Distributed Systems: Assignment 1

Kousshik Raj (17CS30022)

08-02-2021

1. Consider a system with a clock that runs slow with a drift rate of 10^{-5}. Cristian's algorithm is used to synchronize the clock from a time server every 30 minutes. To estimate the delay in getting the time, a simple algorithm is used - the system just measures the round trip delay and divides it by 2. The maximum delay of the link between the system and the time server is 2ms (one way). The processing time at the client after receiving the time from the server is 1ms. Assume that the time server's clock has zero drift. What is the maximum possible difference between the clocks of the time server and the client just before and after a synchronization? Show all calculations.

   Solution.

   - **After Synchronisation:** The maximum difference occurs when the algorithm underestimates the time taken for the message to travel from the time server to the client, as it can then accumulate with the processing time at the client node. The maximum underestimation of time happens when the message takes almost negligible time from the client to the time server but takes the maximum possible delay (2ms) for the other way round.

     Time for message travel from client to timeserver $\approx$ 0ms.
     Time for message travel from timeserver to client = 2ms.
     Estimated time by client for message travel from timeserver to client $\approx \frac{0+2}{2} = 1$ms.
     Difference accumulated due to erroneous estimation $= 2 - 1 = 1$ms.
     Difference accumulated due to processing time at client node = 1ms.
     Total difference between timeserver and client $= 1 + 1 = \boxed{\textbf{2ms}}$.

   - **Before Synchronisation:** Since resynchronisation occurs every 30 minutes, the maximum difference will also accumulate just before the synchronisation.

     Difference accumulated just after synchronisation = 2ms.
     Difference accumulated due to the drift of the client clock $\approx 10^{-5} * 30 * 60\text{s} = 18$ms.
     Total difference accumulated just before synchronisation $= 18 + 2 = \boxed{\textbf{20ms}}$

2. Repeat the above problem for Berkeley's algorithm with the same parameters.

   Solution.

   - **After Synchronisation:** Here, first let's assume the processing time during communication can be neglected due to the fact that the message can convey how much time has elapsed during the processing. So, other than the final processing time at the client, all other processing times can be neglected. And moreover, let's suppose that the client is lagging behind the timeserver (calculation is applicable for both cases) and that the initial time at the client is 0.

     Initial time of client before start of synchronisation = 0.
     Initial time of timeserver before start of synchronisation = a.
     Time taken for message travel from timeserver to client = x.
     The time communicated by the client to the timeserver = x.
     Time for message travel from client to timeserver = y.
     Estimated time of client by timeserver (by avg. of round trip delay) $= x + \frac{x+y}{2}$.
     Time of timeserver during the calculation of avg. $= a + x + y$.
     Avg. time calculated by timeserver $= \frac{a}{2} + \frac{5x+3y}{4}$.
     Adjustment for client calculated by timeserver $= \frac{a}{2} + \frac{y-x}{4}$.
     Time of client when it receives adjustment $= x+y$. (Ignoring msg. delay for adjustment, as it will be cancelled)
     Time of client after adjustment $= \frac{a}{2} + \frac{3x+5y}{4} + 1$ms. (Taking processing time into account).
     Maximum difference between timeserver and client $= \max\left(\frac{y-x}{2} + 1\text{ms}\right) = \frac{2-0}{2} + 1\text{ms} = \boxed{\textbf{2ms}}$.

- **Before Synchronisation:** Since resynchronisation occurs every 30 minutes, the maximum difference will also accumulate just before the synchronisation.

