Distributed Computing

Comparing Token based Mutual Exclusion Algorithms

Shagun

cs23mtech14013

29.04.2024

INTRODUCTION

Introduction:

Mutual exclusion, a fundamental concept in distributed systems, ensures that concurrent processes or threads do not simultaneously access a shared resource, thereby preventing conflicts and maintaining system integrity. In distributed systems, where multiple autonomous nodes communicate and coordinate over a network, achieving mutual exclusion becomes even more challenging due to factors such as communication delays, network partitions, and node failures. Consequently, distributed mutual exclusion algorithms play a critical role in ensuring the correctness and consistency of distributed applications.

Token-based mutual exclusion algorithms are a class of distributed mutual exclusion algorithms that rely on the concept of tokens to coordinate access to shared resources among distributed nodes. In these algorithms, a token acts as a permission or a key, granting the holder the exclusive right to access the shared resource. Nodes pass the token among themselves in a predefined order, ensuring that only the node possessing the token can access the resource at any given time. This approach minimizes contention and ensures that access to the shared resource is serialized, maintaining consistency across the distributed system. Asynchronous and synchronous token-based algorithms are among the variants designed to address different synchronization requirements and network conditions in distributed environments. In this report, we will compare and analyze two token-based mutual exclusion algorithms, examining their strengths, weaknesses, and performance characteristics in various distributed system scenarios.

PROGRAM DESIGN

A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal

Presuppositions:

1) The connectivity between the nodes is guaranteed to be flawless (no loss, duplication, or alteration of messages).

2) Each node in a distributed environment possesses its own identifier as well as the identifiers of its neighboring nodes.

3) The links between any pair of nodes operate in one direction only.

4) The nodes are interconnected in a manner such that each node is directed towards another node until reaching a node devoid of outgoing connections, which initially holds the token.

5) No specific assumptions are made regarding the layout of the processes, apart from their connectivity.

6) There is no requirement for message delivery to adhere to FIFO (First-In-First-Out) order.

Message Complexity
The average number of messages is of O(log(number of processes)) complexity where n refers to the number of nodes in the system.

Design
1. Message Structure: The `Message` struct defines the structure of messages exchanged between nodes. It includes fields such as `type`, `senderID`, and `reqProcess`, representing the

type of message, the sender's ID, and additional data for certain message types.

2. Socket Communication: Nodes communicate with each other using TCP sockets. Each node creates a socket to listen for incoming messages (`createReceiveSocket`) and sends messages to other nodes (`sendMessage`).

3. Node Functions:
   - `do_work`: Simulates the behavior of a node executing critical section code. It generates random intervals for entering and exiting the critical section, requests access to the critical section (`wantToEnter`), performs local computation, and releases access (`wantToLeave`).
   - `do_receive`: Listens for incoming messages on the designated port. It handles various message types, such as privilege requests, requests to enter critical section, and termination messages.
   - `do_run`: Initializes the simulation by creating threads for sending and receiving messages for each node. It also initializes necessary data structures and variables for each node.

4. Concurrency Control:
   - Mutexes (`pthread_mutex_t`) are used to synchronize access to shared resources, such as message queues and counters, to prevent race conditions.
   - Atomic variables (`std::atomic`) are used for counters (`totalReceivedMessages`, `totalResponseTime`) to ensure safe concurrent access.

5. Simulation Parameters:
   - The main function reads simulation parameters from an input file (`inp-params.txt`), such as the number of nodes, mutual exclusion count, and parameters for exponential distributions used for simulating critical section durations.
   - It then initiates the simulation using these parameters.

6. Termination Condition: The simulation continues until all nodes have completed their tasks and sent termination messages. The main thread waits until all nodes have terminated before exiting.

7. Output and Analysis: The code outputs various messages and statistics to an output file (`output.txt`) for analysis, including messages exchanged, average response time, and completion status of nodes.
Overall, the code implements a distributed mutual exclusion algorithm using token-based message passing, simulating the behavior of multiple nodes executing critical section code concurrently while ensuring mutual exclusion.

