CS5320: Distributed Computing, Spring 2020

Programming Assignment 1: Termination Detection

SAI HARSHA KOTTAPALLI

CS17BTECH11036

GOAL

Simulation of model using :

- A spanning-tree-based termination detection algorithm in A&M section 7.5
- The Message-optimal termination detection explained in A&M section 7.6

DESIGN

Let there be n cells(which are threads). Their channels are in such a way that the associated graph is a fully connected graph. The communication via channels can be done through sockets.

Initially the state of each cell will be white(0). It becomes Red(1) when infected and Blue(2) when cured. Here, Red Cell is analogous to the server being active and Blue Cell is analogous to the server being idle.

The Model introduces these blue Cells and Red cells after some period based on input.

Based on an exponential distribution, periodically, the cells spread the infection/cure to its sample set of neighbours based on the input. The Red Cell sending messages is analogous to a basic message.

The spanning tree lets a server know “who” it’s parent is so that it can send appropriate token/terminate messages to it’s parent.

The root server initiates the termination detection algorithm. Each server sends msg to parent regarding termination when particular sets of conditions are fulfilled:

- A&M 7.5
    - Send token to parent with its token colour when server and its children are idle
- A&M 7.6
    - Send a token to the parent when the server is idle, each of its links are coloured, stack is empty and all its children are inactive.

The root server detects the ending when:

- A&M 7.5
    - Its token colour is white and it with its children is idle.
    - Else, it repeats the algo. (resets children server token)
- A&M 7.6
    - The server is idle, each of its links are coloured, stack is empty and all its children are inactive.

Messages:

- Red Message
- Blue Message
- white token **[A&M 7.5]**
- black token **[A&M 7.5]**
- Warning Message **[A&M 7.6]**

- Terminate Message **[A&M 7.6]**
- Remove Entry Message **[A&M 7.6]**

The message passing has been done with the help of sockets and an appropriate channel delay was introduced to simulate a real time channel.

The more intrinsic details have also been implemented using the text in A&M 7.5 and 7.6 which need not be explained here.

Given, Input: number of servers, and graph topology.

Output is the simulation of various events in the model.

The timestamp used in the log file is of the format: sec : nano sec.

where it begins from the start of the program.

IMPLEMENTATION

Class **TS:**

This class is used to simulate each Cell(server). Hence, each server will store the following:

- variables
    - Total -> Total number of servers
    - state -> 0 = white, 1 = red and 2= blue
    - Id -> thread identifier (starts with index 2)
    - parent -> parent based on spanning tree
    - t_children -> total number of children based on spanning tree

    - t_token **[A&M 7.5] ->** total number of token received
    - t_term **[A&M 7.6] ->** total number of termination messages received
    - Exponential_distribution
        - For simulating work done after sending blue msg, red msg and channel delay.
    - token_colour **[A&M 7.5] ->** 0 = white, 1 = black
    - dt **[A&M 7.6] ->** 0 = ndt and 1 = dt
    - sent -> to check if token/term has already been sent.
    - root -> Boolean to check if server is root
    - p, q, lam_{parameters} -> as described in the assignment
    - vector list -> list of of neighbours
    - vector span_child -> list of of children based on spanning tree
    - vector channel_colour **[A&M 7.6]** -> for channel colouring
    - vector queue **[A&M 7.6]** -> for the implementation of stack
- Functions
    - Constructor and assign_attr -> Here I bind the socket for listening to connections for receiving blue/red msg or the tokens/termination msg.
    - set_state -> used when a set of random nodes are turned red/blue.
    - Send_channel -> simulates message being sent over the channel
    - spread_others -> Every period of time, based on a servers state (red/blue) infect/cure correspondingly a fixed percentage of its neighbours.
        - **[A&M 7.6]** Root enters dt and starts sending warning messages to its neighbours.
    - parse_data -> to decrypt the data sent by a channel into message and channel_number
    - Handle_replies -> listens for requests and serves replies accordingly using setInfection.
    - setInfection -> Receives the incoming msg. Gets infected if it is a red msg or gets cured if it is a blue msg. If it is a black or white token,

correspondingly takes action. encodes data to be sent and sends it via send_channel to simulate message via the channel.