  Difference accumulated just after synchronisation = 2ms.
  Difference accumulated due to the drift of the client clock $\approx 10^{-5} * 30 * 60$s = 18ms.
  Total difference accumulated just before synchronisation = $18 + 2 = \boxed{\textbf{20ms}}$

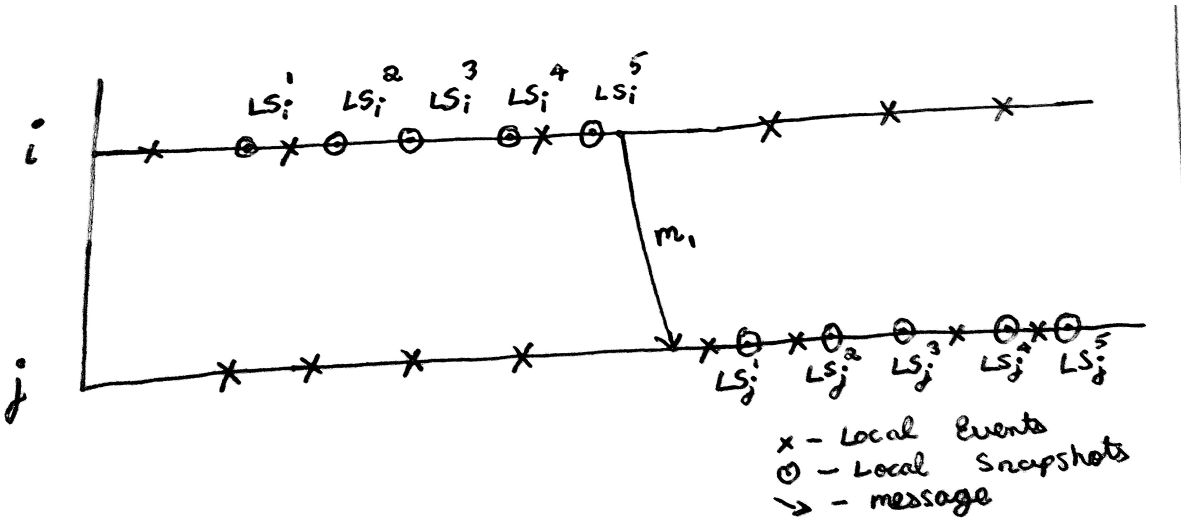
3. Suppose that 3 nodes always have their clocks synchronised to within 1 second of each other. A simple global snapshot is proposed in which each node takes its local snapshot at the start of each hour, i.e., at 9 AM, 10 AM, ... as per its local clock; no other messages are exchanged. If we look at the set of local snapshots of the 3 nodes taken at 10 AM, is it guaranteed to be a consistent checkpoint? Justify.

   Solution. **No**, it is not guaranteed that the snapshots correspond to a consistent checkpoint. Let's consider an example. Assume, w.r.t the global clock, the local clocks of node A, B and C are 0 millisecond, -200 milliseconds and -700 milliseconds ahead, respectively. For simplicity, assume all times mentioned here on are with reference to a global clock at 10 AM.

   So according to the assumed local times, the times of capture of local snapshots for node A (LS_A), B (LS_B) and C (LS_C) are 0 millisecond, 200 milliseconds and 700 milliseconds, respectively. Now, let's say, node A sends a message m_{AC} at 150 milliseconds to node C which is delivered to it at 500 milliseconds. Then, we can see that $m_{AC} \notin LS_A$ and $m_{AC} \in LS_C$, thereby making it an inconsistent message. Thus, we have an inconsistent checkpoint.

4. Suppose that two processes i and j take five snapshots each of their local states independently at arbitrary times, with no communication for snapshot capture between them. Channel states are implicitly captured in the process states using message logs. Is it possible to have a scenario in which no pair of snapshots, one taken from i and one taken from j, is consistent? Justify clearly with the help of a space-time diagram.

   Solution.



   Yes. From the space-time diagram, we can see that $m_1 \notin LS_i^x\ \forall 1 \leq x \leq 5$ and $m_1 \in LS_j^y\ \forall 1 \leq y \leq 5$. This means that $inconsistent(LS_i^x, LS_j^y) = \{m_1\}\ \forall 1 \leq x, y \leq 5$. Thus none of the 25 pair of snapshots is consistent.

