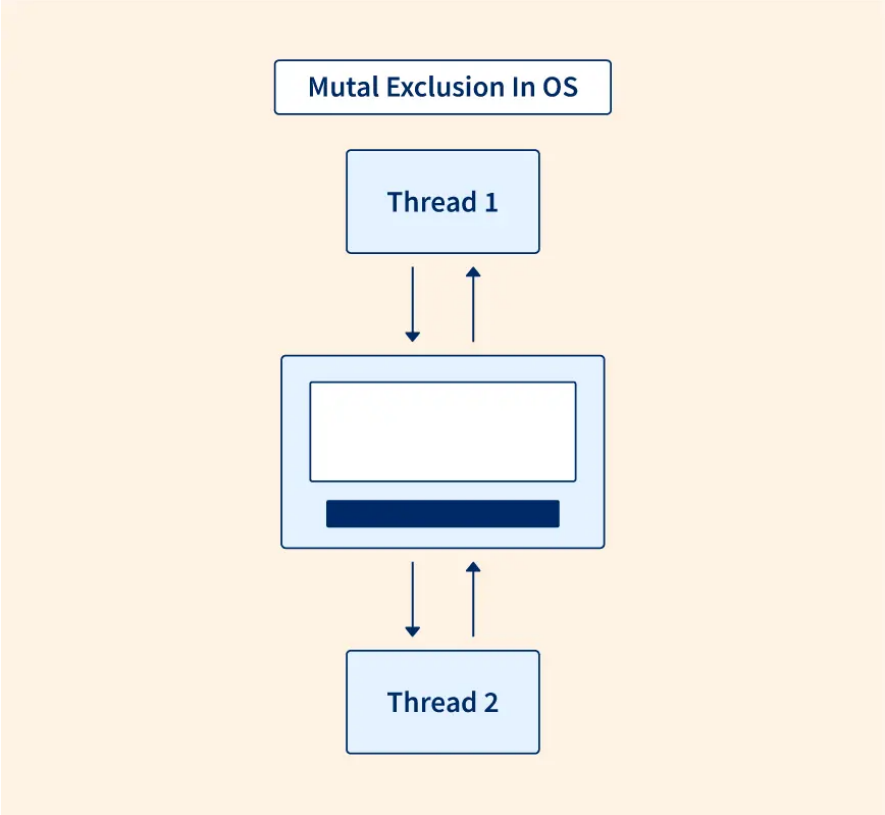
Distributed Mutual Exclusion Using Ricart-Agrawala's Algorithm

Mutual Exclusion

Mutual exclusion refers to a technique used in computer science to ensure that only one process or thread accesses a shared resource or critical section at a time. The goal of mutual exclusion is to prevent concurrent access to the shared resource, which could lead to data inconsistencies, race conditions, or other synchronization issues.

In other words, mutual exclusion ensures that when one process is accessing a shared resource, no other process can access that resource until the first process has completed its work and released the resource. This is typically accomplished through the use of locks, semaphores, or other synchronization primitives that provide exclusive access to the resource for a particular process or thread.



There are several techniques used for achieving mutual exclusion, including:

1. Locks and Semaphores: These are software-based mechanisms that allow only one process at a time to access a shared resource. Locks provide exclusive access to a resource, while semaphores can allow multiple processes to access the resource with certain restrictions.

2. Atomic Operations: These are low-level operations that can be performed without interruption, ensuring mutual exclusion. For example, in multi-threaded programming, incrementing or decrementing a variable can be performed atomically using specific instructions provided by the processor.
3. Monitors: These are high-level language constructs that encapsulate shared resources and provide a set of methods for accessing them. Only one process at a time can execute a method on the monitor, ensuring mutual exclusion.
4. Message Passing: This technique is used in distributed systems where processes communicate with each other by exchanging messages. Mutual exclusion is achieved by ensuring that only one process can access a shared resource at a time.

