M1 Project - Distributed Systems

Simon Mauras

April 23, 2016

Contents

1 Introduction

This report describe the implementation in Erlang of a small middle-ware using a treap topology. This project is part of the evaluation of a master's degree lecture at ENS Lyon.

After a first part in which explicit what a treap is, we are going to explain in detail the functionalities of our middle-ware. A third part will be devoted to give installation instructions. We will conclude by answering to the questions asked in the project assignment.

2 Treaps

2.1 Definitions

The word treap is a portmanteau formed by the words tree and heap. The idea is to combine those two data-structures to build a new object having nice properties.

We recall that a Binary Search Tree (BST) is an indexed binary tree in which the value of each node is bigger than the values of the nodes of its left sub-tree and is lower than the values of the nodes of its right subtree. A heap is an indexed tree in which the value of each node is bigger than the values of its subtree.

If we combine those two kind of trees we can define a new data-structure in which each node is associated to a couple of values, the tree is a BST according to the first coordinate and a heap according to the second.

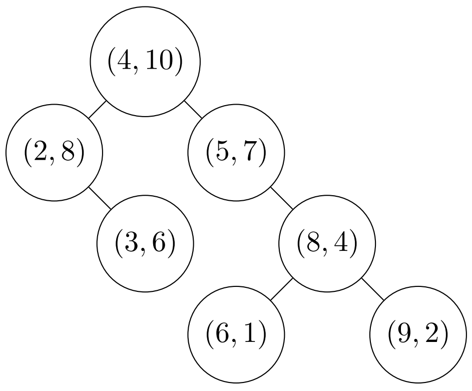

Figure 1: Example of Treap.

We can notice that a treap is unique and can be build greedily. Indeed we have only one candidate for the root (heap property) and all the other nodes can be dispatched between the left and the right subtree (BST property).

One other important remark is that if we choose the second coordinate at random, we get a tree that have a high probability to be **balanced**. The analysis is exactly the same as the one used to prove the amortized complexity of the randomized quicksort.

2.2 Operations split and merge

We can now define two operations on the treaps. The operation **split** takes one treap and one threshold as input and split the treap in to using the threshold on the first (BST) coordinate.

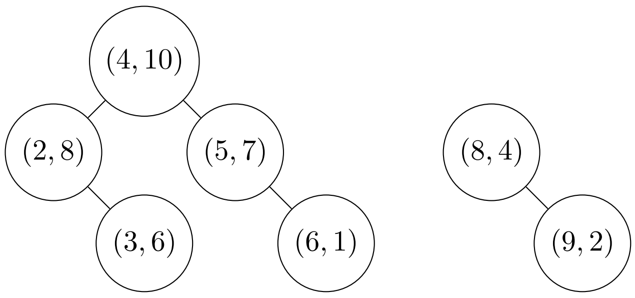

Figure 2: Example of Split with a threshold of 7.

The operation **merge** is the reverse operation. We take two treaps as input, assuming that the values of the first coordinates of the nodes of the first treap are lower than the one of the second treap, and we merge them to get back the initial treap.

We can now give a small pseudo-code for those two operations. It is very easy to write such a code in a functional language.

```
Function merge(treap A, treap B):
  If A is empty: return B
  If B is empty: return A
  If A.heap > B.heap:
    A.right = merge(A.right, B)
    return A
  else:
    B.left = merge(A, B.left)
    return B

Function split(treap T, threshold X):
  If T is empty: return T
  If T.bst <= X:
    A, T.left = split(T.left, X)
    return A, T
  else:
    T.right, B = split(A, B.left)
    return T, B
```

The amortized complexity of those operations is in $\mathcal{O}(\log N)$ where N is the number of nodes. Indeed if we sample the value `heap` randomly, the tree is balanced.

2.3 Applications

Now that we have implemented operations `split` and `merge`, we can easily deduce operations `insert` and `remove`.

```
Function insert(treap T, node N):
  A, B = split(T, N.bst)
  return merge(A, merge(N, B))

Function remove(treap T, value X):
  A, B = split(T, X)
  _, C = split(B, X+1)
  return merge(A, C)
```

We can now notice that with the assumption on the input of the merge function, values of the **first coordinate** (BST) of each node could be implicit. Indeed the structure of the tree imposes an order (infix) on the nodes. So we could replace it by the **size of the subtree** (updated after during each operation). The split threshold now becomes the rank of the value for which the cut is done.

This data-structure is very easy to implement in a classical language of programmation. In the next part we are going to explain how we managed to implement it in Erlang, each node of the tree being handled by a different process.