5. "Highly stable atomic clocks and clock synchronization mechanisms are available today to keep computer clocks synchronised to within a microsecond of the real time, if not better. In view of this, logical clocks are no longer necessary and should not be studied anymore." - Do you agree or disagree with the statement?

   Solution. ***No***, I do not agree with the statement. Even in such cases, logical clocks are necessary.

   - **Cost:** For applications that do not need accurate external global timings, maintaining such highly stable atomic clocks is extremely cost ineffective and unaffordable for a normal distributed system.
   - **Insufficient bound:** Nowadays, modern distributed systems, owing to advancement in technology, have events that occur within less than microseconds of each other as well. Hence, this synchronization bound is still not effective in ordering such events.

6. "Recording a consistent, but not strongly consistent, global state is of no use in practice as messages in transit are lost if the system is restarted from a consistent state" - Do you think this statement is correct? Argue briefly and clearly.

   Solution. ***No***, this statement is not entirely correct.

   - Applications that can tolerate the loss of messages (unreliable communication) do not require the checkpoints to be strongly consistent, as both consistent and strongly consistent checkpoints hold almost the same meaning in such cases.
   - With the help of consistent checkpoints and message logging (taking a log of all messages that are being sent from a process along with their timestamps), restart of consistent checkpoints that are not strong can happen without the loss of messages that are in transit, as we can resend these messages with the help of the logs.

7. Write a distributed algorithm for constructing a spanning tree rooted at a specific node in an asynchronous, reliable system. The root node will initiate the algorithm. At the termination of the algorithm each node should know its parent and children set in the spanning tree. In addition, the root node should know that the spanning tree has been constructed.

   Solution.

   Model/Assumptions

   - Asynchronous System
   - Reliable Communication
   - Arbitrary Connected Topology - Undirected (Bidirectional links)
   - Each node has unique id
   - Each node i knows its neighbour set $N(i)$

   Messages

   - D(sender) - Dummy message
     - sender: the unique id of the sender of the message
   - C(sender, child) - Child message
     - sender: the unique id of the sender of the message
     - child: a binary variable indicating whether the sender is a child of the recipient or not

   Program for node i

   (a) **Variables**
   - parent: the parent of the node i in the spanning tree
   - children: the set of children of the node i in the spanning tree
   - root: whether the node is the root or not
   - seen: total number of replies seen from its neighbours

   (b) **Initialisation**
   - parent $= i$ (Will be updated later except for root)
   - children $= \phi$
   - root $= 0$
   - seen $= 0$ (Initially no messages are seen)

   (c) **Handler Functions**
   - **Init Function** (Called by root node i when it wants to start the algorithm)

```
1: $root = 1$
2: for all $j \in N(i)$ do
3:     send D($i$) to $j$
```

- **Receiving D(j) from node j**

```
1: if $parent = i$ and $root \neq 1$ then                ▷ If receiving for the first time
2:     $parent = j$
3:     $cnt = 0$
4:     for all $j \in N(i)$ do        ▷ Send a dummy message to all outgoing channels other than parent
5:         if $j \neq parent$ then
6:             send D($i$) to $j$
7:             $cnt = cnt + 1$
8:     if $cnt = 0$ then                ▷ If leaf node, send to parent that it is its child
9:         send C($i$, 1) to $parent$
10: else                                ▷ Send to $j$ that it is not a child
11:     send C($i$, 0) to $j$
```

- **Receiving C(j, x) from node j**

```
1: $seen = seen + 1$
2: if $x = 1$ then
3:     Add $j$ to $children$
4: if $seen = |N(i)| - 1$ and root $\neq 1$ then          ▷ When subtree formed, inform parent
5:     send C($i$, 1) to $parent$
6: else if $seen = |N(i)|$ and root $= 1$ then             ▷ If spanning tree is formed
7:     END
```

Message Complexity $= O(|E|)$

Each node knows which channel is its parent and which channels are its children. Moreover, once the root receives a message from all of its outgoing channel, it knows that the spanning tree is completely formed, and terminates the algorithm.

