Distributed Spanning Tree Algorithms in Erlang

International Institute Of Information Technology, Hyderabad

Summer Project 2016

Shaleen Garg (201401069)

supervised by
Dr. Govindarajulu

Contents

0.1 Abstract . . . 2

1 Programming Erlang 3
1.1 About . . . 3
1.2 Installation . . . 3
1.3 Distributed Computing . . . 4
1.3.1 Testing . . . 4

2 Distributed Spanning Tree Algorithms 5
2.1 Definitions . . . 5
2.2 Synchronous Single-Initiator Breadth First Spanning Tree . . . 5
2.2.1 Assumptions . . . 5
2.2.2 The Algorithm . . . 6
2.2.3 Explanation . . . 6
2.2.4 On Terminal . . . 7
2.2.5 Complexity . . . 8
2.3 Asynchronous Concurrent-Initiator Depth First Spanning Tree . . . 8
2.3.1 Assumptions . . . 8
2.3.2 Design . . . 8
2.3.3 Explanation . . . 9
2.3.4 The Algorithm . . . 9
2.3.5 On Terminal . . . 9
2.3.6 Complexity . . . 11

0.1 Abstract

This aim of this project is to get familier with message passing programming languages which are optimised for distributed programming. The language of choice is Erlang. I started learning erlang and implemented some generic sequential programms like sorting and file transfer between a client and server.
Then I started implementing some of the well known distributed spanning tree algorithms in erlang.
In this Summer Project, I have implemented:

1. Synchronous Single-Initiator Breadth First Spanning Tree Algorithm
2. Asynchronous Concurrent- Initiator Depth First Spanning Tree Algorithm.

Keywords: Erlang, Distributed System, Spanning Tree, Graph Algorithms

Programming Erlang

This section will try and help the reader to get a headsup of Erlang. It will also help the reader to install and get a working erlang machine.

1.1 About

Erlang follows the following paradigms:

1. Concurrent: A form of computing where several computations are executed during the same time periods (Concurrently).
2. Functional: A style of computing which avoid change of state and mutable data.

Erlang follows the ideology “Let it crash”. Hence it is ideal to develop distributed, fault-tolerant systems.
It runs on BEAM virutal machine. It has its own schedular, garbage collector. This makes it highly compatible between different operating systems.
Spawning or destroying a process in erlang is as quick as allocating an object in a object oriented language.

1.2 Installation

Erlang is available in almost all flavours of Linux, Mac (Homebrew), Windows and FreeBSD.

- Linux
  Arch Linux:
  ```
  $ sudo pacman -s erlang
  ```
  Ubuntu and Debian:
  ```
  $ sudo apt-get install erlang
  ```
  Fedora:
  ```
  $ yum install erlang
  ```
- Mac
  ```
  $ brew install erlang
  ```

- FreeBSD
  ```
  $ pkg install erlang
  ```

For Distributed Computing, port 4369 has to be opened for TCP and UDP (Incoming and Outgoing traffic).

1.3 Distributed Computing

Most books on erlang don't elaborate on communication between two machines.
Erlang interactive shell can be started on the terminal by:

```
$ erl
```

This spawns a generic virtual machine with no name, hence it can not be used to contact with other virtual machines on the same computer or other VM's on other computers in the same network.
Naming can be done by:

```
$ erl -name M1@192.168.1.2 ##192.168.1.2 has to be replaced by the machines IP
```

Now we have a machine with name M1@192.168.1.2
But in order to have a succesful communication, we must have a common cookie(password) for each VM.
This can be done by:

```
$ erl -name M1@192.168.1.2 --setcookie abc ##here ‘‘abc’’is an example cookie
```

Now M1 can communicate with any machine on the network with cookie "abc"

1.3.1 Testing

We can test if two machines can communicate between each other by typing the following command in erlang shell

```
> net_adm:ping('M2@192.168.10.26').
```

If "pong" is the result, then the two machines succesfully communicate between each other.

Distributed Spanning Tree Algorithms

2.1 Definitions

- **Spanning Tree**: A subset of graph G, which has all the veritces covered with minimum possible number of edges.
  There can be more than one spanning tree in a graph. It can not have cycles.
- **Synchronous**: A kind of execution where, a process waits for the other process to send a message before proceeding further
- **Asynchronous**: A kind of execution where, a process does not wait for the other process to send a message before proceeding. It takes the message into consideration when the message arrives.
- **Single-Initiator**: In a pool of nodes (processes), only a predefined node starts the execution.
- **Concurrent-Initiator**: All the nodes in pool start the execution but partial results form the inferior nodes is discarded.
- **Distributed Algorithm**: An algorithm designed to run between interconnected processes. The processes communicate among themselves using messages.

Distributed graph algorithm makes each vertex in the given graph as a different process (node) and edges act as communication lines between the nodes. The nodes exchange information using predefined messages. There exists a master node which is connected to each node in the graph and its job is to spawn all the nodes corresponding to the graph given as input and collect results from each of the nodes.