        - **[A&M 7.5]** If reset msg is received that means the t.d. algo is going run for another round so reset corresponding variables. Finally call send_tokens
        - **[A&M 7.6]** Handles execution based on the incoming link colour (sends warning to rest accordingly). Calls when stack_cleanup or processes remove_entry as needed. Finally calls send_termiante
    - stack_cleanup **[A&M 7.6]** -> Removes entries and sends remove_entry msgs from the stack until the first entry of the form “TO”
    - send_tokens **[A&M 7.5]** -> Checks appropriate conditions as described in A&M to determine if it can send a token to its parent or end execution if root.
    - send_terminate **[A&M 7.6]** -> Checks appropriate conditions as described in A&M to determine if it can send a terminated msg to its parent or end execution if root,

Global Variables:

Since the concept of shared memory does not exist in distributed systems, I made sure to not use any info from global variables for the simulation of termination detection. They are strictly used for capturing the global state.

I have used only the following vars with explanation for doing so:

- log_file
    - Represents the global state in our simulation where all the events info is recorded.
- mutex
    - Write_lock

        - Since I have to log the timestamps of specified events, I used mutex so that race conditions don't occur while storing them in the log file. Since the log file represents the global state, it is fine to use a global mutex.
    - End_lock
        - used to make a checkpoint in all servers until all of them ready to handle replies..
- default_random_engine
    - This is used for the generation of value from an exponential distribution for sleep to simulate some work done.
- local_clock_mut
    - An array of mutex for doing atomic execution of code when required. Each server uses its own index's mutex and in doing so, there is nothing shared here.
- start_time
    - Store start time so that the timestamp used for recording events is relative to the start of the program.

Other Functions:

- currTime
    - Returns Timestamp, used for logging purpose
- get_input
    - parse input and generate servers accordingly
- server_simuate
    - simulation of servers using threads