Pseudocode:

Define messageType enumeration:
  - PRIVILEGE
  - REQUEST
  - TERMINATE

Define Message structure:
  - messageType type
  - int senderID
  - int reqProcess

Define MAX_NODES as 30
Define FALSE as 0
Define TRUE as 1
Define NIL_PROCESS as -1

Define global variables:
  - int startPort
  - atomic<int> totalReceivedMessages = 0
  - atomic<long long int> totalResponseTime = 0

Define createReceiveSocket function:
  - Input: int nodeID, int sockPORT
  - Output: int sockfd
  - Actions:
    - Create a socket with given parameters
    - Bind the socket
    - Listen to the socket
    - Set timeout for the socket
    - Return the socket file descriptor

Define sendMessage function:
  - Input: int nodeID, int destNodeID, messageType type, int needsReqProcess
  - Output: bool indicating success or failure
  - Actions:
    - Create a message with given parameters
    - Create a socket and connect to destination node
    - Send the message through the socket
    - Close the socket
    - Return true if successful, false otherwise

Define wantToEnter function:
  - Input: int nodeID, int &does_request_cs, int &next_process, int &node_father, int &is_token_present, FILE *outfile, pthread_mutex_t *m_sendrec, Time &start
  - Actions:

- Lock the mutex
- Set does_request_cs to TRUE
- If node_father is not NIL_PROCESS:
  - Send a REQUEST message to node_father
  - Reset node_father to NIL_PROCESS
- Wait until the token is present
- Unlock the mutex

Define wantToLeave function:
- Input: int nodeID, int &does_request_cs, int &next_process, int &node_father, int &is_token_present, FILE *outfile, pthread_mutex_t *m_sendrec, Time &start
- Actions:
  - Lock the mutex
  - Set does_request_cs to FALSE
  - If next_process is not NIL_PROCESS:
    - Send a PRIVILEGE message to next_process
    - Reset next_process and is_token_present
  - Unlock the mutex

Define do_work function:
- Input: int nodeID, int &node_father, int &next_process, int &does_request_cs, int &is_token_present, int noOfNodes, int cnt_of_mutex, atomic<int> &Cnt_EndedProcesses, double lambda1, double lambda2, Time &start, FILE *outfile, pthread_mutex_t *m_sendrec
- Actions:
  - Initialize random generators
  - Perform critical section simulations
  - Update counters and send terminate messages

Define do_receive function:
- Input: int nodeID, int sockPORT, int noOfNodes, int &node_father, int &next_process, int &does_request_cs, int &is_token_present, atomic<int> &Cnt_EndedProcesses, Time &start, FILE *outfile, pthread_mutex_t *m_sendrec
- Actions:
  - Create a socket and start listening
  - Receive messages and handle them accordingly

Define do_run function:
- Input: int noOfNodes, int mutualExclusionCnt, int FirstTokenNode, double alpha, double beta
- Actions:
  - Initialize variables and files
  - Create sender and receiver threads
  - Wait for threads to finish
  - Perform analysis

Define main function:
- Actions:
  - Parse input arguments
  - Call do_run function

A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network

Design
code 1 - helaryalgo.cpp
1. Define Necessary Data Structures: - Define structs for message types: `RequestMessage`, `Token`, and `TerminateMessage`. - Define a struct `RequestID` to store request origin ID and request time. - Use a `std::list` of `RequestID` to maintain a request array for each node. - Maintain a mapping of neighbor IDs to their request arrays. 2. Socket Creation and Communication Functions: - Implement functions for creating receiving sockets (`createReceiveSocket`) and sending messages (`sendMessage`). These functions will handle socket creation, binding, and communication with other nodes. 3. Helper Functions: - Implement helper functions for converting data types, extracting process IDs from messages, and manipulating message fields. 4. Token Transmission and Reception: - Implement functions `transmitToken` and `receiveToken` to handle token transmission and reception respectively. - These functions should manage token ownership, update logical clocks, and transmit tokens to neighbors. 5. Request Handling: - Implement a function `receiveRequest` to handle incoming request messages. - This function should update request arrays, construct and send request messages to

appropriate neighbors, and trigger token transmission if necessary.

