REPORT

Distributed Computing Course Project Submission

Abstract

Mutual Exclusion problem consists in ensuring that at a given time a logically or physically shared object is accessed by one process at most. This problem is inherently difficult than single node systems, due to the absence of shared memory. All synchronizations need to be done via messages.
This project implements two token-based mutual exclusion algorithms in a distributed network over any arbitrary topology.
The first approach is a O(N - 1 + diameter_of_tree) algorithm which uses logical clocks to order the requests and broadcasts messages to its neighbors when requesting the token.
The second is a O(log (N)) tree-based algorithm based on path-reversal. When a process invokes a critical section, it sends a request to the tail of a queue. A dynamical rooted tree gives the path to this tail.
We have also presented a detailed analysis of the message complexity and the response time for the algorithms and a comparison between the two.

A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network.

J. M. Helary N. Plouzeau M. Raynal

Assumptions

The following assumptions are made for this algorithm:

1. Every interprocess communication line is bidirectional.
2. The network has no loss of messages; no message alteration; finite transmission delay.
3. Message delivery need NOT be FIFO.
4. No assumption is made about the network topology, except connectivity.
5. The only knowledge owned by a process is local - it's ID and the name of its neighbors.

Message Formats

```
struct RequestMessage
{
   enum messageType type;
    int senderID;       // Process ID which sent the request.
    int reqOriginId;    // Process ID original request creator.

    LLONG reqTime;      // Lamport's logical clock value of the requesting
                        // process at the request creation time.

    char alreadySeen[MAX_LENGTH];
    /* process-IDs delimited by comma. Every process whose ID is in
       alreadySeen has received or is about to receive the request.
       This is used to reduce the message complexity.
    */
} RequestMessage;
```

```
struct Token
{
   enum messageType type;
    int senderID;               // Process ID which sent the token.

    int elecID;                // Process ID the token's final addressee.

    LLONG lud[MAX_NODES];
    /*
        Array whose ith index stores the value that Process_i logical-clock
        had when Process_i gave the token to another process.
    */
} Token;
```

Overview of the Algorithm

• When a process wants to enter the critical section it sends a request message to its neighbors and waits for the token message.

• Upon receiving a request message, a process P broadcasts that request to its neighbors not belonging to the `alreadySeen` array of the request message.
This is done to reduce message complexity by not sending requests to the nodes that have already been sent.
The request is added to P's set of known pending requests.

• If the process P owns the token and is outside the critical section, it extracts the oldest request R from its known requests sets.
The oldest request R(PID, Time) is found by the following:

$$\forall\ (PID', Time') \in reqArray : (PID < PID') \vee (PID == PID' \wedge Time < Time')$$

P then sends the token to the creator of R through N, the neighbor of P which sent him the request R. If process P is inside the critical section, it will behave as stated above upon exiting it.

• Upon receiving the token message, process P keeps it if the token's addressee is itself. Otherwise, P hands it over to N, the neighbor which sent him the request being serviced.

• A process can enter the critical section only if it owns the token.

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 — l).
Thus the total number of messages per CS request:

$$n << n - 1 + d$$

Inferences from the Algorithm

The following properties hold for the algorithm. (for proof, refer paper)

- The path followed by the token from its sender P_i to its final addressee P_j is acyclic.
- The algorithm is deadlock-free.
- The algorithm is starvation-free.

