Assignent Week 5
Swarm Simulation using Quadtrees

Course: Advanced Programming, Block 1, 2013

Deadline: 20:00, October 6, 2013

Objective

The objective of this assignment is to gain hands-on programming experience with distributed computing in Erlang and insight to some of the problems arising in a distributed model. The goal implement a simple distributed quadtree and a fish school in this.

The assignment consists of 2 parts, and an optional part:

- Part 1: Implement a distributed quadtree.
- Part 2: Use your quadtree framework to implement a simulation of a fish school.
- Part 3: Suggestions for various extensions. This part is **optional**, it will have no influence whatsoever on your grading.

What to hand in?

You should hand in two things:

1. Your code.
2. A short report explaining the code, and an assessment of the quality of code including what this assessment is based on.
3. In your hand-in you are expected to also provide your tests.

Scope

The following topics are *not* within the scope (of Part 1 and 2) of this assignment:

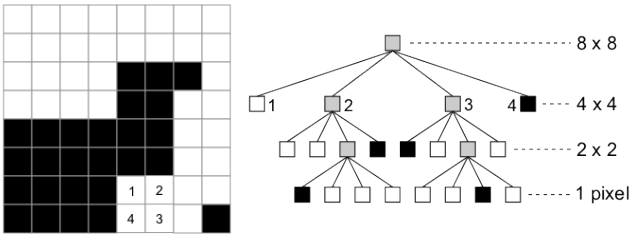
- Network reliability: you do not have to consider what to do if a process crashes.

Part 1: Distributed Quadtree

In this part your task is to implement a framework for a *distributed quadtree*, on which we later will apply a *swarm behavioural simulation*. All implementation must be in Erlang. Two skeleton files (`quadtree.erl` and `swarm.erl`) are provided and must be used. Also a print-module (in `printer.erl`) that writes an SVG-file is provided: this is useful to visualize quadtrees. You can display SVG-files in e.g. your favourite browser.

Quadtree

A quadtree is a special tree data structure, which is often used in computational geometry. Compared to a "normal" (binary) tree, a quadtree is characterised by each node having exactly four children. This gives an obvious geometric interpretation of the quadtree, where each child of a node represents one-fourth of the area that is covered by the node. We will here divide the area as rectangles, where the four children of a node is represents the fourth-parts: north-west, north-east, south-west, and south-east. All children, thus, have bounds of equally sized rectangles (down to the floating number representation in Erlang). Only the leaves of the tree contain data. Figure 1 shows an example of a quadtree that implements a small b/w bitmap.



Figur 1: Example of a quadtree. *Source: Wikipedia.*

Implementation in Erlang

In Erlang you must implement a framework for a distributed quadtree, where each node and leaf is implemented as an Erlang process. In the tree, nodes must *only* know about and communicate directly with its direct parent and children, just as a leaf must *only* communicate directly with its parent. Thus, you must uphold the tree structure and, for example, cannot create a list of all leafs and use this for processing.[1]

Elements of the Quadtree

Only the leaves in the quadtree contains data, which is a list of *Element*s; in the simulation this will be fish. Each *Element* is defined by a position (X, Y)

[1] This communication structure is not much different from the actual implementation of the Internet. Also, it can better communication if you divide nodes optimal across your cluster at home in your basement...

and some *Element*-properties. We will define an element to have the following tuple-type:

$$\texttt{\{element, \{}X\texttt{,}Y\texttt{\},}Property\texttt{\}}.$$

We will later describe how to process *Element*s and update the position and properties.

The Quadtree

When a distributed quadtree is created it only consists of a single leaf with no data elements. Over time more *Element*s can be added to the tree, which will increase the load of the leaf to which they are added. Therefore, when the number of elements in a leaf-process exceeds a given *Limit* it must transform itself to a node and create four new leaf-processes to which its elements are distributed. The quadtree is, thus, not fixed-sized, but must dynamically grow when more elements are added. **Note, that in this implementation quadtrees only grow (when adding more elements), but does not shrink when elements are removed from a leaf-process.**

When a distributed quadtree is created, it (apart from the *Limit*) is also initialised with a *Bound*. This *Bound* defines the area in which elements in tree can be placed. If an element is moved outside this *Bound* (for the entire quadtree) it is lost. If an element is moved outside the bound of a leaf, it must be moved to leaf that covers the area in which the element now is positioned. A bound can be three different things:

- a rectangle that is defined by its upper left (X_1, Y_1) and lower right (X_2, Y_2) corners. This must be represented in Erlang as the tuple:

$$\texttt{\{}X_1\texttt{,}Y_1\texttt{,}X_2\texttt{,}Y_2\texttt{\}}.$$

- the entire `universe`, which is defined by all the area by which the quadtree was initialised. `universe` must be the Erlang atom.

- `empty`, which is the bound that spans no area. Again `empty` must be the Erlang atom.

When a quadtree is create it must be created with a bound as a rectangle and, thus, all bounds of nodes and leafs must also be rectangles.

Your implementation must be efficient in the way, that map-functions and elements must only be send to the specific nodes / leaves as specified by the *Bound*s; you should, for example, not broadcast messages to all leaves if this is not necessary.

The quadtree must uphold the following API

- A function `start(`*Bound*`, `*Limit*`)` that creates a quadtree with elements within the given *Bound* and with *Limit* as the maximum number of elements in each leaf.

- A function `stop(`*Qtree*`)` that stops all processes in the quadtree.

- A function `addElement(`*Qtree*`, `*Pos*`, `*Prop*`)` that adds a new element to the quadtree *Qtree* at the position *Pos* with the properties *Prop*.

