

# Distributed System Project Report

## Suzuki-Kasami Mutual Exclusion Implementation

Utkarsh Upadhyay (2019101010)

Ayush Sharma (2019101004)

## Project Goal

- Implement Suzuki-Kasami's token-based mutual exclusion protocol for Distributed Systems using MPI4PY.
- Record and display the status of different nodes on CLI (user has the option to print the (LN or RN) array and queue state values when he queries it).

## Algorithm and Data Structure

**Suzuki-Kasami algorithm** is a token-based algorithm for achieving mutual exclusion in distributed systems. In token-based algorithms, A site is allowed to enter its critical section if it possesses a unique token. If a process wants to enter its critical section and it does not have the token, it broadcasts a request message to all other processes in the system. The process that has the token, if it is not currently in a critical section, will then send the token to the requesting process. In token-based algorithms, the sequence number is used. Each request for a critical section contains a sequence number. This sequence number is used to distinguish between old and current requests.

The data structure used is:

- **An array of integers  $RN[1...N]$ :** A site  $S_i$  keeps  $RN_i[1...N]$ , where  $RN_i[j]$  is the largest sequence number received so far through the REQUEST message from site  $S_j$ .
- **An array of integer  $LN[1...N]$ :** This array is used by the token.  $LN[J]$  is the sequence number of the request that is recently executed by site  $S_j$ .
- **A queue  $Q$ :** This data structure is used by the token to keep a record of the ID of sites waiting for the token

## Code Execution

Execute code by following cmd:

```
mpiexec -np `X` python3 suzuki-kasami.py `dirc`
```

For ``X`` number of process and pre-existing ``dirc`` directory for logging files.

Message complexity of code :  $(X - 1)$  request messages + 1 reply message  $\rightarrow O(X)$

# Implementation Detail

All the core code related to Suzuki-Kasami has been implemented in the file **suzuki-kasami.py** file. To run the code following modules are required:

- Threading
- Collections
- MPI4PY and Numpy

The following steps have been considered while implementing the code:-

1. To simulate several nodes in a distributed system MPI process has been used in which every process can contact another by message passing. To simulate the critical section, the process goes to sleep when entered in its critical section.
2. A process has the following data structure available:

```
DATA_STRUCTURE = {  
    "token_queue": deque(),  
    "LN": np.zeros(total_nodes),  
    "RN": np.zeros(total_nodes),  
    "has_token" : False,  
    "in_cs": False,  
    "waiting_for_token": False  
}
```

3. The following locks were maintained to maintain the mutual exclusion of threads in a process/node:

```
LOCKS = {  
    "request": threading.Lock(),  
    "send": threading.Lock(),  
    "release": threading.Lock(),  
    "token": threading.Lock(),  
    "cs": threading.Lock(),  
    "rn": threading.Lock(),  
}
```

4. A process has in total two threads: (i) Simulator/Action thread and (ii) Listener thread. However, for printing status on the CLI for a given process rank, we added another thread for process with rank "0" that serves the purpose of interacting with user.

```

if __name__ == "__main__":
    directory = sys.argv[1]
    DATA_STRUCTURE["RN"][0]=1
    # giving a token to start the process 0
    if rank==0:
        print_log("%s: I am %d and have a startup token." % (datetime.now().strftime('%M:%S'), rank))
        sys.stdout.flush()
        DATA_STRUCTURE["has_token"]=True

    try:
        listener_thread = Thread(target=listener)
        listener_thread.start()
    except:
        print_log("Error: unable to start thread!  ")

    if rank==0:
        try:
            print_status_thread = Thread(target=print_node_status)
            print_status_thread.start()
        except:
            print_log("Error: unable to start print status thread!  ")

    while True:
        if not DATA_STRUCTURE["has_token"]:
            sleep(random.uniform(1,3));
            request()
        elif not DATA_STRUCTURE["in_cs"]:
            run_cs()
        while DATA_STRUCTURE["waiting_for_token"]:
            sleep(0.5)