8. Suppose you are given a connected, undirected graph with a special node i and an upper bound T on the message delay over any link. Design a global state collection algorithm that does not rely on FIFO channels and does not use piggybacking. At the end of the algorithm, the global state should be available at node i and it should know that the state collection is complete. Analyze its message complexity. Clocks in the nodes are not synchronized, but you can assume that they have no drift. Assume that every process knows an upper bound on the no. of processes in the system, and processing times at nodes are negligible.

Solution. In this case of non-FIFO channels, we can use **message inhibition** to tackle the problem, as we know the maximum delay over any link. Here, outgoing messages are stopped for certain period of time and are allowed to pile up before being sent. Since there is no clock drift, the delay T can be calculated at the local node itself. We then send the recorded snapshot to the root along its parent, and since we know the maximum delay and upper bound of the number of processes, the root can decide the termination of the algorithm after certain period of time.

Model/Assumptions

- Synchronous System - maximum one-way delay T
- Reliable Communication
- Non - FIFO channels
- Arbitrary Connected Topology - Undirected (Bidirectional links)
- Each node has unique id
- Each node i knows its neighbour set $N(i)$
- Each node i knows the upper bound on the number of processes n

Messages

- M(sender) - Marker message
  - sender: the unique id of the sender of the message
- S(sender, data) - Snapshot Data
  - origin: unique id of the origin of the snapshot
  - data: a local snapshot of a node and its channel states

Program for node i

(a) **Variables**

- parent: the parent of the node i in the spanning tree
- root: whether the node is the root or not
- seen: total number of marker messages seen
- snapshot: the local snapshot (LS_i) and the corresponding channel states (C_{ji})

(b) **Initialisation**

- parent $= i$ (Will be updated later except for root)
- root $= 0$
- seen $= 0$ (Initially no messages are seen)
- snapshot $= \phi$

(c) **Handler Functions**

- **Init Function** (Called by special node i when it wants to start the algorithm)

```
1: $root = 1$
2: **Stop** outgoing messages for time $T$        ▷ Inhibit outgoing messages to clear the channels
3: Record $LS_i$ and add to *snapshot*
4: **for all** $j \in N(i)$ **do**
5:     send M($i$) to $j$
6: **Stop** outgoing messages for time $T$        ▷ Inhibit outgoing messages so that marker msg can reach
7: **Wait** $3nT$ time and **END**        ▷ Time taken for all snapshots to reach the root
```

- **Receiving M(j) from node j through channel C_{ji}**

```
1: $seen = seen + 1$
2: **if** $parent = i$ **and** $root \neq 1$ **then**        ▷ If receiving for the first time
3:     $parent = j$
4:     Record $LS_i$ and add to *snapshot*
5:     Record $C_{ji} = \phi$ and add to *snapshot*
6:     **Stop** outgoing messages for time $T$
7:     **for all** $j \in N(i)$ **do**        ▷ Send a marker message to all outgoing channels
8:         send M($i$) to $j$
9:     **Stop** outgoing messages for time $T$
10: **else**        ▷ If already received a marker message
11:     $Q$ = set of all msgs. arriving by $C_{ji}$ after first and before current marker.
12:     Record $C_{ji} = Q$ and add to *snapshot*
13: **if** $seen = |N(i)|$ **and** $root \neq 1$ **then**        ▷ Sending snapshot back to root
14:     send S($i$, *snapshot*) to *parent*
```

- **Receiving S(x, *data*) from node j**

```
1: **if** $root \neq 1$ **then**
2:     send S($x$, *data*) to *parent*
3: **else**
4:     **Collect** S($x$, *data*) into **global snapshot**
```

Message Complexity $= O(|E| + n^2)$. Here, every node sends a marker message through every outgoing channel, hence $2|E|$ marker messages. Then all the recorded snapshots are being collected at the root, which is quadratic on the number of nodes in the worst case.