6. Critical Section Entry and Exit:
   - Implement functions `requestCS` and `exitCS` to coordinate access to the critical section.
   - `requestCS` function sends request messages to neighbors and waits until granted access to the critical section.
   - `exitCS` function releases access to the critical section and triggers token transmission.

7. Main Execution Loop:
   - Implement the main execution loop where nodes continuously listen for incoming messages, handle requests, and execute critical sections.

8. Error Handling and Termination:
   - Implement error handling mechanisms for socket communication errors and graceful termination of the algorithm.

9. Integration and Testing:
   - Integrate all components into a single program.
   - Test the algorithm with various network configurations and scenarios to ensure correctness and performance.

By following this program design, you can create a distributed mutual exclusion algorithm that effectively coordinates access to critical sections among multiple nodes in a network.

Code 2 - main.cpp

1. Main Function (main):
   - Reads input from a file which specifies the number of nodes, mutual exclusion counts per node, the node with the initial token, and communication parameters (alpha and beta).
   - Parses the network topology from the input file.
   - Initializes the system, including opening a log file, creating data structures, and starting threads for message sending and receiving.
   - Calls the `run` function to start the distributed mutual exclusion algorithm.

2.Run Function (`run`):
   - Initializes shared variables such as `inCS` (indicating whether a node is in the critical section), `tokenHere` (indicating whether the token is with the node), `logicalClock`, and data structures for maintaining requests (`reqarr`), tokens (`sharedTokenPtrPtr`), and finished processes count.
   - Creates threads for message sending (`workerSenders`) and receiving (`workerReceivers`).
   - Waits for all threads to complete.

- Computes and prints statistics such as the total number of messages exchanged and the average response time per critical section request.

3. Worker Functions (`working` and `receiveMessage`):
   - `working` function simulates a node executing critical section tasks.
     - It performs local computation, requests to enter the critical section, enters the critical section, exits the critical section, and sends termination messages to other nodes.
     - It also handles pending transactions after all critical section requests have been completed.
   - `receiveMessage` function listens for incoming messages from other nodes.
     - It handles different message types (token, request, terminate) and updates shared variables and data structures accordingly.

4. Helper Functions and Data Structures:
   - The program uses helper functions for socket communication (`createReceiveSocket`, `sendMessage`) and file I/O (`fprintf`, `fclose`).
   - Shared data structures include atomic variables (`totalRscvMsgs`, `totalResponseTime`), mutex locks, and various message structures (`Token`, `RequestMessage`, `TerminateMessage`).

5. Randomness Generation:
   - The program uses exponential distributions to model the duration of local computation and critical section execution (`distributionLocalComputation`, `distributionCSComputation`).

Pseudocode
Code 1 :
1. Define constants: - MAX_NODES = 50 - MAX_LENGTH = 500 - FALSE = 0 - TRUE = 1 2. Define custom data types: - RequestMessage - Token - TerminateMessage - RequestID - RequestArrayNode

3. Include necessary libraries:
   - map
   - mutex
   - algorithm
   - set
   - vector
   - sstream
   - assert.h
   - iostream
   - string.h
   - arpa/inet.h
   - list
   - chrono

4. Define global variables:
   - startPort

5. Define function prototypes:

   // Communication functions
   - createReceiveSocket(myPID, myPort)
   - sendMessage(myPID, dstID, message)
   - getNeighborIDfromReqArray(myPID, elecID, reqArray)

   // Utility functions
   - getLudFromToken(numNodes, sharedTokenPtrPtr)
   - extractProcessIDFromAlreadySeen(Q[])
   - unionAlreadySeenNeighbors(alreadySeen, neighbors)
   - constructAlreadySeenString(myPID, neighbors)