Algorithms other than Ricart Agrawala's

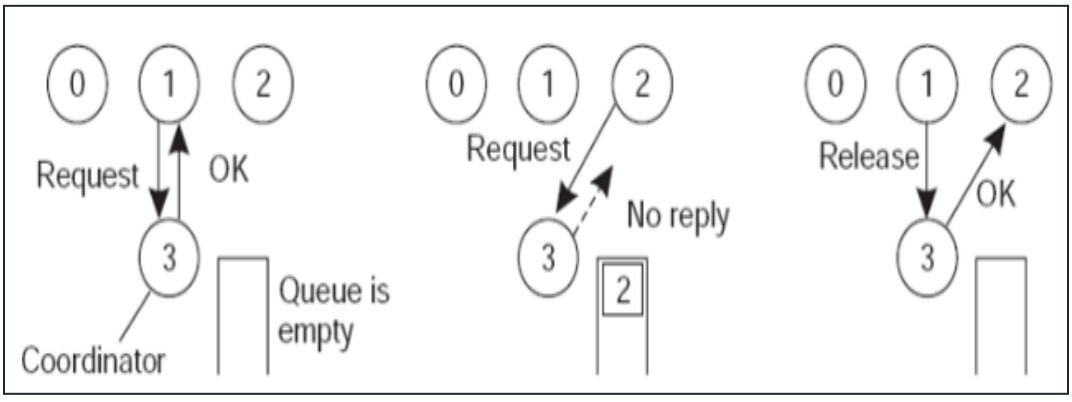
There are several algorithms for achieving mutual exclusion in a distributed system that are centralized and token-based. Some of these include:

1. Lamport's Distributed Mutual Exclusion Algorithm - This algorithm uses a central server to maintain a queue of requests for accessing the critical section. Each node sends a request to the server and waits for a reply before entering the critical section.

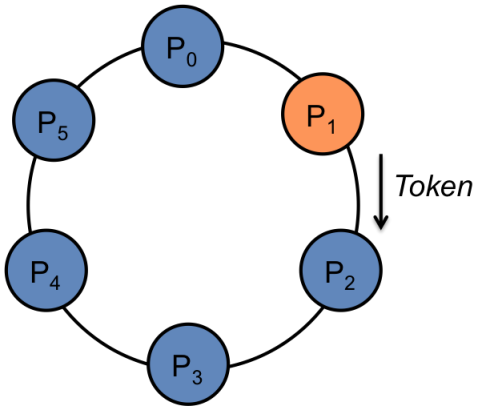
2. Suzuki-Kasami's Mutual Exclusion Algorithm - This algorithm uses a token that is passed around the network to grant access to the critical section. Nodes can only enter the critical section if they possess the token.

3. Maekawa's Algorithm - This algorithm divides the nodes in the network into groups and assigns a token to each group. A node can enter the critical section if it has the token for its group and there are no other nodes from its group in the critical section.

All of these algorithms use a **centralized approach** and a **token-based mechanism** to achieve mutual exclusion in a distributed system.



