public class SocketIO implements Closeable{

    private BufferedReader in;
    private PrintStream out;
    private Socket socket;
    private Thread listener;

    public SocketIO(Socket socket, CallbackAble c){
        try {
            socket.setReceiveBufferSize(2097152);
            socket.setSendBufferSize(2097152);
            this.socket = socket;
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintStream(socket.getOutputStream());
            listener = new Thread(() -> {
                while (!Thread.interrupted()){
                    try {
                        String message = in.readLine();
                        if (message == null){
                            message = "terminate";
                            c.callback(message);
                            socket.close();
                            break;
                        }
                        c.callback(message);
                    }
                    catch (SocketException e){
                        c.callback("terminate");
                        break;
                    }
                    catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            });
            listener.start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void writeLine(String text){
        out.println(text);
    }

    public void close(){
        try {
            listener.interrupt();
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```