3 Middle-ware

3.1 Global overview

The organization of the project is the following. Source files are stored in the `src` directory. The `Makefile` can be used for the compilation and agent deployment. The python script `slsu.py` can be use to extract hostnames from `.hosts.erlang` and launch an agent on each of those nodes.

```
+-- bin
|   +-- .hosts.erlang
|   +-- compiled files...
+-- src
|   +-- agent.erl
|   +-- jobs.erl
|   +-- monitor.erl
+-- Makefile
+-- slsu.py
```

The file `agent.erl` contains all the internal functions used by the agents. We can describe an agent by a finite state machine.

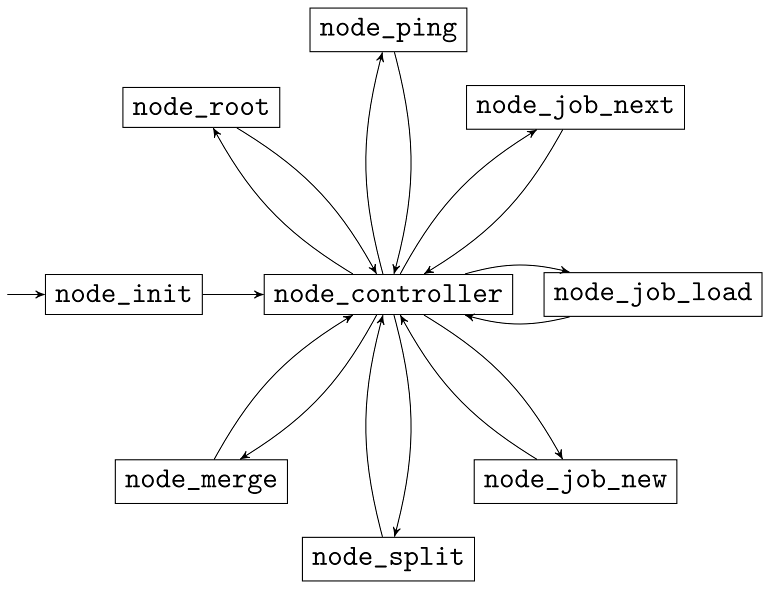

Figure 3: Finite State Machine describing an Agent.

- `node_init` is the initialization state.
- `node_controller` receives messages. It branch toward the state that handle the message that has just been received

- `node_root` send a message to the father of the node in order to return the root of the topoogy.
- `node_ping` sequentially send a message to the left child then to the right child in order to return a list of all the nodes of the topology (in postfix order)
- `node_job_next` is the function responsible to spawn a new process if a new job is received or if a job token is released.
- `node_job_load` is responsible to update upward the load value of the nodes. When one node finishes the execution of one job, the load value of its ancestors is updated
- `node_job_new` search for the best node on which execute a new job, taking into account the current load of each node of the topology.
- `node_split` handles the split operation.
- `node_merge` handles the merge operation.

The file `monitor.erl` implements several interface function to manipulate the treap topology.

- `ask_merge(RootTreap1, RootTreap2)` merges two treaps.
- `ask_split(RootTreap, Rank)` split the treap according to the threshold. It returns two treaps, the first one having a size `Rank` (if the threshold was valid).
- `ask_remove(RootTreap, Rank)` kills the process of rank `Rank` and maintains a coherent topology.
- `ask_root(NodeTreap)` returns the identifier of the root of the treap in which `NodeTreap` is.
- `ask_ping(RootTreap)` returns the list of all the identifiers of the nodes of the treap rooted in `RootTreap`.
- `ask_job(RootTreap, Mod, Fun, Args)` submit one job in the topology. The process that called this function must expect to receive the return value of the job. As nodes at the top of the treap may forward a lot of jobs, a node can accept a job only if the size of its subtree is lower than a given threshold.
- `ask_kill(RootTreap)` kills all the nodes of the treap.
- `ask_graph(RootTreap, Filename)` exports a graph representing the topology (graphviz dot format)
- `ask_say_hello(RootTreap)` ask each node of the treap to say hello!

3.2 Results

3.2.1 Complexity

The complexity of the requests `ask_ping`, `ask_graph` and `ask_kill` is linear in the size of the treap (the tree is traveled sequentially). The amortized complexity of all the other operations is logarithmic in the size of the treap.

One particular case is when we perform several merge request consecutively. When the merging process is finished on one node and continues with one of its children, a new merge request can be started. The complexity of performing N merge request on a treap of size M is therefore $\mathcal{O}(N+\log M)$. It is also true when submitting N jobs to a treap of size M.

The amortized complexity to add one node to the topology or to submit one job is therefore constant.

3.2.2 Graphs

Here is an example of a graph we get when running our topology on 14 nodes of the SLSU cluster (using `slsu.py` and `monitor:ask_graph`).

