Lamport’s Distributed mutual exclusion algorithm

Distributed Artificial Intelligence

CRISTIAN BELLUCCI

27/01/2025

Distributed Mutual Exclusion

- **Goal:**
  - given a set of distributed processes, only one for time must be able to access shared resources (Critical Section)

- Other goals:
  - Minimize message traffic
  - Minimize synchronization delay
    - i.e. no one has the lock and you ask for it, you should quickly get it

- **Challenges in Distributed system:**
  - Lack of global clock
  - Maintain causal consistency between events

Algorithm requirements

- **Safety:**
  - at most one process for time can access the CS
- **Liveness:**
  - Each process must be able to access the CS
- **Fairness:**
  - Bounded waiting time, FIFO order

Introduction to Lamport’s Mutual Exclusion

- **Node connectivity**
  - Fully connected: Every process communicates with every other process.
- **Key mechanism**
  - Client-Server interaction among processes
    - using TCP socket connection

Introduction to Lamport’s Mutual Exclusion

- Each processes:
  - exchange messages
    - **Request**
    - **Reply**
    - **Release**
  - maintaining:
    - its own local request queue (access CS in FIFO order)
    - Lamport logical clock as virtual timestamps

Lamport Algorithm

- **Requesting process Pi:**
  1. Pushing its request in its own queue
  2. Sending a request to every node
  3. Waiting for replies from all other node

  4. Enter the CS, if:
     a. its own request is at the head of its queue;
     b. all replies has been received;

  5. Exiting the CS :
     - Remove its request from the queue
     - Send a release message to other processes

| SOURCE |

Lamport Algorithm

- **Other processes Pj:**
  - After **receiving a request** form process Pi:
    - Push request (Pi clock, i) in its own queue:
    - Reply message to Pi
  - After **receiving release message**:
    - Remove corresponding request from its own queue

| SOURCE |

Lamport logical clock

- Ensure a consistent order of events.

- Rules for for each process:
    - Incrementing the clock before sending a message
    - Pi on receiving a message from Pj
        - ***LC(Pi) = max(LC(Pi), LC(Pj) + 1***

| **SOURCE** |

Managing the Request Queue

- Each process independently manages its queue.

- Request queue is a PriorityQueue (from python queue library)
  - Requests are sorted by **(LC, id)**
    - The head of the queue is always the request with **lowest LC**

Example Queue Update (Python)

- Eg. process sending request
    - In p.request_cs():

```
self.increment_clock()
current_req = (self.lamport_clock, self.my_id)
self.request_queue.put(current_req)
```

Example Queue Update (Python)

- Eg. process receiving request

```python
if msg_type == "REQUEST":
    self.request_queue.put((msg_clock, msg_sender))
    # auto-REPLY
    self.increment_clock()
    self.send_message(msg_sender, f"REPLY {self.lamport_clock} {self.my_id}\n")
```

Example Queue Update (Python)

- Eg. process receiving release

```python
elif msg_type == "RELEASE":
    if not self.request_queue.empty():
        head_req = self.request_queue.get()
        if head_req[1] != msg_sender:
            self.log(f"ERROR: top item {head_req} != {msg_sender} releasing.")
            #rimetti l'elemento in cima alla coda
            self.request_queue.put(head_req)
```

Overview of simulation

- SImulate a distributed system using:
  - ***multiprocessing library***
  - ***TCP sockets*** to exchange messages

Simulating the Distributed Environment

- Processes as Nodes:
  - Each node is a process with a unique ID.
  - Nodes communicate over TCP sockets
    - (e.g., port 5000 + node_id).
- TCP Communication:
  - Dedicated Class handles persistent TCP connections between nodes.

.

Manages Communication

- Class ***ConnectionPool:***
  - Manages persistent TCP connections to other nodes storing them in a dictionary.
  - Methods for:
    - Retrieve or establish a TCP connection to the specified node.
    - Send a message to using the connection to the target node.
    - Closing connections.

Lamport Algorithm Class

- The LamportNode class models the behaviour of a process in the system
    - Key Methods:
        - increment_clock - Increments the logical clock.
        - update_clock - Synchronizes with received clock values.
        - request_cs - Requests critical section access.
        - release_cs - Releases the critical section.
        - broadcast_message - Sends messages to all nodes.
        - broadcast_shutdown - Coordinates simulation termination.
        - run_server - Listens for incoming messages.
        - Handle_client - manage incoming message from client
        - handle_message - Processes received messages.
        - stop - Stops the server and connections.

