16 System Overview

J. Mitchard

16.1 Supervisor-Child Architecture

The swarmer application has been implemented in a hierarchical process structure using Erlang Supervisor processes[4] that comes as part of the OTP[5] [6] (Open Telecom Platform). This means that all processes are children of a supervisor process within the applications' supervision tree. Supervisors are responsible for the stopping, starting and restarting of their child processes. The swarmer application has been implemented in a hierarchical process structure using Erlang Supervisor processes[7], a behaviour that comes as part of the OTP[8] [9] (Open Telecom Platform). This means that all processes are children of a supervisor process within the applications' supervision tree. Supervisors are responsible for the stopping, starting and restarting of their child processes. Swarmer has one main supervisor, called the swarm_sup, this is responsible for the tile_sup, viewer_sup, human_sup, zombie_sup, supplies_sup supervisor processes and the environment process.

Apart from swarm_sup, all of the supervisors are created with a simple-one-for-one restart strategy, meaning that each of the children will be identical processes using the same code, and should be restarted if they crash. swarm_sup is using a one-for-one strategy, which can restart its child processes without affecting the others[10].

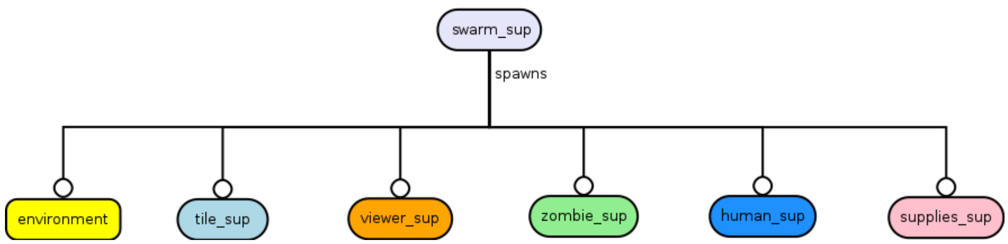

Figure 6: Supervision Tree for Swarmer

16.1.1 OTP Behaviours

Using an Erlang and OTP application with a process supervision architecture provides a standard set of interface functions, behaviours and more advanced error tracing and reporting functionality.

The backbone of swarmer is created using the gen_server[11] (Generic Server) behaviour. gen_server provides a framework for reliable and robust message passing between processes, using either synchronous requests called calls, or asynchronous request called a casts. The environment, tile, viewer and supplies modules have been implemented with gen_server behaviours.

The behaviour of the human and zombie entities in the system are modelled around the gen_fsm[12] (Generic Finite State Machine) behaviour. gen_fsm provides a state machine for the entities to use, and incorporates synchronisation events, such as pause and unpause, for the rest of the system to call. The gen_fsm processes will, once started, run until told to stop. The human_fsm and zombie_fsm modules have been implemented with gen_fsm behaviours.

4 http://www.erlang.org/doc/man/supervisor.html
5 http://www.erlang.org/doc/design_principles/des_princ.html
6 http://learnyousomeerlang.com/what-is-otp
7 http://www.erlang.org/doc/man/supervisor.html
8 http://www.erlang.org/doc/design_principles/des_princ.html
9 http://learnyousomeerlang.com/what-is-otp
10 http://www.erlang.org/doc/design_principles/sup_princ.html#id68643
11 http://www.erlang.org/doc/design_principles/gen_server_concepts.html
12 http://www.erlang.org/doc/design_principles/fsm.html

16.2 System Architecture

As introduced in section 16.1 `swarmer` is built around a Supervisor-Child architecture. This section will explain the setup of the application in a little more detail.

When the application is started, the initial process is an instance of the `swarm_server` module. This is responsible for starting the Websocket and spawning the system supervisor module, `swarm_sup`. The system uses the Cowboy[13] HTTP server to manage Websocket communication to the user client, which is covered in section 17. The web socket is managed by a module called `swarm_handler`.