Centralized approach for achieving mutual exclusion



Token based mechanism, to achieve mutual exclusion

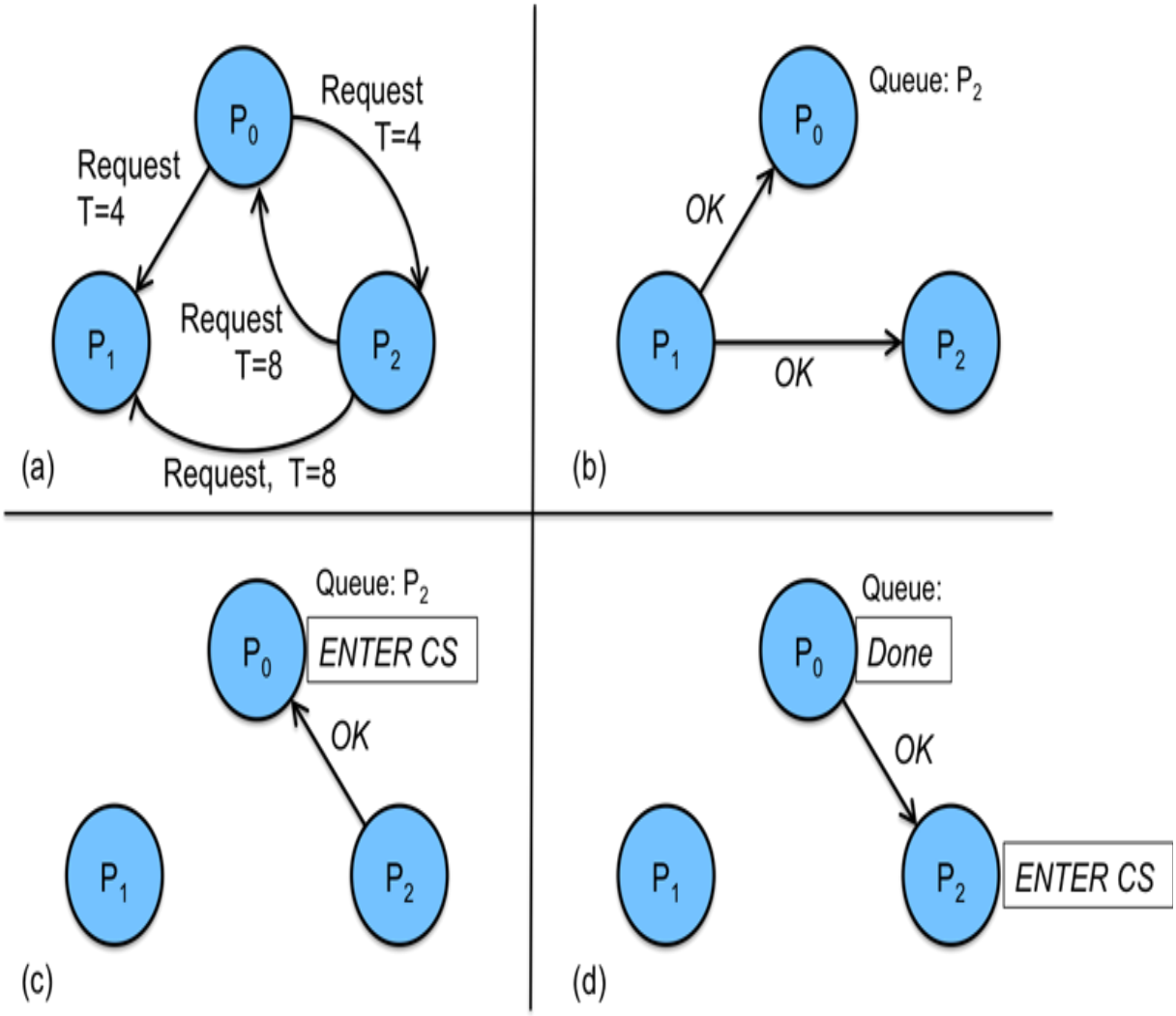
The Ricart-Agrawala algorithm is different from centralized and token-based algorithms in the sense that it is a distributed algorithm and does not rely on a centralized node or token for mutual exclusion.
Centralized and token-based algorithms require a centralized node or a token that is passed around to control access to the critical section.
In contrast, the Ricart-Agrawala algorithm uses a distributed approach where each node communicates with other nodes to request access to the critical section. This approach allows for more flexibility and scalability in distributed systems where a centralized node or token may not be feasible.

The Ricart-Agrawala algorithm

The Ricart-Agrawala algorithm is a distributed mutual exclusion algorithm used in distributed computing systems to ensure that concurrent processes do not interfere with each other while accessing a shared resource. The algorithm allows a set of distributed processes to request and release critical sections of code, ensuring that only one process has access to the critical section at any given time.

The algorithm is based on the concept of **timestamps**. Each process maintains a logical clock that allows it to generate unique timestamps for each request it makes to enter the critical section. A process that wants to enter the critical section sends a request message to all other processes in the system, including its own process. The request message contains the timestamp generated by the requesting process.

When a process receives a request message from another process, it **compares the timestamp in the request message to the timestamp** of its own last request. If the requesting process has a higher timestamp than the receiving process, the receiving process defers its own request and replies to the requesting process with a message containing its own timestamp. If the receiving process has a higher or equal timestamp, it queues the requesting process and waits until it has completed its critical section.



Once a process completes its critical section, it sends a release message to all other processes in the system, indicating that it has finished using the shared resource. The release message also contains the process's timestamp.

When a process receives a release message from another process, it removes the releasing process from its queue and **checks whether** there are **any queued processes waiting** to enter the critical section. If there are, the process with the highest timestamp is granted access to the critical section.

The Ricart-Agrawala algorithm is **completely decentralized** and does not require a central coordinator. It is also **highly fault-tolerant**, as it can handle network partitions and process failures without affecting the correctness of the algorithm. However, it can suffer from high message overhead, especially in systems with many processes or high contention for the critical section.

Advantages of Ricart Agrawala algorithm

The algorithm that has several advantages over earlier algorithms such as the Centralized and Token Ring-based algorithms. One of the significant advantages is that it does not rely on a centralized server, making it more fault-tolerant and scalable. This means that the algorithm can continue to function even if some nodes in the network fail, and the algorithm can be used in larger networks without experiencing performance issues.