Drawbacks/Limitations of the Algorithm

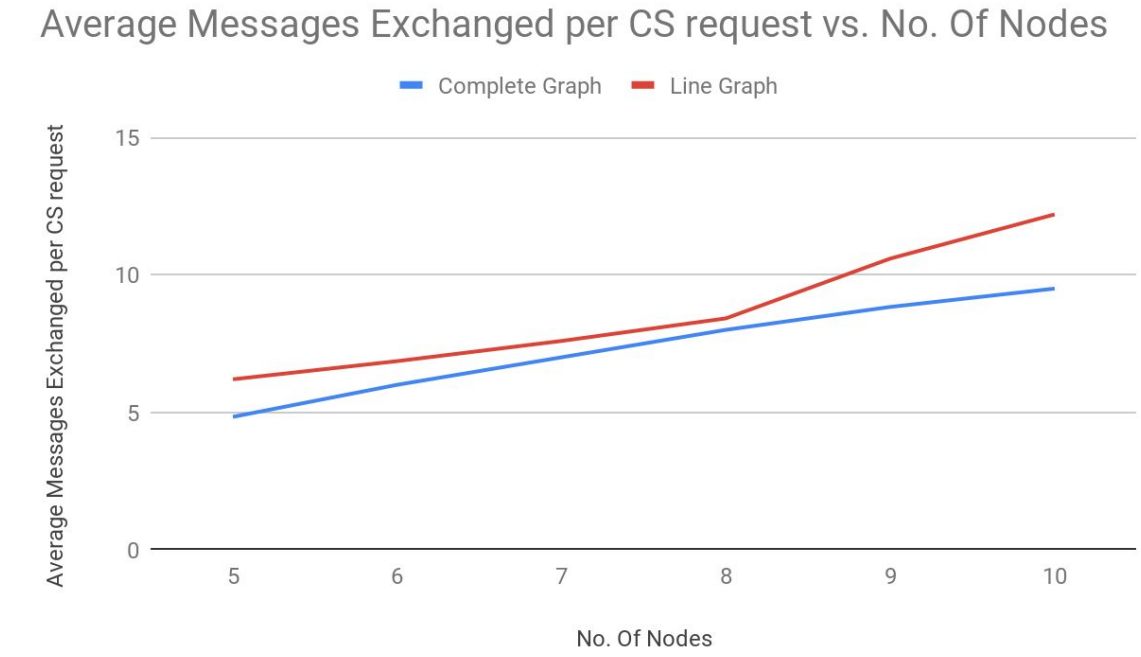
- Logical clock values are assumed to be unbounded.
- The algorithm does not handle cases of node/network failures.
- Reconfiguration of the system, while some requests are pending, is more difficult in this algorithm.

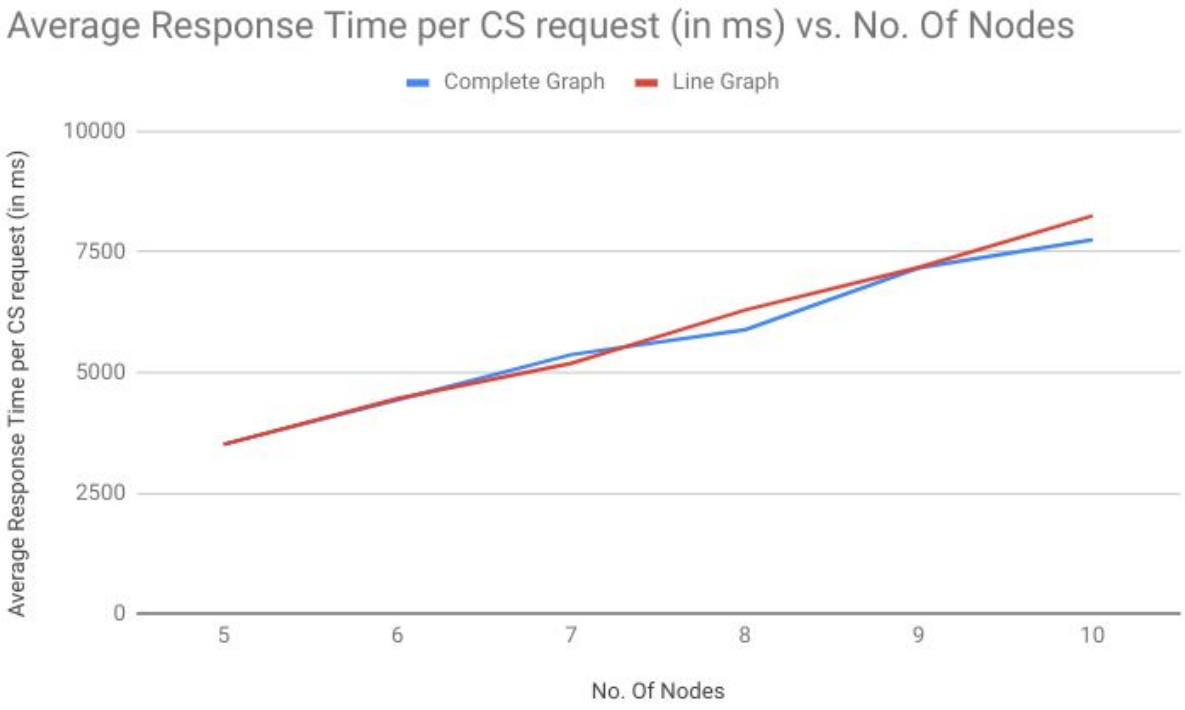
Experiments

Graph Topology: Complete Graph, Line Graph

Parameters:

- Time is calculated using 'chrono high-resolution clock' in C++
- Number of CS executions - 6
- Initial Node With Token - 4
- **α**- 0.65 (Time spent outside the CS is exponentially distributed with an average of **α**)
- **β** - 0.15 (Time spent inside the CS is exponentially distributed with an average of **β**)


Average Messages Exchanged per CS request vs. No. Of Nodes
Complete Graph
Line Graph
Average Messages Exchanged per CS request
15
10
5
0
5
6
7
8
9
10
No. Of Nodes


Average Response Time per CS request (in ms) vs. No. Of Nodes
Complete Graph
Line Graph
Average Response Time per CS request (in ms)
10000
7500
5000
2500
0
5
6
7
8
9
10
No. Of Nodes

Observations

- Message complexity is O(numNodes - 1 + diameter).
  Therefore, the line graph has more message complexity than the complete graph.
- Number of messages increases with number of nodes.
- Response time increases with number of nodes.
  Though response time for line & complete graph is very close to each other.
- The high response time of this algorithm is due to the reason that request messages are broadcasted to all the neighbors.
  Since the algorithm guarantees starvation-free, sometimes a node might be asked to transfer the token to another node(with lesser clock value) even if it wants to enter the CS at that instant.

A Log (N) Distributed Mutual Exclusion Algorithm using Path Reversal

Mohamed Naimi, Michel Trehel, André Arnold

Assumptions

1) The communication between the nodes is assured to be perfect(no message losses, duplications or modifications)
2) Every node in a distributed setting has the information of its ID and the IDs of its neighbors.
3) The channels between any two nodes are unidirectional
4) The nodes are connected in such a fashion that every node is directed to another node until we get a node with no outgoing edges in it. This node will initially have the token.
5) There is no assumption regarding the topology of the processes, except connectivity.
6) Message delivery need NOT be FIFO.

Message Format:

```
// enum for different kinds of messages
enum messageType{
  PRIVILEGE,
  REQUEST,
  TERMINATE
};

// Definition of message structure used in send and receive process
typedef struct Message{
  enum messageType type;
  int senderID;
  int reqProcess;
} Message1;
```

Overview of the Algorithm

- For every node in the system, it maintains two pointers, *father* and *next*.

- Initially, the graph topology of the system is present in the form of a logically rooted tree, whereby the root node holds the token to enter CS and all other processes are pointing to it, either directly or indirectly.

- Father tries to maintain the node to which the request messages are sent. From 'father' to 'father', a request is transmitted to the root which has the token. Furthermore, if the requesting process is not the root, the rooted tree is transformed, the original requesting process is the new root and the processes located between the requesting process and the root will have the new root as the father.

- The next variable is responsible for passing of the token to that request which has requested for the token at the earliest. From 'next' to 'next', the token for entering CS is passed to all the requested processes in a FIFO order.

Message Complexity

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

Inferences from the Algorithm

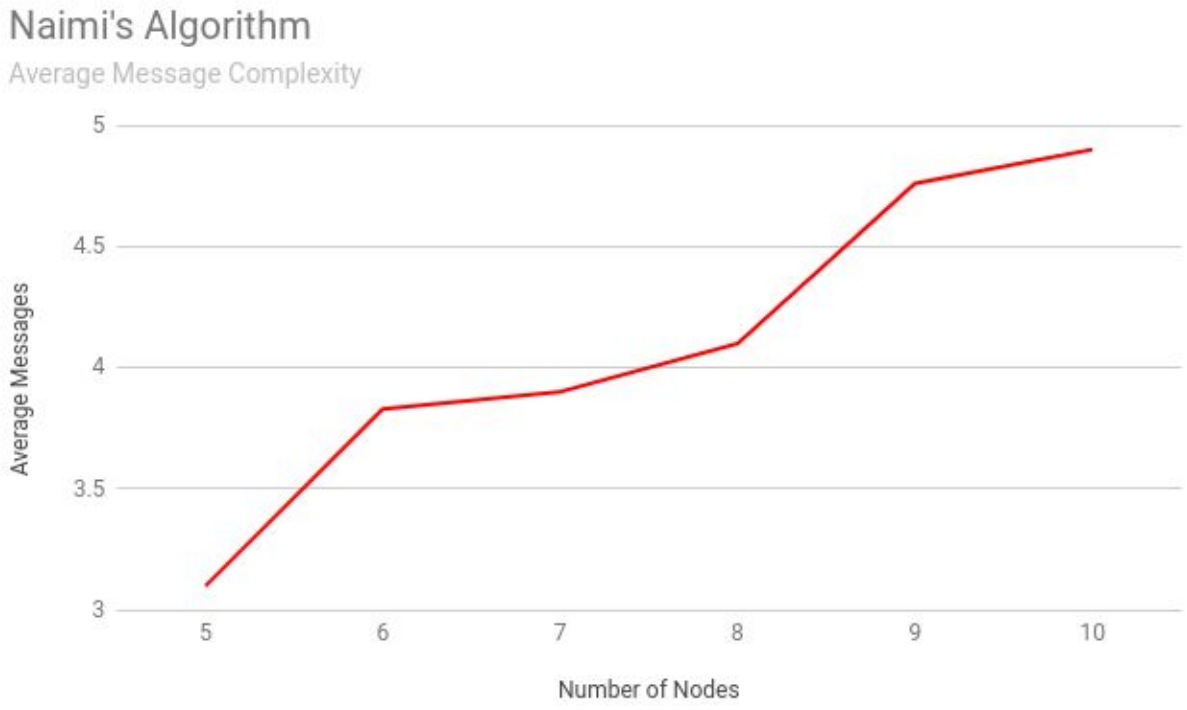
The following properties hold for the algorithm. (for proof, refer paper)

