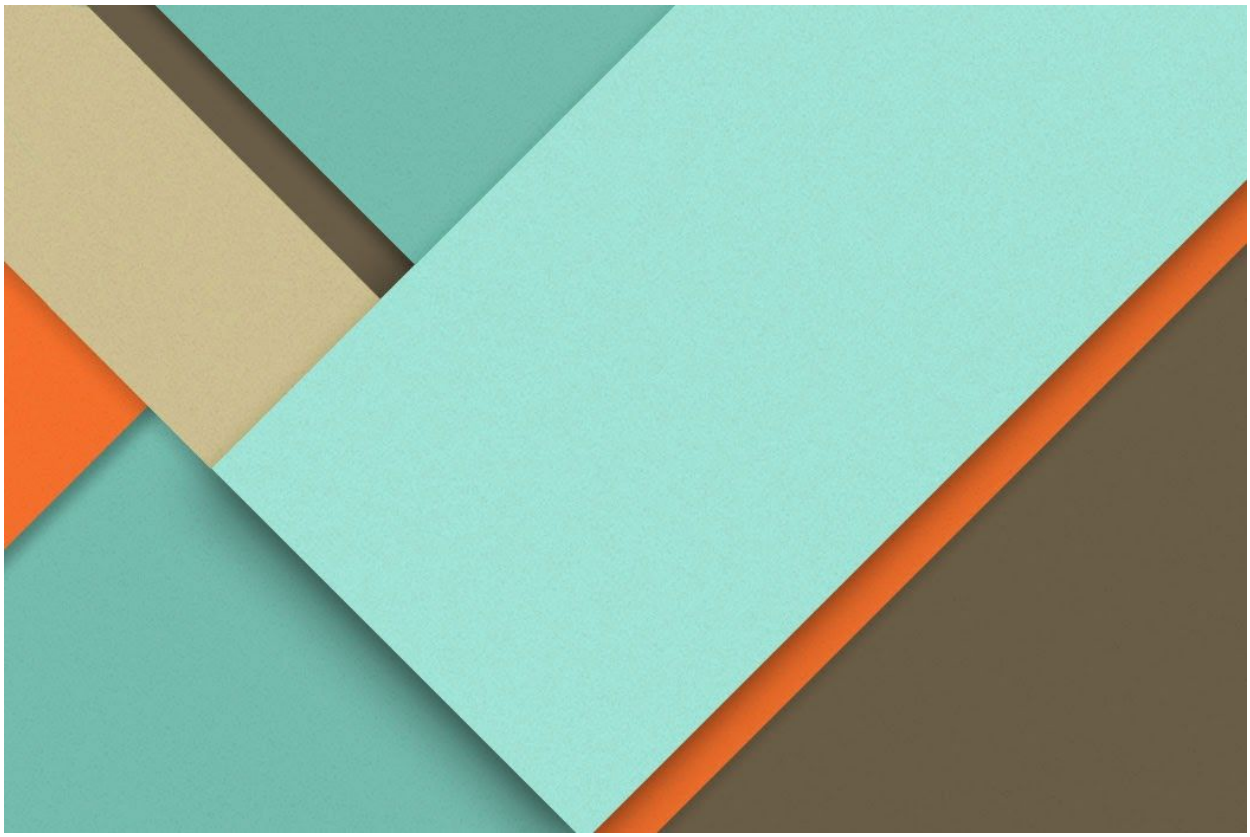


Termination Detection for Rings

02.5.2020

Saahil Sirowa (CS16BTECH11030)

Abhishek Pawar(CS16BTECH11027)

Overview

As a part of this project, we are going to implement 2 termination detection algorithms.

1) Dijkstra-Safra Algorithm
2) Dijkstra-Safra Algorithm with enhancements

Introduction

A problem of termination detection involves detecting a state when all processes are passive and no message is in transit. For rings, it can be solved by using the Dijkstra-Safra algorithm. But the enhancement proposed will aim to reduce the message complexity constant from (2X - 3X) to (0 - X). That is we aim at early stopping.

Goals

- To compare the performances of the above mentioned algorithms
- To analyse the complexity of messages and time involved

Implementation

The module contains two programs named as term_detect.cpp and enhanced_detect.cpp

1. **dijkstra_safra_simple.cpp**
    - This contains a Node class which will have the methods for sending/ receiving messages.
    - The token-passing algorithm runs in a separate thread.
2. **dijkstra_safra_enumBit.cpp**
    - It is the same as the above with enumBit enhancements to the algorithm and addition of new fields to the Node class.
3. **dijkstra_safra_enhanced.cpp**
    - It is the same as the above with enumBit and multiple initiator enhancements to the algorithm and addition of new fields to the Node class.

Algorithm Overview

Dijkstra-Safra (Simple Version)

This is a token passing algorithm with an initiator (process N) sending a token around the ring. Each process has a counter = send - recv. It holds the value of messages in transit. Also , it is associated with a color which can be either black or white. The token also has a color and a counter.

Once the token completes a round around the ring, we check for the validity of termination detection. The following rules are observed as the token passes:

- Token is passed to the next process only when the current process becomes passive
- The token counter is incremented by the passive process counter and it gets the color of the process. The process then becomes white
- If a process receives a message (not a token) then it becomes black.
- Once the token comes back to process N, we check for the termination
- If the token is at N and process N is idle, we check if token.counter + process[N],counter is 0 and the token color is white, then we conclude otherwise we restart.