   // Token related functions
   - transmitToken(myPID, tokenHere, logicalClock, reqArray, ptrToPtrOfSharedTokens, numNodes, start, fp)
   - receiveToken(myPID, inCS, tokenHere, reqArray, ptrToPtrOfSharedTokens, numNodes, start, fp)

   // Request related functions
   - receiveRequest(myPID, inCS, tokenHere, logicalClock, reqArray, request, neighbors, ptrToPtrOfSharedTokens, numNodes, start, fp)

   // Critical section functions
   - requestCS(myPID, inCS, tokenHere, logicalClock, neighbors, start, fp, lock)
   - exitCS(myPID, inCS, tokenHere, logicalClock, reqArray, ptrToPtrOfSharedTokens, numNodes, start, fp, lock)

6. Main program logic:

- Define main function with necessary parameters

- Initialize variables and data structures

- Create receiving socket for the current process

- Loop forever:
  - Accept incoming connections and messages
  - Handle received messages based on message type:
    - If message is TOKEN:
      - Call receiveToken function
    - If message is REQUEST:
      - Call receiveRequest function
    - If message is TERMINATE:
      - Exit loop and terminate program

- Terminate program and close sockets

7. End of main function

8. Implement the actual functionality for each function according to the provided C++ code.

Code 2:

Create a file log_file.txt

Define global variables:
- totalRscvMsgs = 0
- totalResponseTime = 0

Define a function working(myID, inCS, tokenHere, logicalClock, reqarr, neighbors, sharedTokenPtrPtr, numNodes, finishedProcessesCount, meCounts, alpha, beta, start, fp, lock):
  Define LocalComputationGenerator using default random engine
  Define generatorCSComputation using default random engine
  Define distributionLocalComputation as exponential distribution with parameter 1/alpha
  Define distributionCSComputation as exponential distribution with parameter 1/beta

  Print "Node myID: working -> Starting Critical Section Simulations"

  Initialize a loop i from 1 to meCounts:
    Generate csOutTime from distributionLocalComputation
    Generate csInTime from distributionCSComputation

```
        Print "myID is doing local computation"
        Sleep for csOutTime seconds

        Print "myID requests to enter CS for the ith time"
        Record the current time as requestCSTime

        Call requestCS(myID, inCS, tokenHere, logicalClock, neighbors, start, fp, lock)
        Calculate the response time as the difference between current time and
requestCSTime, and add it to totalResponseTime

        Print "myID ENTERS CS for the ith time"
        Sleep for csInTime seconds

        Print "myID EXITS CS for the ith time"
        Call exitCS(myID, inCS, tokenHere, logicalClock, reqarr, sharedTokenPtrPtr,
numNodes, start, fp, lock)

    Print "myID completed all meCounts transactions"

    Initialize a TerminateMessage
    Set senderID of terminateMessage to myID
    Set type of terminateMessage to TERMINATE

    Initialize a variable k to 0
    While k < numNodes:
        If k is not equal to myID:
            Print "myID sends TERMINATE to k"
            If sending the TERMINATE message to node k fails:
                Print "ERROR :: Node myID: working -> Could not send TERMINATE to k"
                Exit program
        Increment k by 1

    Increment finishedProcessesCount by 1
    While finishedProcessesCount is less than numNodes:
        Acquire lock
        If tokenHere is TRUE:
            Release lock
            Call exitCS(myID, inCS, tokenHere, logicalClock, reqarr, sharedTokenPtrPtr,
numNodes, start, fp, lock)
        Else:
            Release lock

    Print "myID finished any pending transactions"

Define a function receiveMessage(myID, myPort, inCS, tokenHere, logicalClock, reqarr,
finishedProcessesCount, neighbors, sharedTokenPtrPtr, numNodes, start, fp, lock):
```

```
    Create a socket sockfd using createReceiveSocket(myID, myPort)

    Print "Node myID: receiveMessage -> Started listening for connections"

    While finishedProcessesCount is less than numNodes:
        If a connection is accepted:
            Receive data into recvBuffer
            Record the current time as now

            If data is received:
                If the received message type is TOKEN:
                    Increment totalRscvMsgs
                    Record the current time as sysTime
                    Print "myID receives TOKEN from senderID at sysTime"

                    Deserialize the received token message into token

                    Acquire lock
                    Set senderID of token to myID
                    Set sharedTokenPtrPtr to point to token
                    Call receiveToken(myID, inCS, tokenHere, reqarr, sharedTokenPtrPtr,
numNodes, start, fp)
                    Release lock

                Else if the received message type is REQUEST:
                    Increment totalRscvMsgs
                    Record the current time as sysTime
                    Print "myID received REQUEST from senderID with alreadySeen as
|alreadySeen| at sysTime"

                    Acquire lock
                    Call receiveRequest(myID, inCS, tokenHere, logicalClock, reqarr,
requestMessage, neighbors, sharedTokenPtrPtr, numNodes, start, fp)
                    Release lock

                Else if the received message type is TERMINATE:
                    Record the current time as sysTime
                    Print "myID received TERMINATE from senderID at sysTime"
                    Increment finishedProcessesCount by 1

            Close the client socket

    Print "myID stopped receiving threads"

Define a function run(numNodes, meCounts, initTokenNode, alpha, beta, topology):
    Open a file log_file.txt in append mode and store the file pointer in fp
    Record the current time as start
```