Workflow

- N processes, each running as a node.
- Each node:
  - Starts a server to listen for messages.
  - it executes loop of form:
    - **while(true):**

      p.request(self.clock, self.id)

      #wait all replies

      p.sleep() #access CS and do some operation

      p.release(self.clock, self.id)

- Broadcasts shutdown messages upon completion.

Experiments

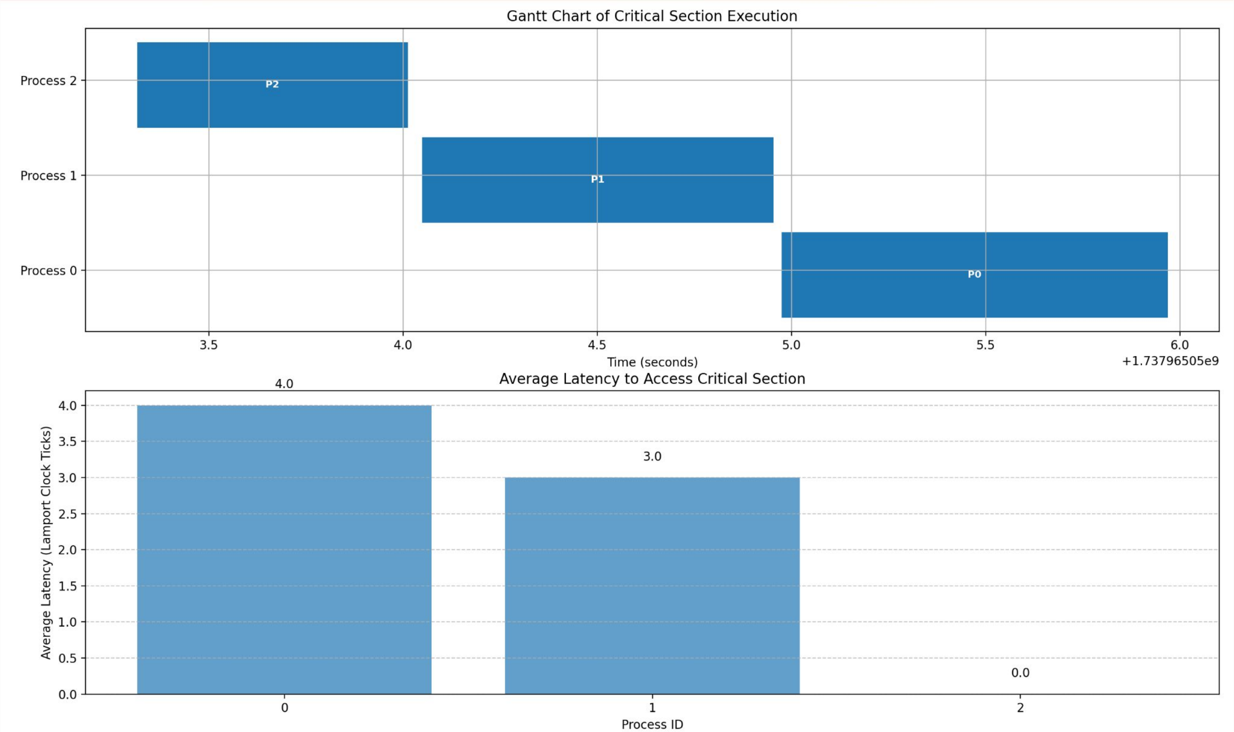
- Test the system increasing number of processes

Results

- N° processes = 3
- iteration = 1
-

```
--- Final Results ---
cs_history: [(2, 9), (1, 12), (0, 16)]
shared_counter: 3
message_count: {0: 6, 1: 6, 2: 6}
Total messages sent: 18
Execution time: 7.5975s
```