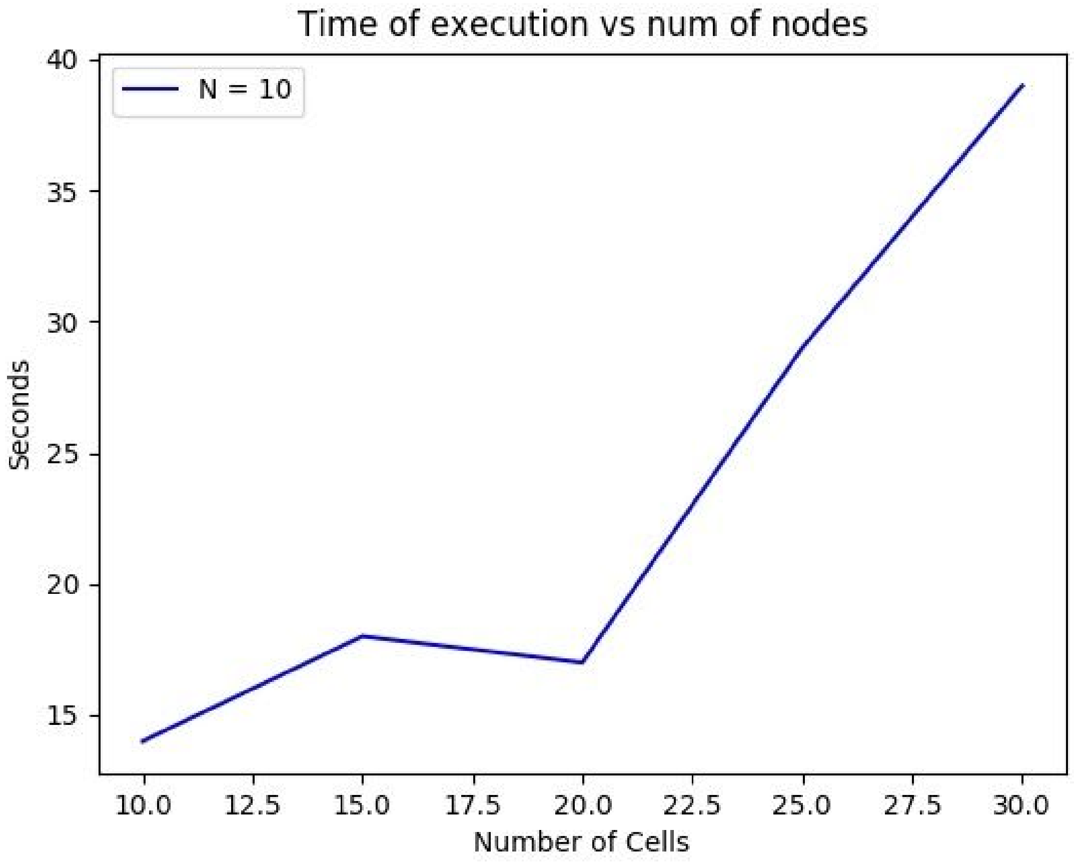
EXECUTION

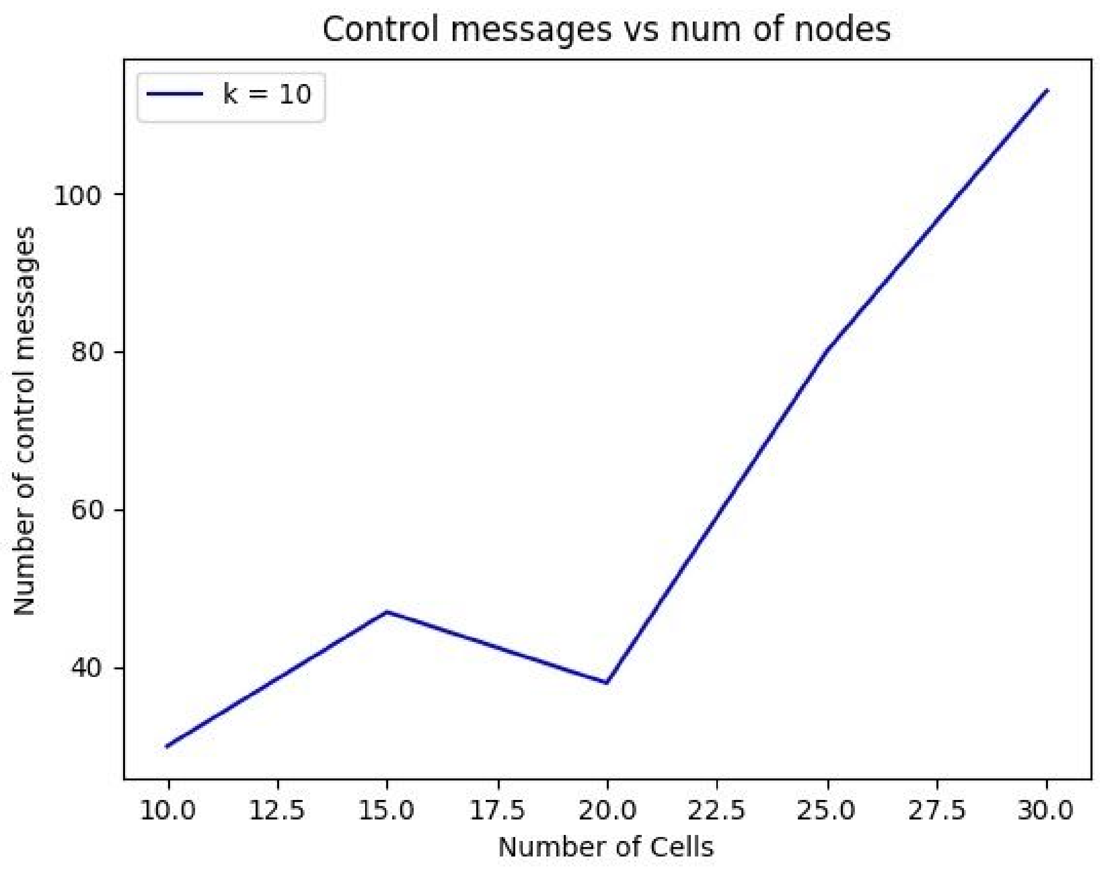
- Parse input from inp-params.txt and initialize the required number of TS(Cells).
- In initialization, construct required exponential distributions and bind the server socket accordingly. (Set 127.0.0.x as server IP address) where x ∈ [2, number of time servers + 1]
- Open the log file and record events when necessary throughout the program.
- Store the start_time, so that calculation of timestamps is done relative to this.
- Spawn threads equal to the number of time servers, where it, in turn, spawns 2 threads which are th_send and th_recv.
    - Th_send
        - It calls spread_others function.
        - waits for all listeners of all servers to be ready..
        - **[A&M 7.6]** Root starts the termination algorithm and sends warning messages to its neighbours
        - Every period of time, based on a servers state (red/blue) infect/cure correspondingly a fixed percentage of its neighbours.
        - If the channel is coloured, then store any red msg into its stack in the form of "TO".
    - Th_recv
        - It calls handle_replies function.
        - This function inturn spawns a thread for setInfection and continues listening so that multiple requests can be served concurrently and there is no deadlock.
        - Its use case has already been explained above in implementation.
        - **[A&M 7.5]**
            - Every red or blue message sets the state of the server.