Initialize arrays and variables for each node:
- inCS[numNodes] initialized to FALSE
- tokenHere[numNodes] initialized to FALSE, except for initTokenNode which is set to TRUE
- logicalClock[numNodes] initialized to 0
- reqarr[numNodes] initialized as empty maps
- sharedTokenPtrPtr[numNodes] initialized as NULL pointers
- finishedProcessesCount[numNodes] initialized to 0
- locks[numNodes] initialized as mutexes

Create a token object with senderID as initTokenNode and type as TOKEN, and initialize its lud array to -1

Print "Creating receiver threads"
For each node i from 0 to numNodes-1:
    Create a receiver thread with receiveMessage function passing appropriate arguments

Sleep for 1 second

Print "Creating CS executor threads"
For each node i from 0 to numNodes-1:
    Create a sender thread with working function passing appropriate arguments

Join all sender and receiver threads

Calculate the average number of messages exchanged per CS request as totalRscvMsgs divided by (numNodes * meCounts)
Calculate the average response time per CS request in milliseconds as totalResponseTime divided by (1000 * numNodes * meCounts)

Print the total number of messages exchanged and the average messages exchanged per CS request
Print the average response time per CS request in milliseconds

Define the main function:
If number of arguments is less than 2:
    Print an error message indicating missing input file path in arguments
    Exit program with failure status

Open the input file specified in arguments and store the file stream in fin
If opening the input file fails:
    Print an error message indicating error in opening input file
    Exit program with failure status

If number of arguments is less than 3:

```
    Set startPort to 10000
  Else:
    Parse the second argument as startPort

  Seed the random number generator with the current time

  Read input parameters from the file:
  - numNodes
  - meCounts
  - initTokenNode
  - alpha
  - beta

  If numNodes is greater than MAX_NODES:
    Print an error message indicating that number of nodes exceeds MAX_NODES
    Exit program with failure status

  Read the topology from the input file into the topology vector

  Call the run function with input parameters numNodes, meCounts, initTokenNode,
alpha, beta, and topology
```

Message Complexity

Broadcasting a request requires exactly n-1 message. Moving the token requires between 1 (if the sender and the addressee are neighbors) and d messages, where the diameter d is here the length of the longest path (1<d<n-1).