Results



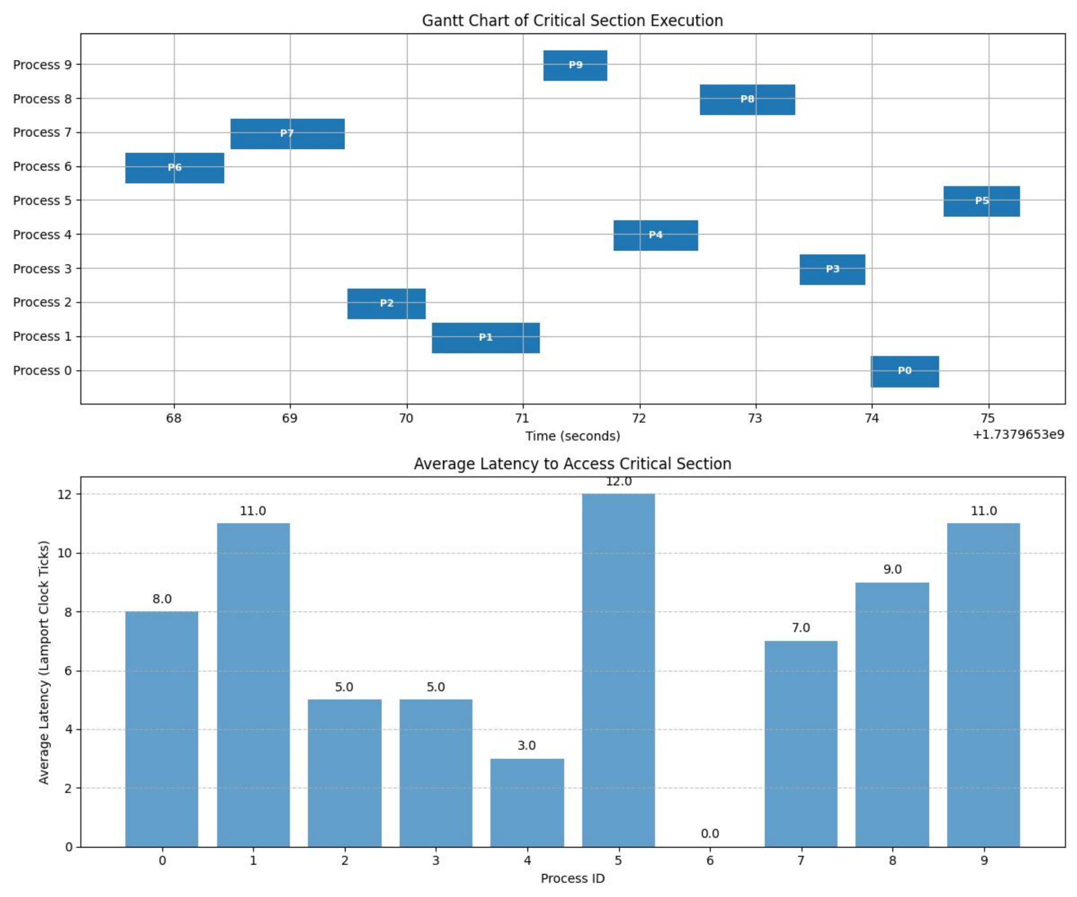
Gantt Chart of Critical Section Execution
Process 2
Process 1
Process 0
P2
P1
P0
3.5
4.0
4.5
5.0
5.5
6.0
Time (seconds)
+1.73796505e9
Average Latency to Access Critical Section
Average Latency (Lamport Clock Ticks)
4.0
3.0
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
1
2
Process ID

Results

- N° processes = 10
- iteration = 1

```
--- Final Results ---
cs_history: [(6, 30), (7, 37), (2, 42), (1, 53), (9, 64), (4, 67), (8, 76), (3, 81), (0, 89), (5, 101)]
shared_counter: 10
message_count: {0: 27, 1: 27, 2: 27, 3: 27, 4: 27, 5: 27, 6: 27, 7: 27, 8: 27, 9: 27}
Total messages sent: 270
```