2.2 Synchronous Single-Initiator Breadth First Spanning Tree

2.2.1 Assumptions

1. Each node in a graph has a pratial view of the graph, ie. it can only see and communicate with nodes directly connected to it.
2. Edges in the graph are unweighted and bidirectional.
3. The graph is connected.

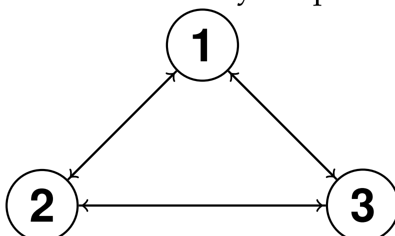
This algorithm has a symmetrical code structure, ie. each node in the pool is going to execute the same piece of code. THe first node in the graph provided is assumed to be the root node, hence the single-initiator. The root node initiates QUERY message in the graph.
A node sends a QUERY request to it's neighbours as soon as it receives a QUERY message. The first one to send a QUERY to a node becomes the parent of that node.
Each node sends IN_TREE message to the root node with its parent node once it receives a QUERY message to confirm its inclusion to the spanning tree.

2.2.2 The Algorithm

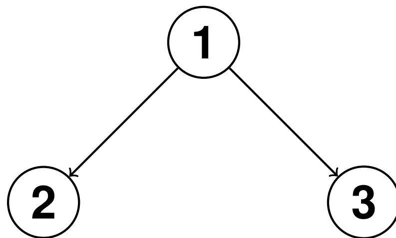
1: **procedure** BFS
2: $(Local_Variables)$
3: **int** $Visited, Depth \leftarrow 0$
4: **int** $Parent \leftarrow \perp$
5: **int** $Self \leftarrow PID$
6: **Tuple of PID** $Neighbours = Set_Of_Neighbours$
7: $(Message_Types) \rightarrow \{QUERY, Self\}, \{IN_TREE, Self, Parent\}$
8: **if** $i = Root$ **then**
9: $Visited \leftarrow 1$
10: $Depth \leftarrow 1$
11: **end if**
12: **while** $n(In_Tree) \neq Total_Number_Of_Nodes$ **do**
13: **if** $Visited = 0$ **then**
14: **if** $QUERY\ arrived$ **then**
15: $Parent \leftarrow First_To_Send_QUERY$
16: $Visited \leftarrow 1$
17: **send** $\{IN_TREE, Node, Parent\}\ to\ Root$
18: **send** $QUERY\ \ to\ Neighbours - Parent$
19: **end if**
20: **end if**
21: **end while**
22: **if** $Root\ \&\ n(IN_TREE) = Total_Nodes$ **then**
23: **print** $all\ \{Parent, Node\}$
24: **end if**
25: **end procedure**

2.2.3 Explanation

Let us take a very simple example.



Here, node 1 is assumed as the root node. It will start by sending QUERY message to nodes 2 and 3. Node 2 and 3 will assume node 1 as their parent. Node 2 and 3 will send IN_TREE to root node, announcing its inclusion in the spanning tree.
Now both node 2 and 3 will send QUERY to its other neighbours who are not their parents. Node 2 will send QUERY to 3 and node 3 will send it to node 2. Both the QUERY requests will be rejected because both the nodes have decided their parent.
Root node has received node 2 and 3's IN_TREE message. Hence Now it can print out its tree like so.



It is the breadth first spanning tree of the graph.

2.2.4 On Terminal

The code expects 2 files:

1. List of Machines.
   This file contains name of all the files available on our disposal.
   Example

   'M1@192.168.1.2'.
   'M2@192.168.1.3'.

2. Graph.
   This file contains the number of nodes in the graph and all the edges in the graph.
   Example

```
3.
[2, 3]. %%This represents that there exists an edge between node 2 and node 3
[1, 3].
[1, 2].
```

The algorithm can be invoked by:

```
> bfs:execute(''graph.txt'', ''computer.txt'').
```

Result

The programme returns a list of tuples of the form {Parent, node}

{[{1,2},{1,3},{0,1}]}

Note that for node 1, we consider an imaginary parent, node 0.

2.2.5 Complexity

Complexity of a typical message passing distributed algorithm is measured in the order of the messages sent across by each node.
In this algorithm, each node sends a linear amount of messages per edge.
Hence complexity is O(l) messages where l is the number of edges.

2.3 Asynchronous Concurrent-Initiator Depth First Spanning Tree

This algorithm involves a lot of non-determinism because, node with highest PID is selected as the root node. PIDs are selected at random while spawning the node.

2.3.1 Assumptions

1. Each node in the graph has complete view of the graph but can send messages to its immediate neighbours only.
2. Edges in the graph are unweighted and bidirectional.
3. The graph is connected.
4. PID is a unique identifier of an individual node.