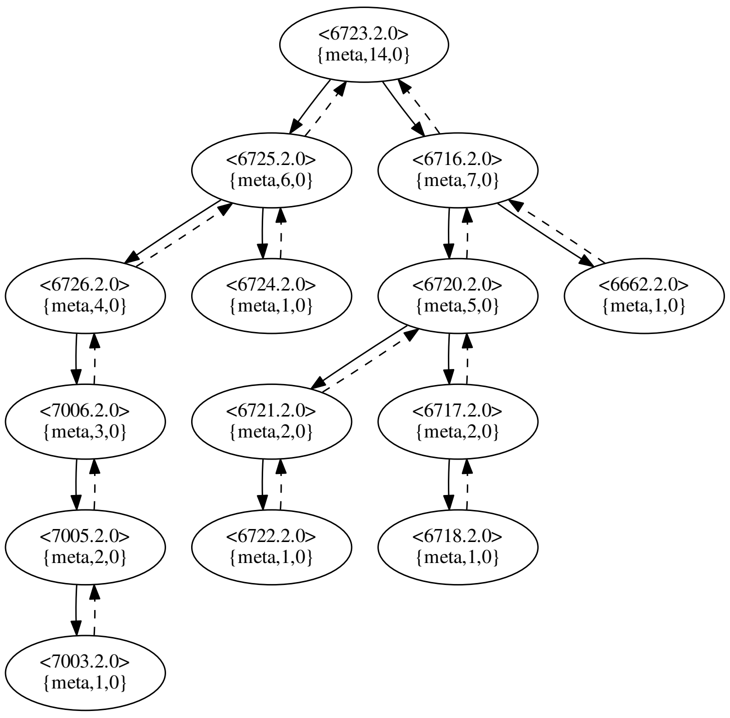

Figure 4: Example of Graph

3.3 Questions answered

Here is a quick summary of what we have implemented regarding to the questions asked in the project assignment.

Agent deployment

a) Spawn an agent that wait for a remote command: `spawn(agent, node_init, [1])`.

b) Have this agent part of a coherent topology: `monitor:ask_merge/2`.

c) Be able to have a new agent join an existing topoogy: `monitor:ask_merge/2`.

d) Have the possibility to remotely kill an agent: `monitor:ask_remove/2`.

Job execution

a) Be able to receive job request and send back the results: `agent:node_job_next`.

b) Keep a waiting list (see `treap.jobs`) so that no more than one job is performed simultaneously on this specific node (the argument of `agent:node_init/1` is the number of job tokens).

c) Have a voting that selects where to execute the job: `agent:node_job_new`.

Agent monitoring

a) Given an hostname, remotely spawn an agent on that node and have it join the topology: see installation instructions, either manually with the makefile or automatically using the python script.

b) Query the topology for statistics: `monitor:ask_ping/1` and `monitor:ask_graph/2`.

c) Submit job to one specific node: not implemented because of d), but easy with two splits, one job request and two merges.

d) Submit job to be executed on any node: `monitor:ask_job/4`.

e) Build a script that, given a set of nodes, automatically deploy the platform so that it is ready to accept new jobs: `slsu.py`.

Going further

a) Have multiple monitors: just run `make monitor` twice.

d) Merge and Split operations are useful if we want to use our middleware with several kinds of jobs (some time consuming jobs and a lot of fast jobs). Our voting process considers that jobs are all similar. If it is not the case, you can just split the network in two!

4 Installation

4.1 Using python script

Fill the hosts file (`bin/.hosts.erlang`) with the name of the servers you want to use as nodes. Make sure that you can access those nodes using ssh without password (e.g. public key authorized). Then run the python script `slsu.py`.

4.2 Create agents

On each machine, run `make agent`.

You can rename the node using for example `make NAME=node@192.168.0.1`.

If you want to run two independant instances of the middleware at the same time, you can change the cookie of the erlang nodes with `make COOKIE=seed`. To nodes will be able to connect to one other if and only if their cookies are identical.

4.3 Create hosts file

Update the hosts file (`bin/.hosts.erlang`) using to the following syntax.

```
'192.168.0.1'.
'192.168.0.2'.
```

One possible error is that the agents you deployed don't answer to request if their name is the default one or because of the DNS server. If it happens, please use IP adresses and rename the agents.

4.4 Launch the monitor

Finally you can merge the agents you just deployed in a coherent topology. The hosts file must be on the computer on which you we run `make monitor`.

You can now have fun with erlang's shell !

```
nodes().
Pid = global:whereis_name(agent).
monitor:ask_graph(Pid, "graph.dot").
monitor:ask_job(Pid, jobs, wait, [2000]).
receive ok -> ok end.
monitor:stop().
```