Results



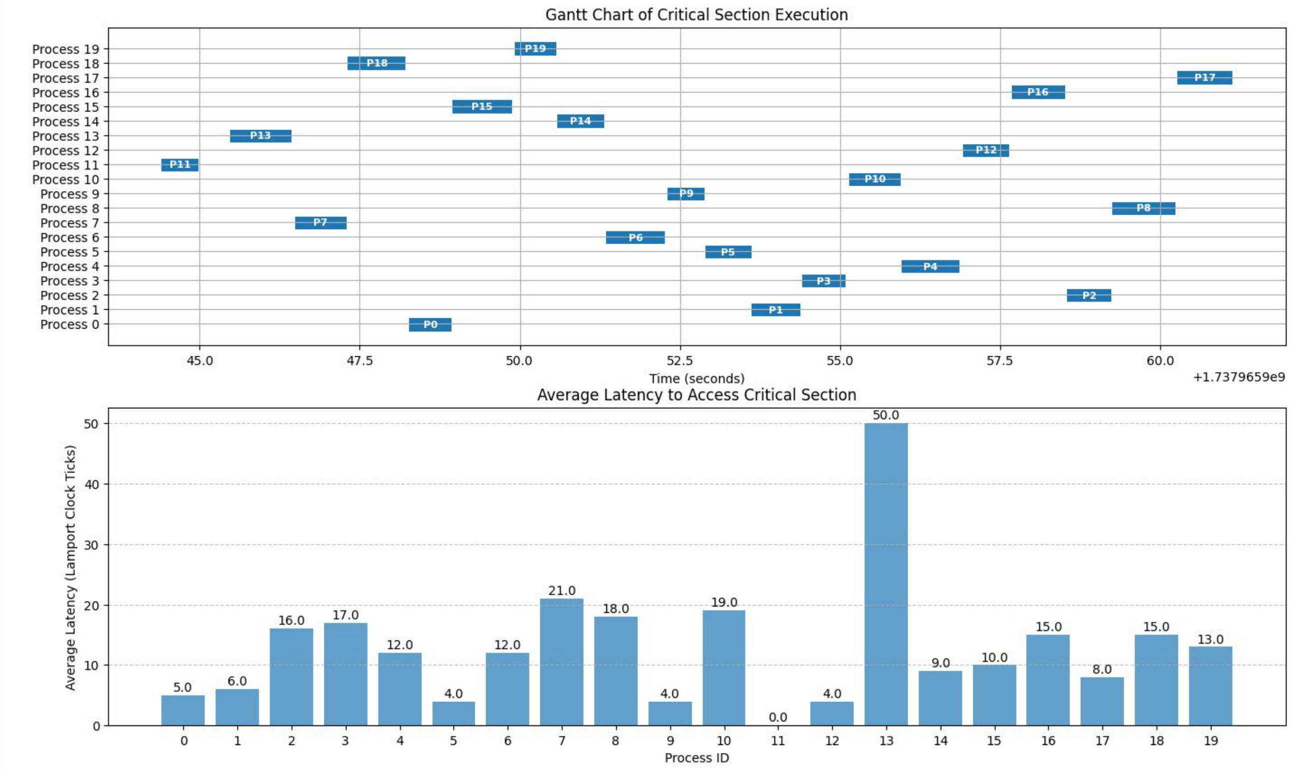
Gantt Chart of Critical Section Execution
Process 9
Process 8
Process 7
Process 6
Process 5
Process 4
Process 3
Process 2
Process 1
Process 0
P9
P8
P7
P6
P5
P4
P3
P2
P1
P0
68
69
70
71
72
73
74
75
Time (seconds)
+1.7379653e9
Average Latency to Access Critical Section
Average Latency (Lamport Clock Ticks)
0
2
4
6
8
10
12
8.0
11.0
5.0
5.0
3.0
12.0
0.0
7.0
9.0
11.0
0
1
2
3
4
5
6
7
8
9
Process ID

Results

- N° processes = 20
- iteration = 1

```
--- Final Results ---
cs_history: [(11, 22), (13, 72), (7, 93), (18, 108), (0, 113), (15, 123), (19, 136), (14, 145), (6, 157), (9, 161), (5, 16
5), (1, 171), (3, 188), (10, 207), (4, 219), (12, 223), (16, 238), (2, 254), (8, 272), (17, 280)]
shared_counter: 20
message_count: {0: 57, 1: 57, 2: 57, 3: 57, 4: 57, 5: 57, 6: 57, 7: 57, 8: 57, 9: 57, 10: 57, 11: 57, 12: 57, 13: 57, 14:
57, 15: 57, 16: 57, 17: 57, 18: 57, 19: 57}
Total messages sent: 1140
```