```

5. **Simulator/Active thread:** Actually, simulator thread keep running in infinite loop. When a site  $S_i$  is running and don't have token, it firstly sleeps for 2-3 seconds then wants to enter the critical section but since it does not have the token it increments its sequence number  $RN_i[i]$  and sends a request message  $REQUEST(i, seq)$ , by calling function **request()**, to all other sites in order to request the token. Here seq is update value of  $RN_i[i]$ . In case, site  $S_i$  has token and is not executing it's critical section then **run\_cs()** function will get executed. Otherwise, it'll wait for requested token.

```

def request():
    global DATA_STRUCTURE
    with LOCKS["request"]:
        if not DATA_STRUCTURE["has_token"]:
            DATA_STRUCTURE["RN"][rank] = DATA_STRUCTURE["RN"][rank] + 1
            print_log("%s: I'm %d and want a token for the %d time." \
                    % (datetime.now().strftime('%M:%S'), rank, DATA_STRUCTURE["RN"][rank]))
            sys.stdout.flush()
            DATA_STRUCTURE["waiting_for_token"]=True
            for i in range(total_nodes):
                if rank!=i:
                    comm.send([
                        REQUEST_MESSAGE_TYPE,
                        rank,
                        DATA_STRUCTURE["RN"][rank]
                    ], dest=i)

```

```

def listener():
    global DATA_STRUCTURE
    while True:
        mes = comm.recv(source=MPI.ANY_SOURCE)
        if mes[0] == REQUEST_MESSAGE_TYPE:
            with LOCKS["rn"]:
                rid = mes[1]
                seq = mes[2]
                DATA_STRUCTURE['RN'][rid] = max([seq, DATA_STRUCTURE['RN'][rid]])
                if DATA_STRUCTURE['RN'][rid] > seq:
                    print_log(
                        "%s: Request from %d has expired." % (datetime.now().strftime('%M:%S'), rid))
                    sys.stdout.flush()
                if DATA_STRUCTURE['has_token'] and \
                    not DATA_STRUCTURE['in_cs'] and \
                    DATA_STRUCTURE['RN'][rid] == DATA_STRUCTURE['LN'][rid]+1:
                    DATA_STRUCTURE['has_token'] = False
                    send_token(rid)
        elif mes[0] == TOKEN_MESSAGE_TYPE:
            with LOCKS['token']:
                print_log("%s: I'm %d and I got a token." % (datetime.now().strftime('%M:%S'), rank))
                sys.stdout.flush()
                DATA_STRUCTURE['has_token'] = True
                DATA_STRUCTURE['waiting_for_token'] = False
                DATA_STRUCTURE['LN'] = mes[1]
                DATA_STRUCTURE['token_queue'] = mes[2]
                run_cs()

```

6. **Listener thread:** Listens to two types of messages

- a. **REQUEST\_MESSAGE\_TYPE** — When a site  $S_j$  receives the request message  $REQUEST(i, seq)$  from site  $S_i$ , it sets  $RN_j[i]$  to maximum of  $RN_j[i]$  and  $seq$  i.e  $RN_j[i] = \max(RN_j[i], seq)$ . After updating  $RN_j[i]$ , Site  $S_j$  sends the token to site  $S_i$  if it has token and  $RN_j[i] = LN[i] + 1$  by function **send\_token()**.

```

def send_token(recipient):
    global DATA_STRUCTURE
    with LOCKS["send"]:
        print_log("%s: I'm %d and sending the token to %d." \
            % (datetime.now().strftime('%M:%S'), rank, recipient))
        sys.stdout.flush()
        comm.send([
            TOKEN_MESSAGE_TYPE,
            DATA_STRUCTURE["LN"],
            DATA_STRUCTURE["token_queue"]
        ], dest=recipient)