Each node in the graph can spontaneously initiate a spanning tree with it as the root node provided that it has not been invoked locally.

2.3.2 Design

Newroot is the root node of the node sending the message. Myroot is the root of the node receiving the message.

When **QUERY(newroot)** form j arrives to i, there are three possibilities.

- **newroot** > **myroot**: process i should suppress its current execution due to its lower priority. It re initializes the data structures and joins js subtree with newroot as the root.
- **newroot = myroot**: js execution is initiated by the same root as is initiation, and i has already identified its parent. Hence REJECT is sent to j.
- **newroot** < **myroot**: js root has a lower priority and hence i does not join js subtreei sends a REJECT. j will eventually receive a QUERY(myroot) from i; and abandon its current execution in favour of is myroot (or a larger value).

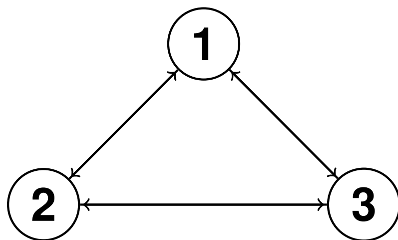
When **ACCEPT(newroot)** from j arrives to i, there are three possibilities.

- **newroot = myroot**: The ACCEPT is in response to a QUERY sent by i. The ACCEPT is processed normally.
- **newroot** < **myroot**: The ACCEPT is in response to a QUERY i had sent to j earlier, i has updated its myroot to a higher value. So this case is ignored.
- **newroot** > **myroot**: The ACCEPT is in response to a QUERY i has sent earlier. But i never updates myroot to a lower value. So this case cannot arise.

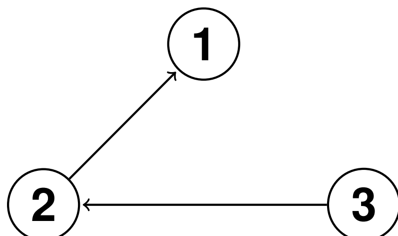
The three possibilities when **REJECT(newroot)** from j arrives at i are the same as for the ACCEPT message.

2.3.3 Explanation

Let us take a simple example:



After running the algorithm, We get the result.



2.3.4 The Algorithm

2.3.5 On Terminal

The code expects 2 files:

1. List of Machines.
   This file contains name of all the files available on our disposal.
   Example

   'M1@192.168.1.2'.
   'M2@192.168.1.3'.

2. Graph.
   This file contains the number of nodes in the graph and all the edges in the graph.
   Example

```
1:  procedure DFS
2:      (Local_Variables)
3:      int Parent, myroot ← ⊥
4:      int Self ← PID
5:      Tuple of PID Neighbours, Unknown = Set_Of_Neighbours
6:      Tuple of PID Children = ∅
7:      (Message_Types) → {QUERY, Self}, ACCEPT, REJECT
8:      if Parent = ⊥ then
9:          sendQUERY(i) to_itself
10:     end if
11:     When QUERY(newroot)arrives_from_j
12:     if myroot < newroot then
13:         parent ← j; myroot ← newroot
14:         Unknown ← Unknown − {j};
15:         if Unknown ≠ ∅ then
16:             Delete some_x_from Unknown
17:             Send QUERY(myroot) to_x
18:         else Send ACCEPT(myroot) to_j
19:         end if
20:     elseif myroot = newroot
21:         Send REJECT to_j
22:     end if
23:     When ACCEPT(newroot) REJECT(newroot) arrives_from_j
24:     if newroot = myroot then
25:         if ACCEPT arrives then
26:             Children ← Children ∪ j
27:         end if
28:         if Unknown = φ then
29:             if parent ≠ i then
30:                 SendACCEPT(myroot) to_Parent
31:             else set i as_the_root; terminate
32:             end if
33:         else
34:             Delete some_x_from Unknown
35:             Send QUERY(myroot) to_x
36:         end if
37:     end if
38: end procedure
```

```
3.
[2, 3]. %%This represents that there exists an edge between node 2 and node 3
[1, 3].
[1, 2].
```

The algorithm can be invoked by:

```
> dfs:execute(‘‘computer.txt’’, ‘‘graph.txt’’)
```

Result

The programme will return children of each node
Each tuple of the form Node, Pid
Example:
3 is the root
{2, < 0.74.0 >} %% Node 2 is the child of Node 3
Children of Node 1
-NIL-
Children of Node 2
{1, < 0.73.0 >}

2.3.6 Complexity

Time complexity of the algorithm is O(l) messages, and the number of messages is O(nl).

References

1. Distributed Computing by Ajay Kshemkalyani and Mukesh Singhal
2. Programming Erlang by Joe Armstrong
3. Distributed Algorithms by Nancy A. Lynch
4. https://en.wikipedia.org/wiki/Erlang_(programming_language)
5. http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/index.htm