Time Complexity $= 2(n-1)T + (n-1)T = O(nT)$. Since every node waits for T before sending marker message, it effectively takes $2T$ for one marker message to reach from one node to next. Since, maximum hops is bound by the number of nodes, all nodes receive the marker message by $n-1$ hops and it takes another $n-1$ hops to send the snapshots back to the root in the worst case.

9. Sometimes we want to restart a system from a predefined reset state such that computation starts again from this reset state where all processes are reset to a predefined local state and all channels are empty. Design an algorithm to restart a system from a predefined reset state in an asynchronous distributed system with FIFO channels. The underlying graph is undirected and connected. For simplicity, you can assume that there is only one initiator of the restart algorithm. What is the message complexity of your algorithm?

Solution. The main source of problem we have to tackle is to differentiate messages sent from reset nodes and from nodes that have still yet to be reset. We can do this by sending a marker message along the channel. Messages sent after the marker message are the ones sent after the node reset. Thus, we have the following algorithm.

Model/Assumptions

- Asynchronous System
- Reliable Communication
- FIFO channels
- Arbitrary Connected Topology - Undirected (Bidirectional links)
- Each node has unique id
- Each node i knows its neighbour set $N(i)$

Messages

- M(sender) - Marker message
  - sender: the unique id of the sender of the message
- P(data) - Application messages (Not part of algorithm)
  - data: the data and metadata present in the application message

Program for node i

(a) **Variables**

- seen: binary variable to determine whether a marker message has already arrived
- V: a set of valid nodes that has been reset

(b) **Initialisation**

- seen = 0
- V = ϕ

(c) **Handler Functions**

- **Init Function** (Called by the node i when it starts the restart algorithm)

```
1: $seen = 1$
2: Reset to predefined state
3: for all $j \in N(i)$ do                    ▷ Send marker message through all channels
4:     send M($i$) to $j$
```

- **Receiving M(j) from node j**

```
1: Add $j$ to $V$                                        ▷ Mark the channel as valid
2: if $seen \neq 1$ then                                 ▷ If receiving for the first time
3:     Reset to predefined state
4:     for all $j \in N(i)$ do                 ▷ Send a marker message to all outgoing channels
5:         send M($i$) to $j$
6:     $seen = 1$
7: if $|V| = |N(i)|$ then          ▷ Once all neighbours have been reset, terminate algorithm
8:     END
```

- **Receiving P($data$) from node j** (Receiving an application/process message)

```
1: if $seen = 1$ and $j \notin V$ then       ▷ If the node is reset but the sender is not, ignore msg
2:     Drop P
3: else
4:     Process P
```

In this algorithm, once a marker message comes from a particular channel, we know that the node at the other end has already been **reset**. Hence, we can consider all messages after that to be **valid messages** and the ones before the marker message as invalid ones for the current node that has been reset.

Message Complexity $= 2|E| = O(|E|)$.

10. Given an arbitrary synchronous network with unique ids, give a protocol that elects exactly k leaders in the network. When your protocol ends, exactly k nodes will declare themselves as leaders, and all other nodes will know that they are not leaders. Make any additional assumptions you need, but write them down clearly. Analyze the time and communication complexity of your algorithm.

    Solution. We will use an algorithm similar to FloodMax. The algorithm runs in rounds, and in each round r, every node maintains the k largest ids it has seen so far. At the end, the k largest ids elect themselves as the leader.