        - A white token increases token count while a black token increases token count as well as sets its colour.
        - On reset token msg, server has to reset token count and its colour while also forwarding this msg to all its children in the spanning tree.
        - send_tokens is then called whose usage is also explained.
    - **[A&M 7.6]**
        - If dt is not set, that means it didn't get any warning msg yet. The first warning msg triggers the server to send warning msg to all its neighbors. else, colour appropriate channel when needed on warning msg.
        - If the server is in dt and the incoming channel is in dt, then on a red msg we have to store into the stack in the form of "FROM"..
        - If it is a blue msg, then it triggers stack_cleanup as the server essentially becomes idle.
        - Remove_entry calls do the necessary by removing required entry and calling stack_cleanup if it is idle.
        - send_terminate is then called whose usage is also explained.

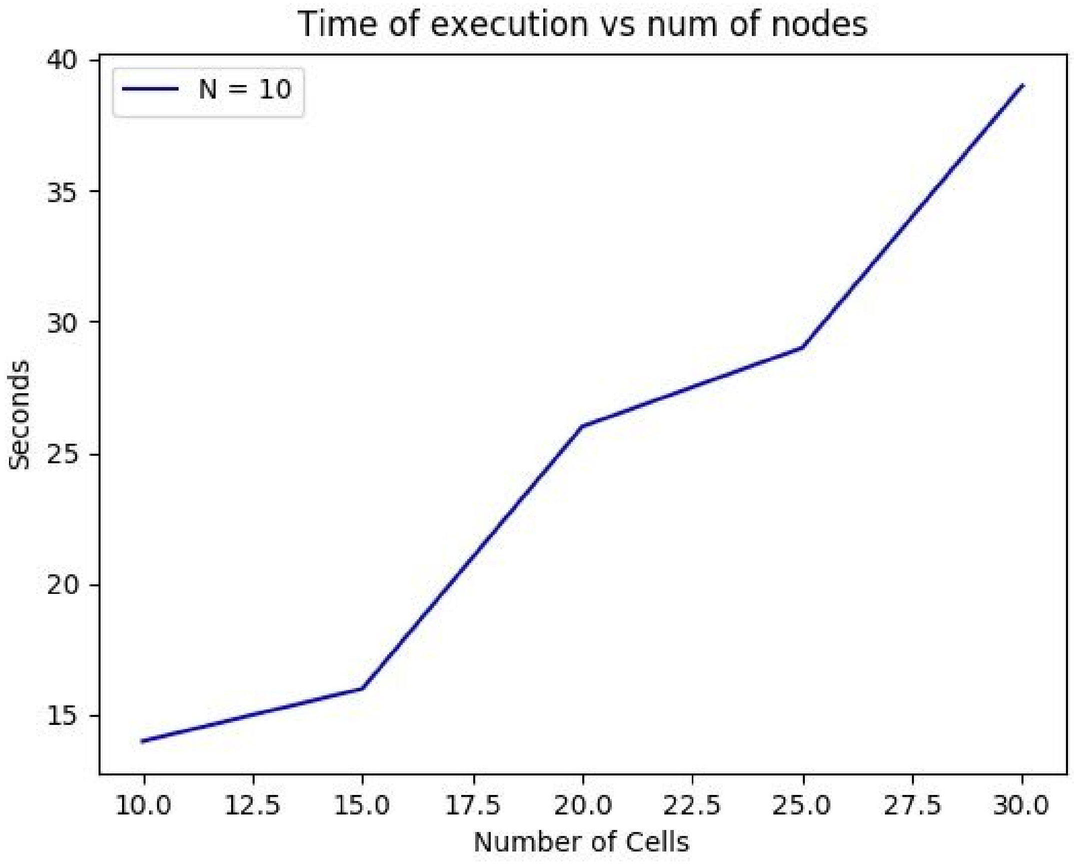
GRAPHS

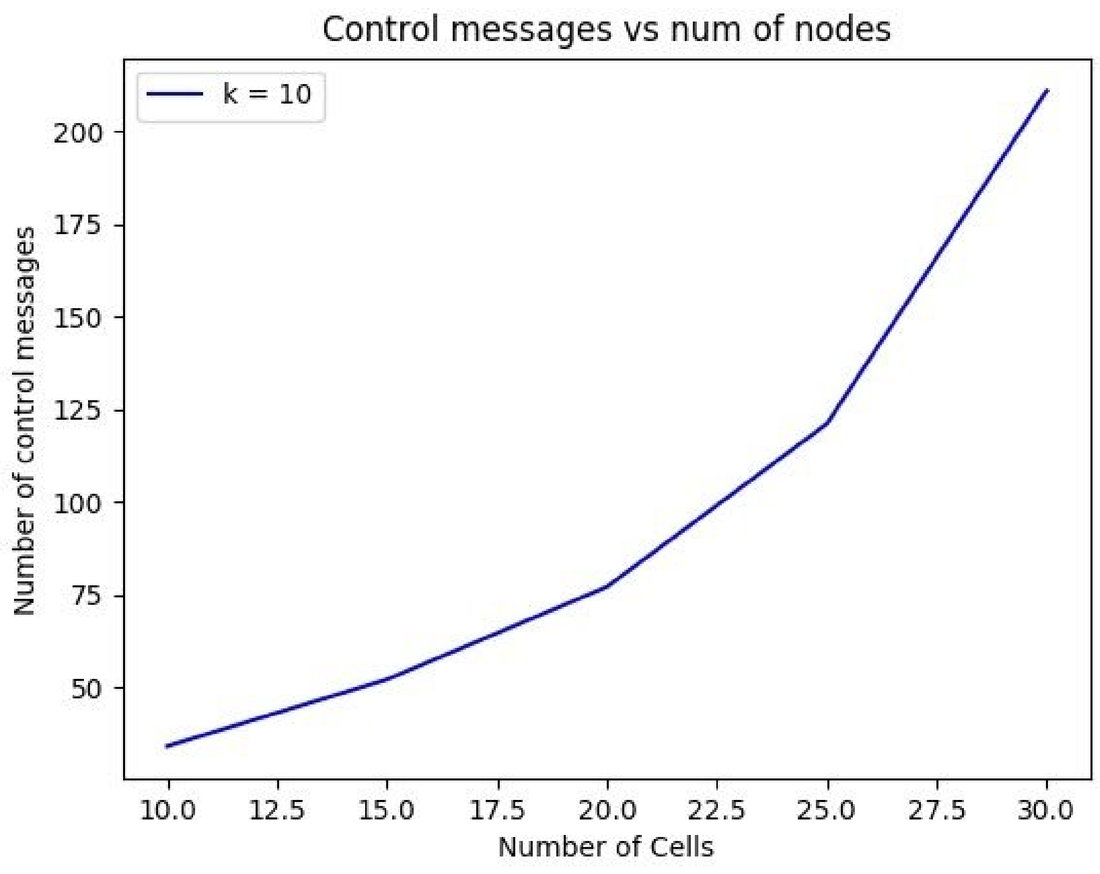
A&M 7.5


Time of execution vs num of nodes
N = 10
Seconds
Number of Cells
40
35
30
25
20
15
10.0
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0


Control messages vs num of nodes
k = 10
Number of control messages
Number of Cells
100
80
60
40
10.0
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0

A&M 7.6


Time of execution vs num of nodes
N = 10
Seconds
Number of Cells
40
35
30
25
20
15
10.0
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0



OBSERVATIONS