```

- b. **TOKEN\_MESSAGE\_TYPE** — Site  $S_i$  executes the critical section if it has acquired the token. And execute function **run\_cs()**.

```
def run_cs():
    global DATA_STRUCTURE
    with LOCKS["cs"]:
        if DATA_STRUCTURE["has_token"]:
            DATA_STRUCTURE["in_cs"] = True
            print_log("%s: I am %d and execute %d CS." \
                    % (datetime.now().strftime('%M:%S'), rank, DATA_STRUCTURE['RN'][rank]))
            sys.stdout.flush()
            sleep(random.uniform(2, 5))
            DATA_STRUCTURE["in_cs"] = False
            print_log("%s: I'm %d and finished %d CS." \
                    % (datetime.now().strftime('%M:%S'), rank, DATA_STRUCTURE['RN'][rank]))
            sys.stdout.flush()
            release_cs()
```

7. **Releasing the critical section:** After finishing the execution Site  $S_i$  exits the critical section and does following: sets  $LN[i] = RN[i]$  to indicate that its critical section request  $RN[i]$  has been executed. For every site  $S_j$ , whose ID is not present in the token queue  $Q$ , it appends its ID to  $Q$  if  $RN[j] = LN[j] + 1$  to indicate that site  $S_j$  has an outstanding request. After above updation, if the Queue  $Q$  is non-empty, it pops a site ID from the  $Q$  and sends the token to site indicated by popped ID. If the queue  $Q$  is empty, it keeps the token.

```
def release_cs():
    global DATA_STRUCTURE
    with LOCKS["release"]:
        DATA_STRUCTURE["LN"][rank] = DATA_STRUCTURE["RN"][rank]
        for k in range(total_nodes):
            if k not in DATA_STRUCTURE["token_queue"]:
                if DATA_STRUCTURE["RN"][k] == (DATA_STRUCTURE["LN"][k]+1):
                    DATA_STRUCTURE["token_queue"].append(k)
                    print_log("%s: I'm %d and it adds %d to the queue. Queue after adding: %s." % (
                        datetime.now().strftime('%M:%S'), rank, k, str(DATA_STRUCTURE['token_queue'])))
                    sys.stdout.flush()
        if len(DATA_STRUCTURE["token_queue"]) != 0:
            DATA_STRUCTURE["has_token"] = 0
            send_token(DATA_STRUCTURE["token_queue"].popleft())
```

8. **print\_log()** and **print\_node\_status()** are the function that are logging and reading latest status of the node/process/site respectively. **print\_log()** appends latest status of the of site in a file dedicated to that site/process itself, along with it's LN, RN and token queue understanding. Function **print\_node\_status()** prints the same on **stdout** when a user enters and node rank on the CLI, which it reads from the log file.
9. Point to note that functions that both Simulator/Action and Listener thread have access: **run\_cs()**, **send\_token()**, **release\_cs()**, **print\_log()**. However, only Action thread access **request()** and only listener thread access **listen()** function.

## Example Run

We ran on 2, 3, 5, 8, 15 and 20 number of processes and made the simulation to run for atleast 15 minutes and atmax 25mins. Multiple log files have been obtained. For 3 process we get logs as follow:

### P1:

36:42: I am 0 and have a startup token.  
36:42: I am 0 and execute 1 CS.  
36:47: I'm 0 and finished 1 CS.  
36:47: I'm 0 and it adds 1 to the queue. Queue after adding: deque([1]).  
36:47: I'm 0 and it adds 2 to the queue. Queue after adding: deque([1, 2]).  
36:47: I'm 0 and sending the token to 1.  
36:48: I'm 0 and want a token for the 2 time.  
36:54: I'm 0 and I got a token.

### P2:

36:45: I'm 1 and want a token for the 1 time.  
36:47: I'm 1 and I got a token.  
36:47: I am 1 and execute 1 CS.  
36:51: I'm 1 and finished 1 CS.  
36:51: I'm 1 and sending the token to 2.

### P3:

36:44: I'm 2 and want a token for the 1 time.  
36:51: I'm 2 and I got a token.  
36:51: I am 2 and execute 1 CS.  
36:54: I'm 2 and finished 1 CS.  
36:54: I'm 2 and it adds 0 to the queue. Queue after adding: deque([0]).  
36:54: I'm 2 and sending the token to 0.

# Analysis Done

## 1. Time period between request for token and getting the token vs number of process/node/site