Thus the total number of messages per CS request:

n<<n-1+d

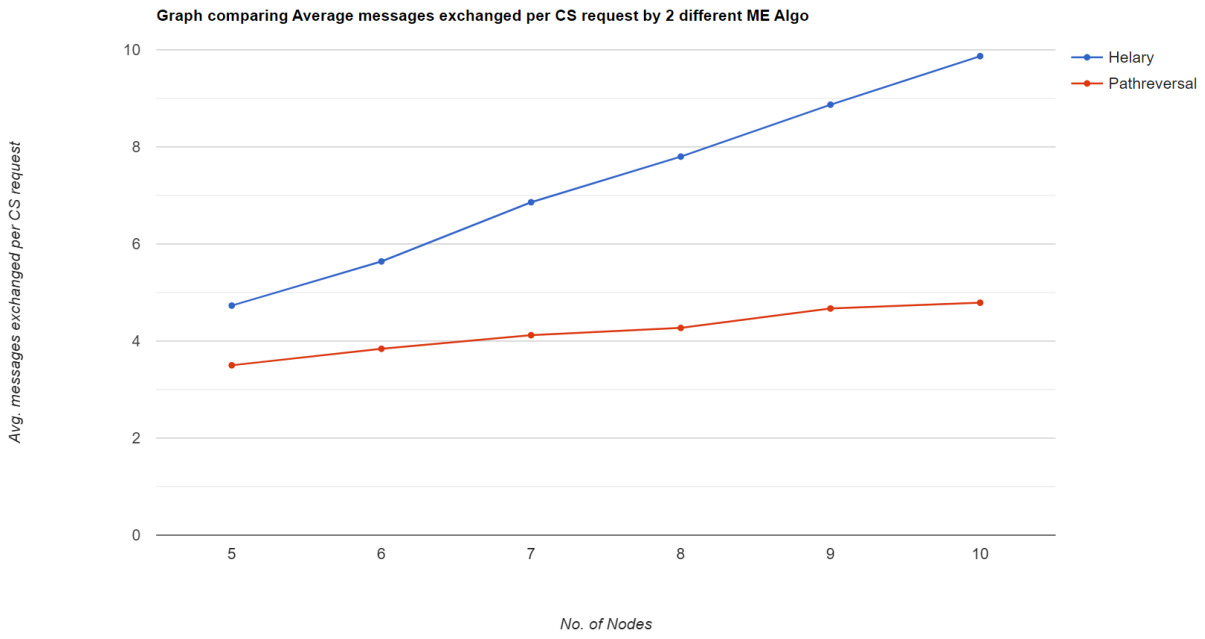
OBSERVATIONS AND RESULTS

1. Average messages exchanged per CS request vs Number of nodes

HELARY'S MUTUAL EXCLUSION ALGORITHM						
Average messages exchanged per CS request						
n ->	5	6	7	8	9	10

	4.6	5.6	7	7.6	8.6	9.8
	4.8	6	6.8	7.8	9	9.8
Average message size	4.8	5.33	6.8	8	9	10
Average	4.73	5.6433	6.86	7.8	8.866	9.86

PATHREVERSAL MUTUAL EXCLUSION ALGORITHM						
Average messages exchanged VS Number of nodes						
n ->	5	6	7	8	9	10
	3.5	3.888889	4.09524	4.1875	4.7778	4.8833
	3.4	3.83333	4.2381	4.35417	4.68518	4.83333
	3.5	3.77778	4.2619	4.375	4.5	4.63333
Average messages exchanged	3.6	3.86111	3.88095	4.1875	4.7222	4.8
Average of all values	3.5	3.84027725	4.1190475	4.2760425	4.671295	4.78749