Another advantage is that the Ricart-Agrawala algorithm does not require a token or a quorum, which can reduce network traffic and improve efficiency. In a token-based algorithm, the token is passed around between nodes, which can lead to unnecessary network traffic. In contrast, the Ricart-Agrawala algorithm allows each node to request access to the critical section independently without requiring a token or quorum, reducing network traffic.

Finally, the Ricart-Agrawala algorithm uses a queue-based approach that prioritizes requests based on their timestamps, which can improve fairness and reduce the possibility of starvation. This means that requests for the critical section are granted based on their timestamp, which ensures that all nodes eventually get a chance to access the critical section. In earlier algorithms, a node could be starved of access to the critical section if other nodes kept requesting access, but the queue-based approach of the Ricart-Agrawala algorithm ensures that each node is granted access to the critical section eventually.

Implementations aspects

We had to Implement the Ricart-Agarwala's mutual exclusion protocol that should handle
i. Handle node failures
ii. Record state of different nodes
iii. Add new node to system

The solution design

The overall solution and implementation of the requirements of a distributed system for achieving mutual exclusion using the Ricart-Agrawala algorithm.

1. The system consists of N nodes, where each node is identified with a unique number ranging from 0 to N-1. These nodes are distributed and communicate with each other using TCP socket communication.
2. Each node in the system has a Server and Client Component built into it. Upon starting, each node activates its server component, which begins listening for messages from other nodes. Server components are created in N-1 threads to receive messages concurrently from other nodes.
3. To enter its critical section, a node connects to all other nodes in the system using the Client component and sends a REQUEST message. After receiving the REPLY

message from all other nodes, the node enters the critical section to execute its critical code. If there are any deferred messages, they are processed while exiting the critical section. This algorithm ensures that only one node executes its critical code at a time, providing mutual exclusion in the distributed system.

Implementation Details

Script
The Python script that starts a server and multiple processes to simulate mutual exclusion in a distributed system.

Initially we have **`num_simulations`, `server_port`, and `initial_port`**, which are used to configure the simulation. It prompts the user to input the number of nodes to start with. This is feeded into the server node for facilitating the algorithm.

The script creates an empty file "**config.txt**". Which is used to store the to store the IP address and port number of other nodes in the network.

Server Node is started with the args as {server_port} {num_proc} {num_simulations}` in a The server listens on `server_port` for incoming connections from nodes and the `num_proc` variable specifies the number of processes to start with.

After starting the server, the script iteratively starts new process nodes. The function takes the `self_id` of the node and **increments the `ports_begin`** variable, which is used to assign a unique port to each process node. & the 'num_proc' number of processes are started.

The script also keeps track of the `process_id_port` dictionary, which maps a process ID to the port number that it is running on.

There's also an option to add further nodes, and the user is given an option to choose if they want to add more process nodes or not.

Server node
We need a server node here to manage node failure and to add process nodes. It facilitates communication among the process nodes, incase of node failures and further addition of nodes, so that the processes work smoothly even in case of some faults.

The system has three major components: a **heartbeat server**, a **live checker**, and a **main server**. The system architecture is a peer-to-peer network, where all nodes communicate with each other.

The **heartbeat server**. It initializes a socket and binds it to a port. The server listens for incoming messages and updates the `beatDict` dictionary when it receives a heartbeat message. To keep track of if a process is alive or not.The server also prints a message indicating that it received a heartbeat message.

The **live checker** initializes a socket and binds it to a port. The function checks the `beatDict` dictionary for processes that have sent a heartbeat message in the past 10 seconds. The function **sends a dictionary of alive processes** to the main server. Also prints a message indicating that it received a "send alive" message.

The **server** listens for incoming messages from other processes. It initializes a socket and binds it to a port. The server listens for incoming messages and sends a dictionary of other nodes in the network to the requesting process. The function uses the `**config.txt**` file to get the IP address and port number of other nodes in the network. The server sends the dictionary as a JSON-encoded string.