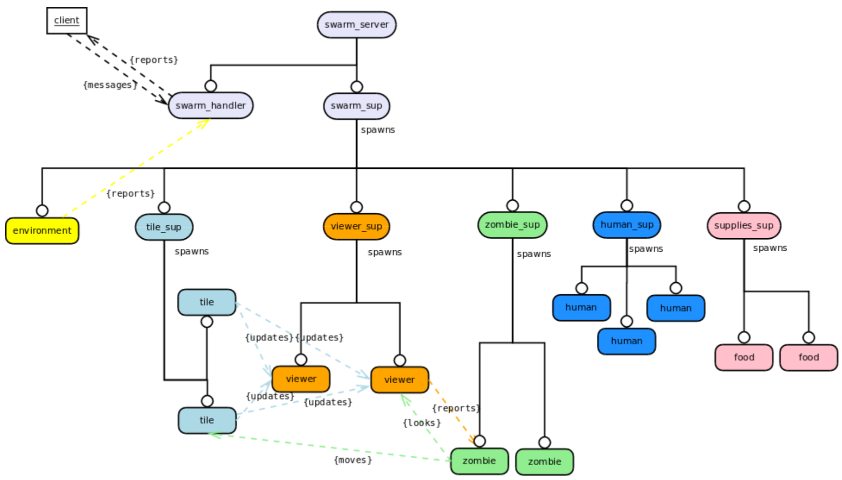

Figure 7: System Architecture

On initialisation, the `swarm_sup` will spawn the children processes shown in figure 6 and the system will wait for a message from the client to define what to spawn, this will be covered in section 16.3. Once received, it will then continue waiting until `swarm_handler` receives a message to start the system.

In order to visualise what is happening in the system, the client systematically requests a report from `swarmer`. To report back to the client, the `environment` module requests the status of all of the supervisors children nodes. This is returned to `swarm_handler` which is encoded into a JSON object using the `jsx` dependency and sent to the client.

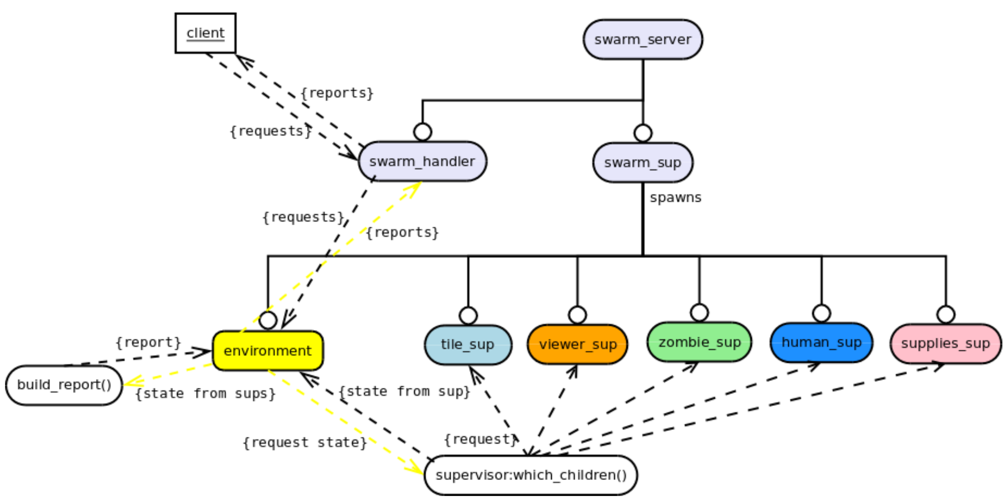

Figure 8: Reporting to the Client

[13] http://ninenines.eu/docs/en/cowboy/1.0/guide

The map is created out of a grid of `tile` instances with assigned `viewer` instances. This is explained in more detail in section 18.1. When the entities are initially spawned they are initialised in a paused state until the system is told to start. After this, the entities begin to run carrying out their type specific behaviours. The way in which entities interact with the environment is explained in detail in section 18.3. Zombies, covered in section 20, in the simulation are created as an instance of the `zombie_fsm` module, and attempt to find human entities in the environment. Over time they will slow down if they have not eaten in a long time, and can turn human processes into zombies if they succeed in killing one, thus spreading the swarm. Humans on the other hand are trying to survive, meaning they must scavenge for food and escape the horde of zombies. Their behaviour is covered in section 21.

16.3 System Setup

The initial set-up of the simulation environment is handled by the `environment` module. This deals with spawning `tile`, `viewer`, `supplies`, `human_fsm` and `zombie_fsm` processes according to the set-up instructions received from `swarm_handler`. On a normal set-up, the system would be told to create an amount of tiles, defined by a 'grid arrity' and an amount of each entity type to spawn. Grid arrity would be an integer between the value of 1 and 10, 1 telling the system to build a 1x1 grid and 10 telling the system to build a 10x10 grid.

Obstructions, which are covered in section 19, play a big part in the way our simulation runs, they block line of sight and prevent movement through them. During the initial setup, this is created through a list of blocked coordinates passed into the tiles on initial set-up.

16.3.1 Making the Grid

When the `environment` process receives a `make_grid` request it will firstly purge the system of currently spawned processes; restarting the system from the ground up. This prevents old instances of processes remaining in the supervision tree.

The process will then proceed to create a grid of given arrity by spawning `tile` instances of a given size. In order to create a grid like structure, a row of tiles is spawned, assigning each an origin coordinate and an end coordinate before creating another row of tiles. All `tile` processes are registered to Erlang as a named process of the form "tileXOYO", O being the origin point for each axis. Each of these then has a viewer process assigned to it.