Results

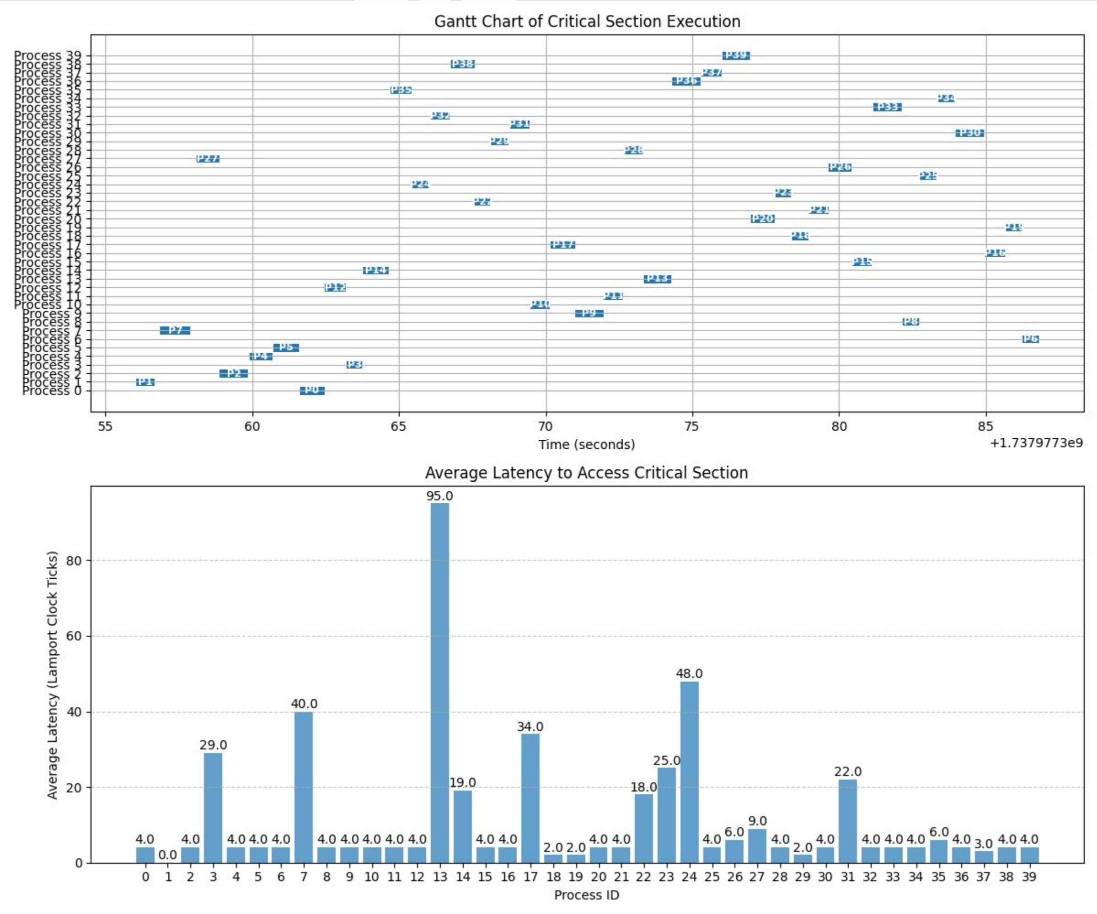

Gantt Chart of Critical Section Execution
Process 19
Process 18
Process 17
Process 16
Process 15
Process 14
Process 13
Process 12
Process 11
Process 10
Process 9
Process 8
Process 7
Process 6
Process 5
Process 4
Process 3
Process 2
Process 1
Process 0
P19
P18
P17
P16
P15
P14
P13
P12
P11
P10
P9
P8
P7
P6
P5
P4
P3
P2
P1
P0
45.0
47.5
50.0
52.5
55.0
57.5
60.0
Time (seconds)
+1.7379659e9
Average Latency to Access Critical Section
Average Latency (Lamport Clock Ticks)
0
10
20
30
40
50
5.0
6.0
16.0
17.0
12.0
4.0
12.0
21.0
18.0
4.0
19.0
0.0
4.0
50.0
9.0
10.0
15.0
8.0
15.0
13.0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Process ID