`num_proc` is the number of processes, **`server_port` is the port number** for the main server, and **`num_simulations` is the number of simulation** runs. **`port_seed`** is the initial port number used to communicate with other nodes, and `MAX_CAPACITY` is the maximum buffer size for sending and receiving data.

```
server_port = int(sys.argv[1])
num_proc = int(sys.argv[2])
num_simulations = int(sys.argv[3])
```

The code defines a global dictionary `beatDict` and a lock `dictLock`. `**beatDict**` stores the heartbeat messages(explained further) from each process, and `dictLock` ensures that the dictionary is accessed by only one thread at a time. There are three functions that manipulate the dictionary: `update`, `extractSilent`, and `extractAlive`.

`update(remote_message)` is called whenever the heartbeat server receives a message from a process. The function updates the `beatDict` dictionary with the new message.

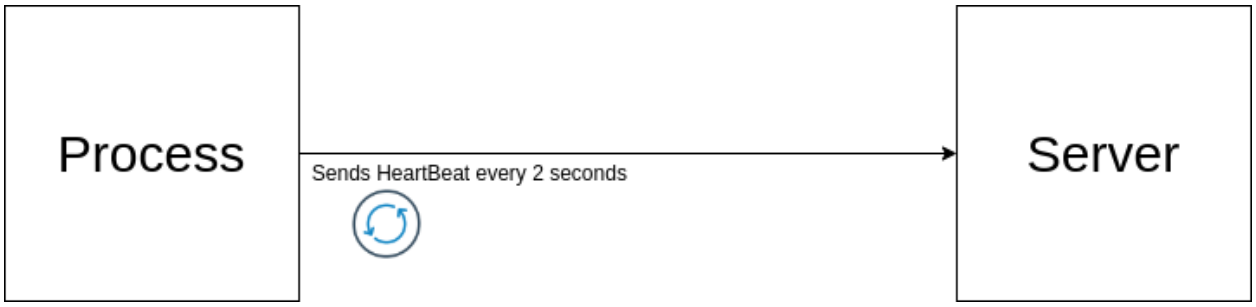
`extractSilent(howPast)` returns a list of processes that have not sent a heartbeat message in the past `howPast` seconds. The function iterates over `beatDict` and checks each entry's timestamp. If the entry's timestamp is older than `howPast`, it is added to the `silent` list.

`extractAlive(howPast)` returns a dictionary of processes that have sent a heartbeat message in the past `howPast` seconds. The function iterates over `beatDict` and checks each entry's timestamp. If the entry's timestamp is newer than or equal to `howPast`, it is added to the `silent` dictionary.

Process

Several functionalities for creating a distributed system that uses Ricart-Agrawala Algorithm for mutual exclusion among the nodes of the system. Here are the details of the different parts of this section:

Heartbeat Sender Thread
This is a separate thread that sends a heartbeat message to the main server every 2 seconds. The heartbeat message contains the following information:
- `process_id`: It represents the ID of the current node.
- `port`: It represents the port number on which the current node is listening for incoming requests.
- `msg_type`: It represents the type of message, which is "ZINDA" in this case.
- `time_stamp`: It represents the timestamp at which the message is sent.



Thread for Accepting Connections
This thread listens for incoming requests on the port number `self_port`. When a request is received, it logs the event to a file named `log.txt`. If the received data is empty, it releases the lock and breaks the loop. Otherwise, it reverses the received data and sends it back to the client.

Thread for Request
The `thread_for_request(server_port)` function is used for establishing a connection with the main server and requesting the list of peers in the system. Once the list is received, it establishes connections with the peers and sends a request to each peer. The request contains the following information:
- `process_id`: It represents the ID of the current node.

- `port`: It represents the port number on which the current node is listening for incoming requests.

Start Process Communication
To start the communication between the nodes. It creates a socket and binds it to the port number `self_port`. It then logs the ID and port number of the current node to a file named `**config.txt**`. Next, it starts a separate thread for requesting the list of peers from the main server.

Then, it runs a loop for `**num_simulations**` iterations, where **each iteration simulates a critical section**. In each iteration, it sends a request to all the peers using the Ricart-Agrawala Algorithm for mutual exclusion. Once the lock is acquired, it **sleeps for a certain interval of time based on its process ID**, releases the lock using, and then sleeps for a fixed interval of 10 seconds.

The loop then increments the simulation count i_sim_count by 1, and repeats the above process until the number of simulations is reached. After completing all the simulations, it closes the socket object s.