    Model/Assumptions

    - Synchronous System: Max one way delay α, Max processing time β.
    - Reliable Communication
    - Arbitrary Connected Topology - Undirected (Bidirectional links)
    - Diameter, maximum degree and the number of edges of graph are D, Δ and m, respectively
    - Each node has unique id
    - Each node i knows its neighbour set $N(i)$

    Messages

    - M(sender, ids) - Max IDs message
        - sender: the unique id of the sender of the message
        - ids: a set of at most k ids

    Program for node i

    (a) **Variables**
    - r: a variable to indicate the round number
    - lim: number of neighbour message received
    - L: a set of at most k largest ids seen so far
    - temp: a temporary variable to merge sets

    (b) **Initialisation**
    - r = 0
    - lim = 0
    - L = $\{i\}$
    - temp = $\{i\}$

    (c) **Handler Functions**
    - **At start of each round** (Called by every node)

```
1: for all $j \in N(i)$ do                                   ▷ Send $L$ to all neighbours
2:     send M($i$, $L$) to $j$
```

    - **Receiving M(j, L_j) from node j**

```
1: $lim = lim + 1$
2: $temp = temp \cup L_j$                 ▷ Merge all the ids it receives into a single set
3: if $lim = |N(i)|$ then                 ▷ When message received from all neighbours
4:     $L = \min(k, |temp|)$ largest elements from $temp$
5:     $temp = L$
6:     $lim = 0$
7:     $r = r + 1$
8:     if $r = D$ then                     ▷ When $D$ rounds are done
9:         if $i \in L$ then
10:            Declare as leader
11:        END
```

    Time Complexity $= O((\alpha + \beta)D) = O((\alpha + k\Delta \log(k\Delta))D)$. Here, the processing time β is expanded, and if we utilise a balanced binary search tree to find the k largest ids from the resultant union of the messages from all the neighbours, we can achieve the mentioned complexity.

    Communication Complexity $= O(k \log I \cdot m \cdot D)$. Here, it is assumed that I is the maximum possible node id and hence we need $\log I$ bits to represent a single id, and as each message has at most k such ids, it takes up $O(k \log I)$ space. Furthermore, in each of the D rounds, each node sends such a message to all its neighbours. Hence, the mentioned complexity.

11. Suppose you are given a distributed algorithm to find an undirected spanning tree of an arbitrary connected network. Assume that any two adjacent nodes have distinct ids (but all nodes in network need not have distinct ids). How can you use this algorithm to elect a single leader node in an arbitrary connected network? At the end of the algorithm, only one node will declare itself as the leader (others might not know who the leader is). What is the message complexity of the algorithm ?

    Solution. Here, we can use an algorithm similar to trimming the leaves after we have the spanning tree. All leaves will send a vote message to its unique parent and when a node receives vote messages from all but one of its neighbours, it sends a vote message to that neighbour. This keeps happening till two vote message are sent in the same edge, after which the node with maximum id of the two endpoints is elected as leader. A maximum can be chosen because any two adjacent nodes have distinct ids.

    Model/Assumptions

    - Asynchronous System
    - Reliable Communication
    - Arbitrary Connected Topology - Undirected (Bidirectional links)
    - Algorithm to convert the Arbitrary Connected Topology to Spanning Tree
    - Each node i knows its neighbour set $N(i)$

    Messages

    - V(sender) - Vote message
        - sender: the id of the sender of the message

    Subroutine

    - **UnRootSpanTree(i)** (The blackbox algorithm to find a unrooted ST from arbitrary network)
      Input: Node i
      Output: Set of neighbours of node i in the spanning tree

    Program for node i

    (a) **Variables**
    - dest: target of the vote message this node will send
    - R: Senders of vote messages
    - E: Neighbours in the spanning tree

    (b) **Initialisation**
    - dest = -1 (Yet to send a vote message, -1 is not any of the ids in network)
    - R = ϕ
    - E = ϕ

    (c) **Handler Functions**
    - **Init Function** (Called by every node at the start of the algorithm)

```
1: $E$ = $UnRootSpanTree(i)$      ▷ Use the given subroutine to find the neighbours in the spanning tree
2: if $|E| = 1$ then              ▷ Send vote message to the only neighbour
3:     $dest$ = the only node in $E$
4:     send V($i$) to $dest$
```