Results

- N° processes = 40
- iteration = 1

```
--- Final Results ---
cs_history: [(1, 182), (7, 222), (27, 231), (2, 235), (4, 239), (5, 243), (0, 247), (12, 251), (3, 280), (
14, 299), (35, 305), (24, 353), (32, 357), (38, 361), (22, 379), (29, 381), (31, 403), (10, 407), (17, 441
), (9, 445), (11, 449), (28, 453), (13, 548), (36, 552), (37, 555), (39, 559), (20, 563), (23, 588), (18,
590), (21, 594), (26, 600), (15, 604), (33, 608), (8, 612), (25, 616), (34, 620), (30, 624), (16, 628), (1
9, 630), (6, 634)]
shared_counter: 40
message_count: {0: 117, 1: 117, 2: 117, 3: 117, 4: 117, 5: 117, 6: 117, 7: 117, 8: 117, 9: 117, 10: 117, 1
1: 117, 12: 117, 13: 117, 14: 117, 15: 117, 16: 117, 17: 117, 18: 117, 19: 117, 20: 117, 21: 117, 22: 117,
 23: 117, 24: 117, 25: 117, 26: 117, 27: 117, 28: 117, 29: 117, 30: 117, 31: 117, 32: 117, 33: 117, 34: 11
7, 35: 117, 36: 117, 37: 117, 38: 117, 39: 117}
Total messages sent: 4680
Execution time: 45.1788s
```

Results



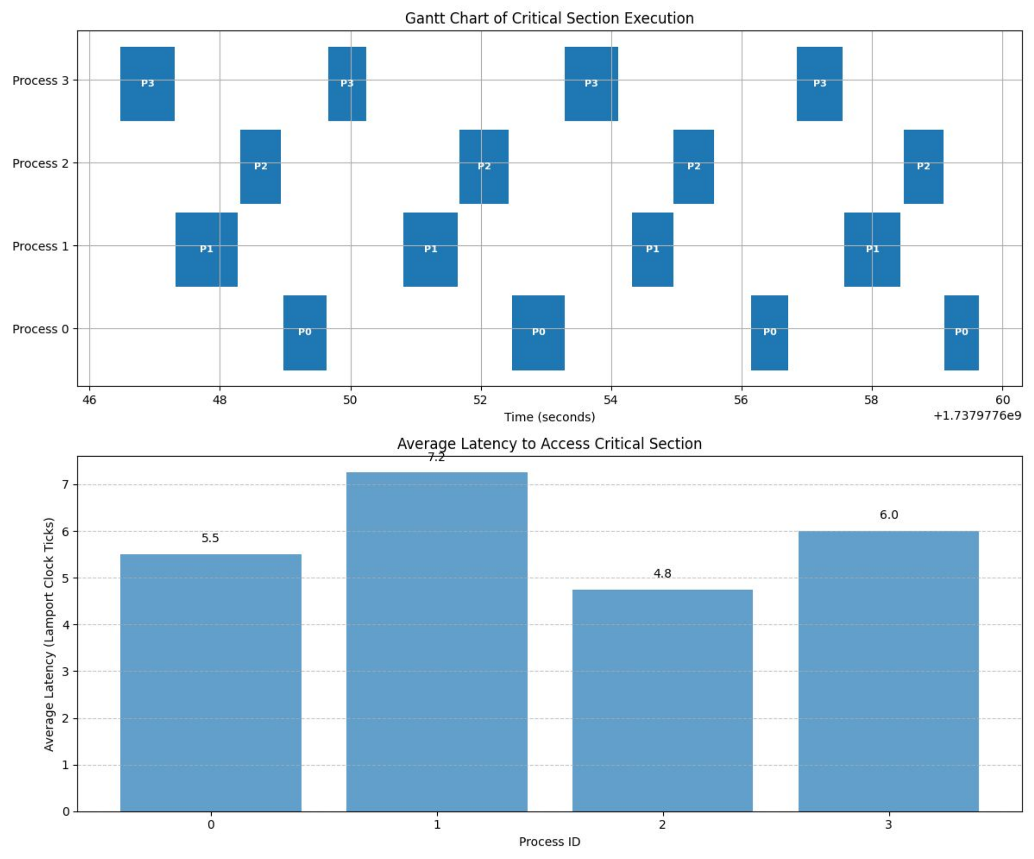
Gantt Chart of Critical Section Execution
Time (seconds)
+1.7379773e9
Average Latency to Access Critical Section
Average Latency (Lamport Clock Ticks)
Process ID