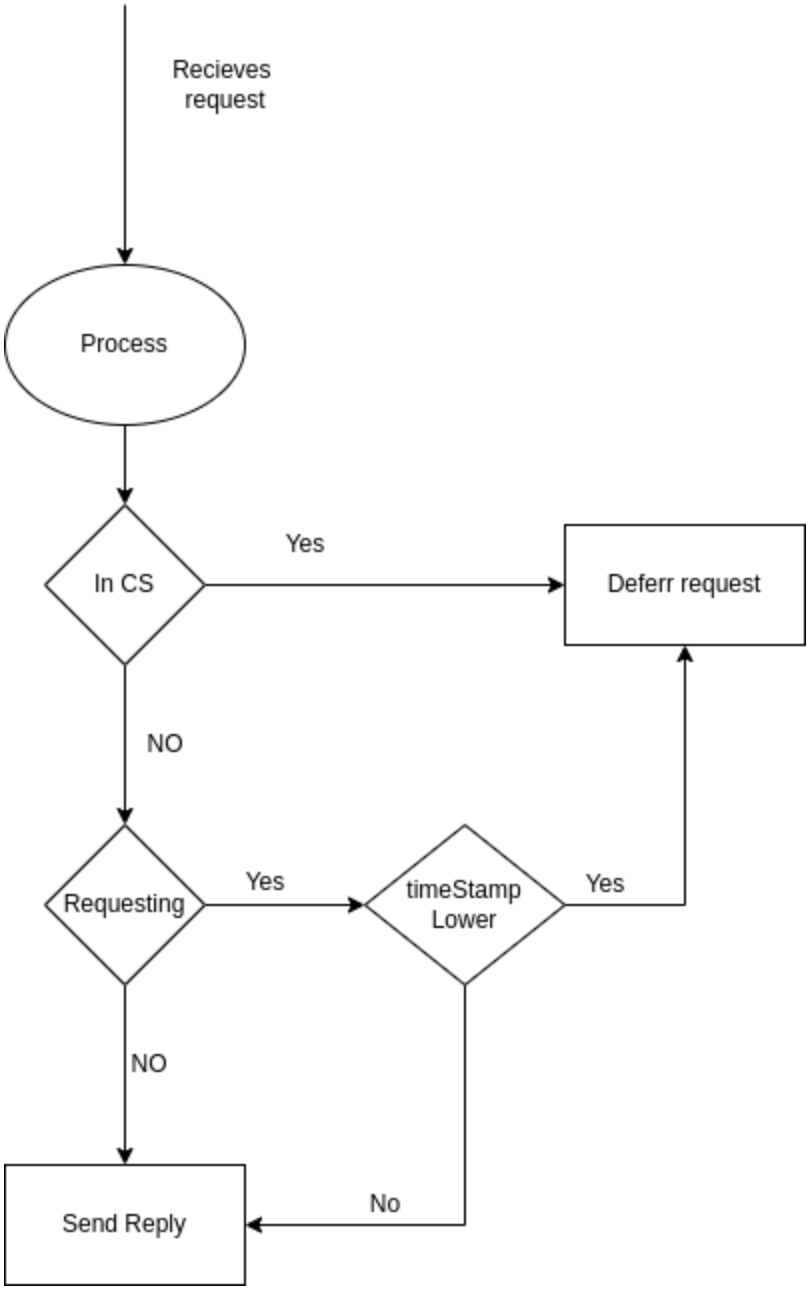
Overall, this code implements a distributed system using the Ricart-Agrawala algorithm for mutual exclusion, where each process communicates with the main server to obtain the details of other processes, and acquires the mutex using the Ricart-Agrawala algorithm.

RA_Mutex
It has some major components, **Lock_Mutex, Message_Handler, Release_Mutex, Initilize_Mutex,** also an important dictionary with the name as map_state. This is used by the processes to communicate with each other and maintain the mutual exclusion as well as implementing the mutual exclusion as in the Ricart-Agrawala algorithm.

A given process has a couple of choices, at a given time, with regard to entering into the critical section as **Wanted, Requested, Released, Held.**

The process nodes, while in execution, call the various components from this section as per their requirements, either to lock the mutex, release the mutex, or initialize the mutex. And for communicating in between the messages.

Message handler
It processes incoming messages for a distributed mutual exclusion algorithm. It takes a single argument `message` which is expected to be a JSON string representing a message from the process



The message type is checked. If the message type is `**REQUEST**`, the function **checks if** the current process is in the **critical section** or if it has already requested the resource **and the requesting process has a higher timestamp**. **If either of these conditions are true**, the current process **defers** the request by adding it to a dictionary called with the requesting process ID as the key and the requesting process address as the value.