- As the number of cells increases, *generally* control messages increase(except when edge cases occur as explained later). For second algo though the number of warning messages exchanged also increased by quite a bit.
- Time taken increases as expected as the number of cells increases *for similar flow of execution.*

- For the first algo, the number of messages exchanged can't be bounded unlike the second algo and its data is too skewed. just one black token results in restart of the entire algo. So the result can’t exactly be inferred from. Generally, it shows monotonic behavior for best case scenarios. For large graphs, the messages will be too high.
- For second algo, messages generated by the algorithm is 2* E + V −1 + M. Since, the graph is fully connected, we can say number of control messages >= 3* V. We notice the number during multiple runs is not too skewed. For very large graphs we can notice that this algo performs much better than previous one.
- If during execution all cells become red then the program does not halt(as expected from problem description).
- Each run is very different as the parameters given are used to generate randomness. So, the same input might lead to program running for a very long time or for a small duration. Hence, the graphs don't exactly show/prove why one t.d. algorithm might be better than the other. Even x axis changing doesn't give a valid curve as for each run, the graph topology is significantly changed.
- Hence, multiple runs are used for generating the above graphs.
- Sometimes, the execution is very quick. This is because a red cell might turn blue before it is able to infect others and hence, red cells would be eradicated completely before even any others are cured.