Results

- N° processes = 4
- iteration = 4

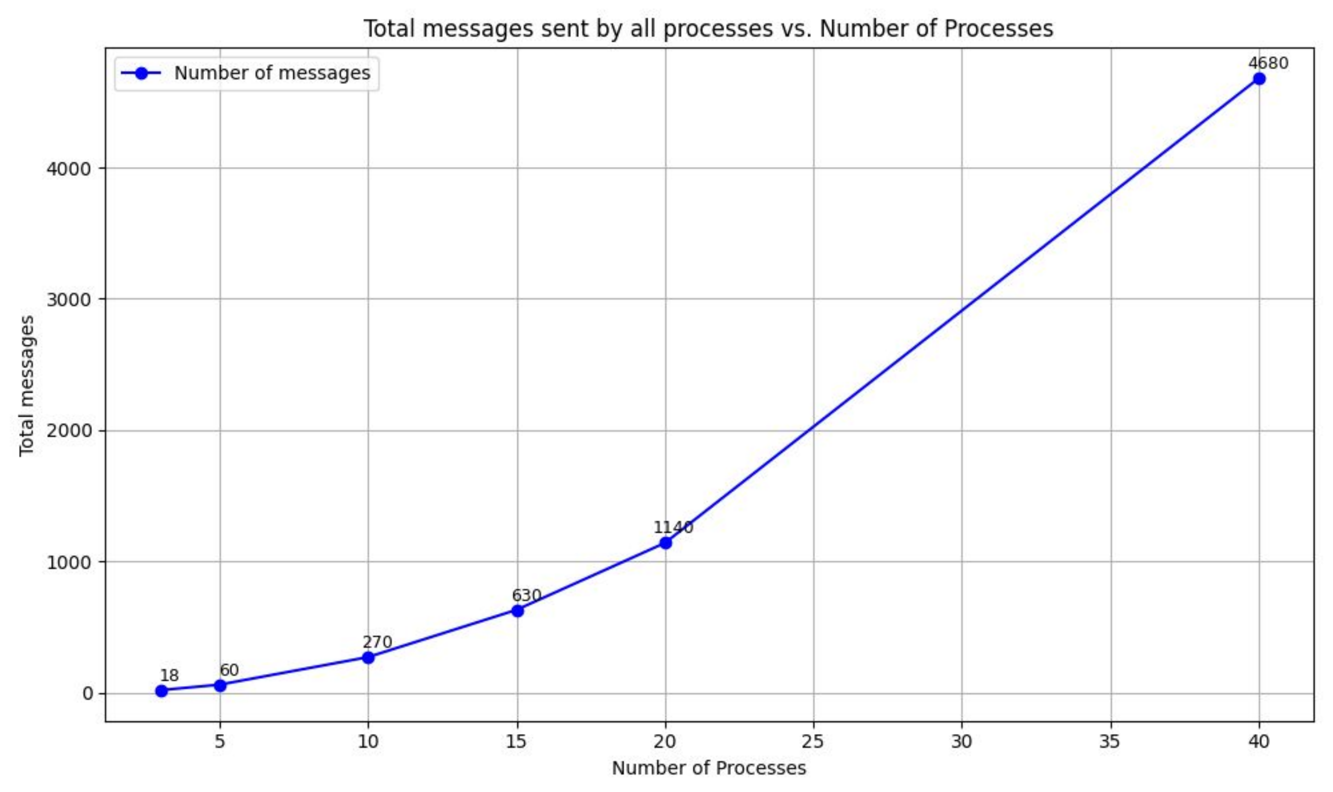
```
--- Final Results ---
cs_history: [(3, 12), (1, 16), (2, 20), (0, 25), (3, 29), (1, 39), (2, 44), (0, 50), (3, 54), (1, 64), (2,
 67), (0, 75), (3, 85), (1, 90), (2, 97), (0, 100)]
shared_counter: 16
message_count: {0: 36, 1: 36, 2: 36, 3: 36}
Total messages sent: 144
Execution time: 18.1081s
```

Results



Gantt Chart of Critical Section Execution
Process 3
Process 2
Process 1
Process 0
P3
P2
P1
P0
46
48
50
52
54
56
58
60
Time (seconds)
+1.7379776e9
Average Latency to Access Critical Section
Average Latency (Lamport Clock Ticks)
5.5
7.2
4.8
6.0
0
1
2
3
4
5
6
7
Process ID