If the current process is not holding the resource and has not already requested it **or the requesting process has a lower timestamp**, the current process sends a `REPLY` message

to the requesting process. The message contains the type `REPLY` and the `procInfo` dictionary of the current process. The `send_message` function is used to send this message to the requesting process.

If the message type is `REPLY`, then it means that it got a reply from a process from which it was expecting the reply message and it then removes the corresponding process ID from the `reply_pending_queue`.

Three more significant functions, **initialize_mutex(), lock_mutex() and release_mutex()**, work together to provide mutual exclusion to the critical section of a distributed system.

The **initialize_mutex**() takes the local address of the process, its process ID (PID), process name, remote addresses, number of remote processes, and a self socket as inputs. It sets the initial state of the process and starts a new thread to listen for incoming messages from other processes.

The **lock_mutex**() is used when a process wants to enter the critical section. It sets the process state to WANTED and sends a request message to all alive processes, except itself, to enter the critical section. It waits until it receives a reply message from all processes in the map_state. Once it receives the reply from all processes, it enters the critical section.

The **release_mutex**() is called when a process leaves the critical section. It sets the process state to **RELEASED** and sends a reply message to all processes in the deferred_requests_queue. It then clears the deferred_requests_queue.

```
def release_mutex():
    #print(f"{current_process_info['procPID']}In release_mutex\n")
    current_process_info['procState'] = RELEASED
    print(f"{Style.BRIGHT} {Fore.YELLOW} Exited CS {current_process_info['procPID']} {Style.RESET_ALL}\n\n")

    replyMessage = {"type": REPLY, "procInfo": current_process_info}


    #print(f"{reply_pending_queue=}")
    #print(f"{deferred_requests_queue=}")

    copy_def_q = deferred_requests_queue.copy()
    for key in list(copy_def_q):

        if int(key) in map_state:
            print(f"{Fore.GREEN}Sending deferred reply message from {current_process_info['procPID']} to {key} {Fore.RESET}\n\n")

            send_message(copy_def_q[key], json.dumps(replyMessage))
            #print(f"{deferred_requests_queue=}")
        deferred_requests_queue.pop(key)

    #print(f"{current_process_info['procPID']}Exit CS")
    return True
```

The algorithm works as follows:
When a **process wants to enter the critical section**, it first sets its state to **WANTED** and sends a request message to all other processes. The message includes the process information, such as its ID, current state, and timestamp. The timestamp is used to resolve ties in the case of simultaneous requests from multiple processes.

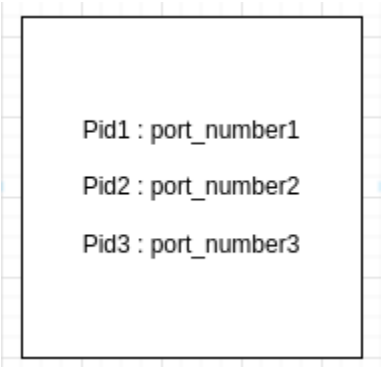
When a process receives a request message, it checks if it is currently holding the mutex. If it is not holding the mutex, it sends a reply message to the requesting process, depending upon the certain condition explained before. If it is holding the mutex, it defers the request by adding it to a deferred_requests_queue.

When a process receives a reply message, it checks if it has received replies from all alive processes in the **map_state**. If it has, it enters the critical section. Otherwise, it waits until it receives replies from all processes in the map_state.

The code also defines various status codes, such as REPLY, REQUEST, WANTED, HELD, and RELEASED, which are used to represent the state of the process and the type of messages being sent between processes.

Overall, this code implements a distributed mutual exclusion algorithm that allows processes to enter and leave a critical section in a coordinated way, ensuring that only one process is in the critical section at any given time.

Map state
This is a dictionary which stores the currently alive processes, stored as, {process_id,port_numer}



The update_map_state() function is used to update the map_state global variable by querying a separate process that is responsible for monitoring the liveness of all other processes in the system.

The function decodes the JSON object into a dictionary temp, which it then uses to update the temp_map dictionary.

Overall the map state dictionary stores the information of alive processes, which is used by the processes for checking the current alive processes while they are expecting the reply message and if one of such expected nodes fails, still the process waiting will no longer expect the reply from the not alive process.