    - **Receiving V(j) from node j**

```
1: if $dest \neq -1$ and $dest < i$ then     ▷ If already sent vote message to a smaller id, declare leader
2:     Declare Leader
3:     END
4: else if $dest = -1$ then
5:     Add $j$ to $R$
6:     if $|R| = |E| - 1$ then               ▷ Received votes from all but one neighbour, send vote to it
7:         $dest$ = the only node in $E$ not in $R$
8:         send V($i$) to $dest$
```

    Message Complexity = $|E| + 1$. Every edge has exactly one vote message travelling along it except for the final edge, where there are two messages.

12. Design a permission based distributed mutual exclusion algorithm in which a node will require 0 messages per critical section entry in the best case. However, in the worst case, it will require $2(n-1)$ messages per critical section entry similar to Ricart-Agarwala's algorithm. The proposed algorithm need not maintain the fairness property of Ricart-Agarwala's algorithm.

Solution. Here, we will use an algorithm similar to Ricart-Agarwala but make a change that when a node receives permission from all other nodes to enter CS, it can do so **multiple times without further permissions** from other nodes. It is easy to observe that, though this achieves a better best time complexity, it might also lead to starvation, as a particular node might permanently take hold of the CS.

Model/Assumptions

- Asynchronous System
- Reliable Communication
- Fully Connected Topology
- Each node has unique id
- Every node knows the total number of nodes n

Messages

- R(sender, ts) - Request message
  - sender: the unique id of the sender of the message
  - ts: time stamp of request
- A(sender) - Reply/ACK message
  - sender: the unique id of the sender of the message

Program for node i

(a) **Variables**

- acks: number of acknowledgements received
- cnt: number of requests for node i entering CS
- already: if node is already in CS or not
- best: time of most earliest request made by node i
- pending: pending requests from other nodes waiting ack

(b) **Initialisation**

- acks = 0
- cnt = 0
- already = 0
- best = ∞ (∞ indicates no requests)
- pending = ϕ

(c) **Handler Functions**

- **Init Function** (Whenever node i wants to enter CS)

```
1: $cnt = cnt + 1$                                  ▷ Increase the self request count
2: if $already \neq 1$ and $best = \infty$ then     ▷ If node not in CS, send reqs to other nodes (if first request)
3:     send R($i$, $time$) to all other nodes
4:     $best = time$
```

- **Release Function** (Whenever node i exits CS)

```
1: $cnt = cnt - 1$                                  ▷ Decrease Self Request Count
2: $already = 0$
3: if $cnt > 0$ then                                ▷ If same node has further requests, directly enter CS
4:     $already = 1$
5:     Enter CS
6: else                                             ▷ Acknowledge all other pending requests
7:     $best$ = inf
8:     for all $R \in pending$ do
9:         send A($i$) to $R$.sender
10:        remove $R$ from $pending$
```

- **On receiving R(j, ts) from node j** (Whenever node i receives requests to enter CS)

```
1: if $already = 1$ or $best < ts$ then        ▷ If already in CS or request is low priority, defer ack
2:     Add R($j$, $ts$) to $pending$
3: else                                        ▷ Else send ack back
4:     send A($i$) to $j$
```

- **On receiving A(j) from node j** (Whenever node i receives acks)

```
1: $ack = ack + 1$
2: if $ack = n - 1$ then                       ▷ If received acks from all other nodes, enter CS
3:     $ack = 0$
4:     $already = 1$
5:     Enter CS
```

Best Case: When a node i is already in CS, and if it again wants to enter CS, then the node can do so without further permission once the CS becomes free. In this case, it requires 0 messages.

Worst Case: All other cases of entering CS has $2(n-1)$ message invocation.

Not Fair: Might lead to **starvation** as a single node might just keep entering and exiting the CS as long as it has sufficient requests.