- A function `mapFunction(`*Qtree*`,` *MapFun*`,` *Bound*`)` that maps the function *MapFun* over all *Element*s that is within *Bound*. Thus, *MapFun* must be a function from an *Element* to an *Element*.

- A function `mapTreeFunction(`*Qtree*`,` *MapFun*`)` that maps the function `MapFun` over the bound of each node and leaf in *Qtree*. A quadtree SVG printer are available and this function is needed to print the structure of the quadtree. The mapper-function is not expected to return anything and must not do any updates to the quadtree.

Applying a map-function (both to tree and element) must be robust, such that a node or leaf does not crash if a function call fails.

In the skeleton file, functions which body contains `....` are the one that you need to implement. Also there are some suggestions for helper functions which you can implement and use (or not).

The Quadtree Coordinator

Located above (considered parent to) the topmost node (or leaf if the tree consists of a single leaf) is coordinator process. The main function of this process is to provide the interface between the quadtree and the "outside world". Having such a process separating concerns, such that nodes and leafs does not have to consider if they are topmost of not.

Part 2: Simulating a Fish School with Swarm Behaviour

From the outside the behaviour of fish schools and insect swarms looks unpredictable, but by modelling *each* fish (or insect) using some very simple rules it is possible to have good simulations of school (or swarm) as a hole. In this part you must create a quadtree and implement a function that performs the swarm behaviour simulation. More specifically you must use the quadtree API to map a function the calculate the updates specified below.

The basic idea is that each fish can locate all other fish within a certain distance. If another fish is too close it will try to get further away from it; but a fish would still like to be in a school, so it will try to get closer to fish that are further away. See Figure 2.

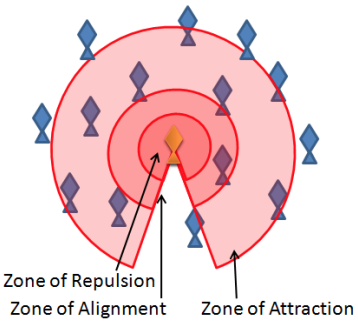
Properties of a fish

Apart from the obligatory position for an element, a fish will have as property

- a unique ID, such that we always can locate a specific fish

- a velocity vector (V_x, V_y) that determines the direction and speed of the fish; we will define the maximum velocity of a fish to 3, meaning that this is the maximum length of the vector.

- a velocity change vector (D_{vx}, D_{vy}) and an integer N_d that contains the number of times the change vector has been updated. We will discuss how these are used in a moment.



Figur 2: Focal fish pays attention to all fish within a certain distance. *Source: Wikipedia.*

You must use the following tuple-type for a property: {ID, {Vx,Vy}, {Dvx, Dvy}, Nd}.

The simulation

The simulation will work in two time stepped phases:

1. A move phase: This moves all fish according to its velocity and velocity change vectors.

2. A attraction / repulsion phase: Changes the velocity of a fish according to the other fish that are within it zones.

These two phases will repeat one at a time every 50 ms to make a simulation that runs in time steps. There is no guarantee (except from the large time difference) that one phase has completed all operations before the other phase starts.

Attraction / Repulsion Phase

Attraction and repulsion happens by adding or subtraction a vector to / from the velocity change vector, while incrementing the number of changes. In practice this vector will be a unit vector with a direction from the fish to the fish in the attraction or repulsion zone. You are not asked to implement this, so these functions are available in the skeleton.

We will define the repulsion zone the have a radius of 3 (points!) and attraction zone to lay in a radius between 7 and 10. You must implement that this attraction / repulsion is calculated between the right fish, and ensure that this information is applied to the fish at the centre of the zones. Remember to use the tree API for this.

This means that the change in velocity can be at most 1, which models that a fish can not change direction arbitrarily.

Move Phase

A move must applied to all fish "at once". It will add the velocity change vector to the velocity vector (with the limit on velocity). Also the change vector and

number of changes are reset to zero. The skeleton also provides a function that applies this to one element.

Again you are to implement how this is performed over all fish in the school (quadtree).

Assessment

Your hand-in should contain a short section giving an overview of your solution, including **an assessment** of how good you think your solution is, and how you have tested you solution. It is also important to document all *relevant* design decisions you have made. Likewise, if you have failed to complete your implementation, describe in detail the implementation strategy and tests you had planned.

Hints

- Start by thinking about which messages the different kinds of processes should send to each other and what information these messages should contain. Then design a representation of these messages.
- Then consider what the quadtree looks like and (informally) write down paper what effect each type of message will have on nodes and leafs. Take special care when a leaf transforms to a node.
- Make yourself familiar with the manual page for lists and the manual page for dict for information about functions in the lists and dict modules. It will come handy.
- While developing the program I've found it helpful to use `io:format` to print various information to the console. For example, my `coordinator_loop` had the following as the first line for a while:

  ```
  io:format("~p is coordinator with the mappers ~p ~n", [self(), Mappers]),
  ```

  (Yes, it's a side-effect but it's useful.) See the manual page for the io module for more information about control sequences available in the format string (like ~p in the example).

Part 3 (optional): Extensions

Here are a couple of suggestions for extensions to the assignment, in no particular order.

- Things can move out of the known bound. If element is moved of of the known bound the coordinator is told and it can extend the area of the quad tree by 4 times (of what is needed) and spawn the nodes and leafs needed for the extended. What is needs to be changed and (perhaps) implement this.

- Quadtree that can both grow and shrink. The total number of element in the leafs of a (bottommost) node can easily become less that the *Limit*. Devise a strategy that can solve this (consider and argue what might be most efficient[2]), outline the changes needed, and (perhaps) implement this.

[2]Hint: spawning and killing processes a lot of processes can be more costly that keeping some unused for some extra time.