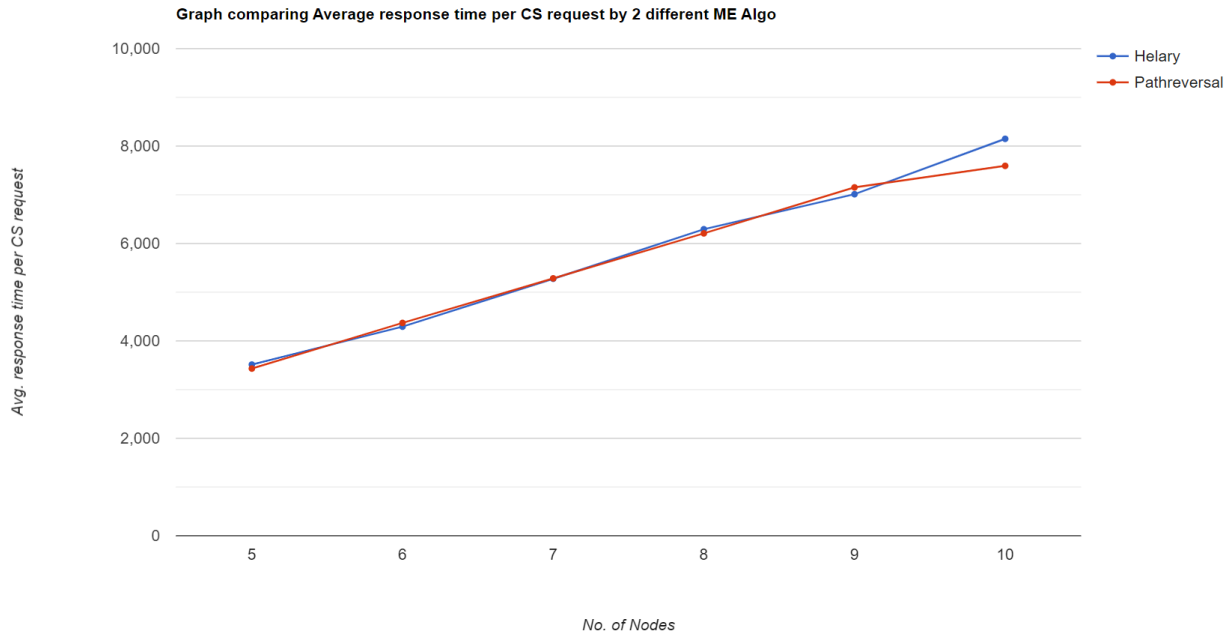


In both the case - Helray's as well as Pathreversal, the average messages exchanged per CS request increases as we increase the number of nodes but the rate of increase in case of Helray is faster than that of Pathreversal.

2. Average response time per CS request VS Number of nodes

HELARY'S MUTUAL EXCLUSION ALGORITHM						
Average response time Per CS request						
n ->	5	6	7	8	9	10
	3508.9	3996	5079	6227	6471	7996
	3511	4455	5383	6336	7260	8214
Average message size	3523	4427	5368	6315	7303	8234
Average	3514.3	4292.66	5276.66	6292.66	7011.33	8148

PATHREVERSAL MUTUAL EXCLUSION ALGORITHM						
Average Response time (in millisecs) VS Number of nodes						
n ->	5	6	7	8	9	10
	3143.65	4327.69	5340.05	6089.22	7123.02	8143.65
	3515.88	4421.84	5173.64	6213.96	7229.95	6729.68
	3527.51	4419	5224.16	6336.53	7272.3	8220.55
Average messages exchanged	3544.51	4308.72	5394.96	6191.1	6982.2	7278.96
Average of all values	3432.8875	4369.3125	5283.2025	6207.7025	7151.8675	7593.21



In both the case - Helray's as well as Pathreversal, the average response time per CS request increases as we increase the number of nodes and the rate of increase is nearly the same.

CONCLUSION

1. Pathreversal Mutual exclusion algorithm exchanges a lesser number of messages as compared to Helary's Mutual Exclusion algorithm. Helary is O(num_of_nodes + diameter -1) and Pathreversal is O(log (num_of_nodes)) because in Path reversal algorithm, messages are sent along a logically connected tree rather than broadcasting the message to every other node in the distributed setup.
2. The response time of both the algorithms is approximately the same.

REFERENCES

1. Mohamed Naimi, Michel Trehel, and André Arnold. 1996. A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal. J. Parallel Distrib. Comput. 34, 1 (April 10, 1996), 1–13. https://doi.org/10.1006/jpdc.1996.0041
2. J. M. Helary, N. Plouzeau, M. Raynal, A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network, *The Computer Journal*, Volume 31, Issue 4, August 1988
3. Book : Distributed Computing by Ajay D. Kshemkalyani and Mukesh Singhal