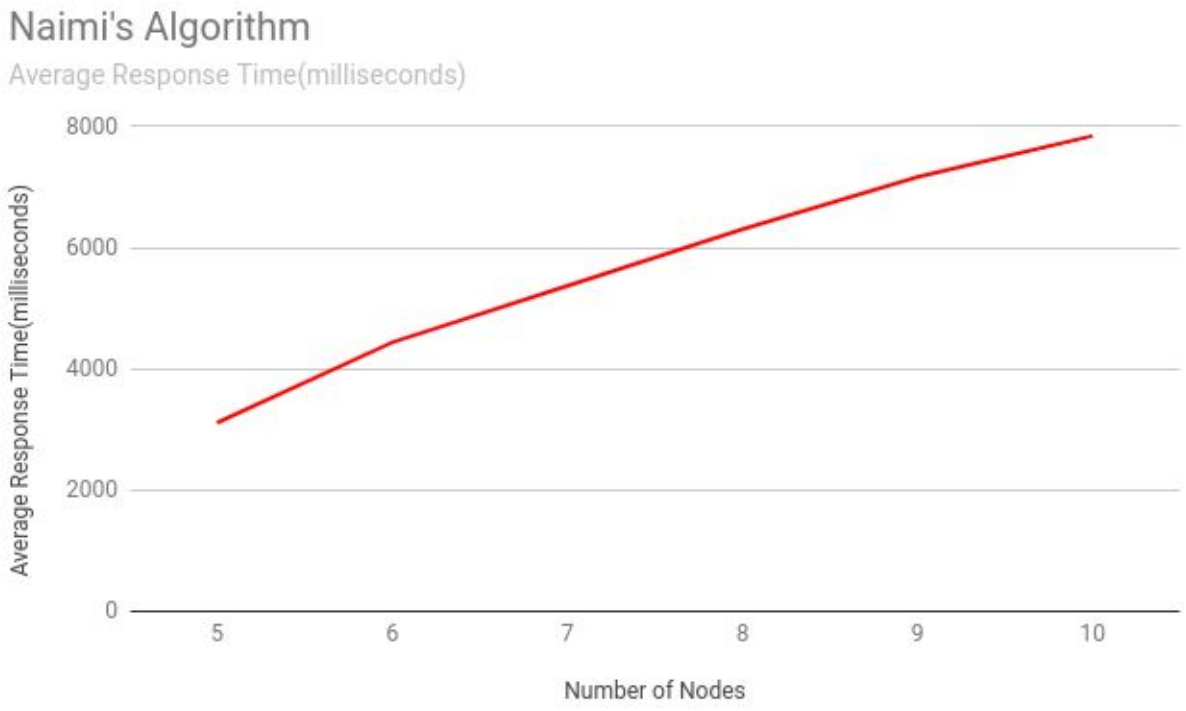
1) The root node of the logically connected tree will always contain the token and all other nodes point to it, either directly or indirectly.
2) The algorithm is deadlock-free.
3) The algorithm is starvation-free.