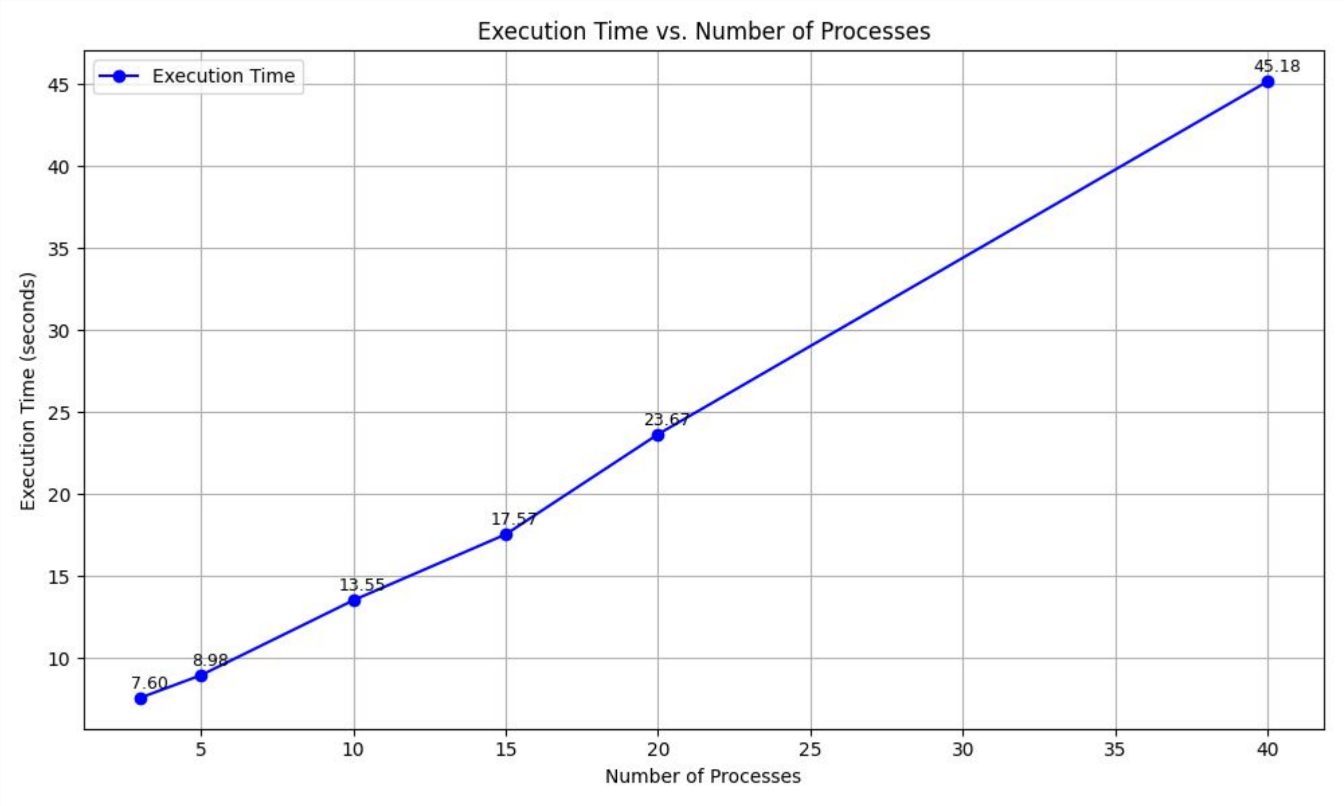
Total number of messages per iteration



*N_mex = N*3*(N-1)) = 3(N**2 - N)*

Where N is the number of Processes accessing the CS

Execution time



Advantages

- **Fully distributed mutual exclusion**
- **FIFO Order**: Ensures requests are processed in the order they were sent
- Guarantee:
  - Safety
  - Liveness
  - Fairness

Disadvantages

- **Complexity:**
  - ***3(N-1)***
- **No Fault Tolerance**:
  - there are n point of failure -> NOT ROBUST
- **Scalability Issues**:
  - Quadratic growth in connections makes it impractical for large systems.
- **Not adaptive:**
  - Number of processes is known a priori

Thanks for the attention