In the implementation, we have simulated the active-passive states using red-blue-white colors as discussed in the Programming Assignment-2. Since red means active, we have accordingly modified the algorithm to consider only the messages with a process-id of a red process.

Message Complexity : O(N*M)

Dijkstra-Safra (with EnumBit Addition)

A process maintains an enumBit to keep track of whether the token has passed through it. With each round of token reaching a process, it's enumBit is flipped. The enumBit is added to the message packet. If the enumBit of process and enumBit received in message is different and the sender id of message is less than the receiver process, we then color it black. If message's enumBit and process's enumBit are the same it means that either both are not visited or both are visited. Hence the exchange of messages between them has been recorded or yet to record. In this case, we will not miss out on any event and hence there is no problem of false detection.

Dijkstra-Safra (Enhanced Version)

The simple version suffers from redundancy and needs more rounds to terminate. Early detection is possible if we avoid unnecessary coloring and use more than one initiators. We combine these enhancements to improve the original algorithm. Rest of the algorithm remains the same more or less.

The following rules associated with enhancements are:

- We keep a track of token visiting processes by enumBit. After each round, we can see that the bits flip from 0 to 1 or vice versa.
- The token will reset the enumBit of the process to mark it as visited
- This can be useful in reducing useless coloring.
- A recv process will change the color to black iff sender.enumBit != recv.enumBit and sender.id < recv.id
- Instead of a single initiator, we can think in terms of many initiators and maintain an array of counter sums. The array is then changed at certain indices in case of transmission/retransmission
- We combine above two improvements in original algo.

Message Complexity : O(N*M) (However in most of the cases, will terminate faster)

Code Entities

Node: This is a class of Process. Each Node maintains the following entities:

- **state:** blue or white or red (indicates passive/active)
- **counter:** sent -recv
- **color:** white or black as per the algo
- **idle:** true if passive
- **flag:** a flag for correct termination of recv socket processes
- **tokenHas**: true if process has the token
- **sendMsg:** to send messages
- **recvMsg:** to recv messages and update accordingly

Msg: This structure has the following entities**:**

- **msgType:** whether token or normal
- **sender_id:** sender process id
- **sender_state:** state of the sender
- **tokenVal:** to pass counterSum and color of token to next process

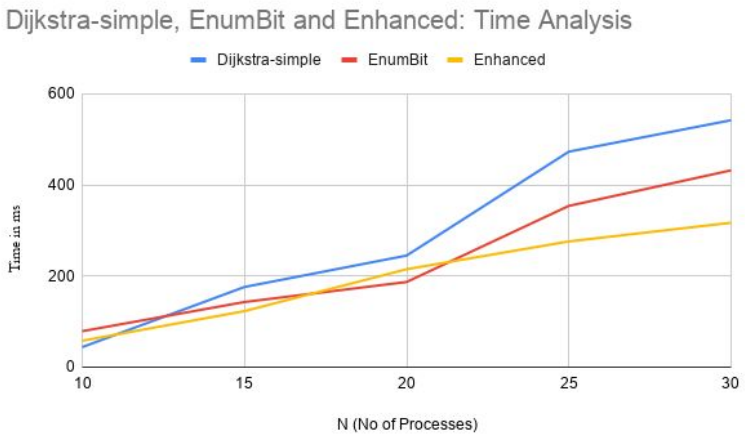
Token: This structure has two entities:

- **counterSum:** keep track of the sum of counters
- **Color:** color of the token

Results-

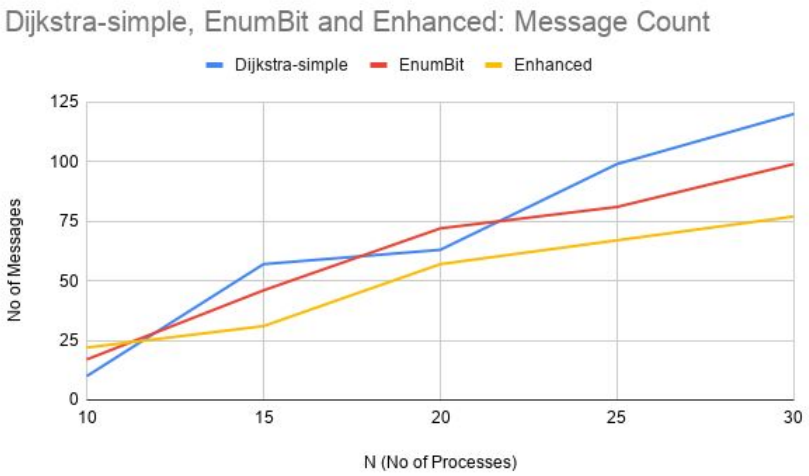
Here we present the comparison graphs:

Time Comparison



Some fluctuations are possible in the initial stages. This can be attributed to the fact that for small no of processes, the simple algorithm will have less check overload and termination detection will be fast. It can be seen that for higher values, the expected results are visible.

No of Messages comparison



Here also the same thing happens. For small values of N, a simple algorithm can terminate fast as stated above. This in turn ensures less rounds and thus less control messages (or token transfers from one process to the next). Again for higher N, we see the expected results.

THANK YOU