Experiments

Parameters:

1) Time is calculated using 'chrono high-resolution clock' in C++
2) Number of CS executions - 7
3) Initial Node With Token - 3
4) **α**- 0.65 (Time spent outside the CS is exponentially distributed with an average of **α**)
5) **β** - 1.55 (Time spent inside the CS is exponentially distributed with an average of **β**)

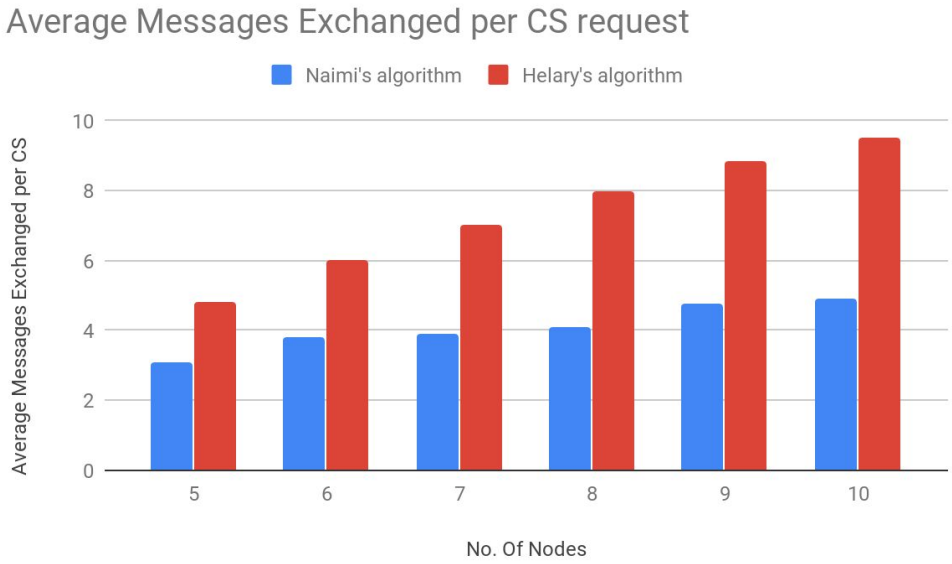


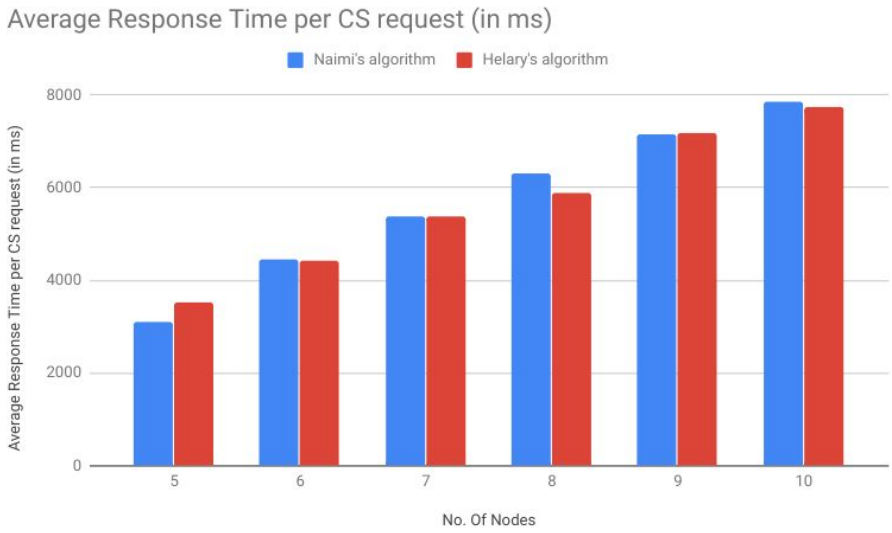


Observations

1) Both average message complexity and response time increase with the number of nodes in the system.
2) Average message complexity is of the form O(log(n))
3) The low message complexity for Naimi's algorithm is due to the reason that messages are only passed along the logically connected tree and the nodes also help in routing the request messages sent by another node to the root of the tree.

Comparison between Helary's Algorithm and Naimi's Algorithm





Observations

- Helary is O(n-1+d) and Naimi is O(log(n)).
  Therefore, Helary exchanges more messages than Naimi. This is beacause of the fact that in Naimi's algorithm, messages are sent along a logically connected tree rather than broadcasting the message to every other node in the distributed setup
- The response times of both the algorithms are almost the same.

References

- A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network - https://academic.oup.com/comjnl/article/31/4/289/380471
- A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal - https://dl.acm.org/citation.cfm?id=234302
- Distributed Computing: Principles, Algorithms, and Systems
  Textbook by Ajay D. Kshemkalyani and Mukesh Singhal - https://eclass.uoa.gr/modules/document/file.php/D245/2015/DistrComp.pdf