Simulating MP Programs with Unreliable Communication on Shared Memory

CS403/534 - Distributed Systems

04 November 2023

PA1 Solution Report

Ataollah Hosseinzadeh Fard
ID: 28610

ConSet.py

1) Have imported needed libraries and modules.

```
8   from threading import Lock, Semaphore
9   import random
```

2) Have defined the constructor. Firstly, it creates an empty dictionary called set which will be used as out main set. Declared “mtx” as a lock which as from the name indicates it will be used as mutex, to ensure that critical sections are ran as atomic without interruption. Declared “sem” as semaphore with initial value of 0.

```
11  class ConSet:
12      def __init__(self):
13          self.set = dict()
14          self.mtx = Lock()
15          self.sem = Semaphore(0)
```

3) In insert method, it will add the new element if it does not exist in dictionary or if it exists and its value is false. Additionally, when it adds successfully, it signals (or releases) the semaphore so that it will wake up a single pop(). Since checking existence and then adding is a critical section it is wrapped in mutex block.

```
17      def insert(self, newItem):
18          self.mtx.acquire()
19          if newItem not in self.set or self.set[newItem] == False:
20              self.set[newItem] = True
21              self.sem.release()
22          self.mtx.release()
```

4) In pop method, it waits for semaphore to be signalled. If signalled, it creates a link of keys of dictionary, which are visible and hidden elements inside out dictionary. Then uses ‘random.shuffle()’ to shuffle the keys, this helps us to have random sequence of keys. By using this random sequence, I iterate over dictionary, and the first element that has True value, I change its value to false and then return that element. Accessing dictionary elements and changing its element is a critical section so that I have wrapped that part in mutex block. By usage of semaphore, it is ensured that every pop() will wait for an insert() to be operated, in other words for every successfully added new element a single pop() call will be allowed.

```
    def pop(self):
        res = None

        self.sem.acquire()
        self.mtx.acquire()
        keys = list(self.set.keys())
        random.shuffle(keys)
        for k in keys:
            if self.set[k] == True:
                self.set[k] = False
                res = k
                break
        self.mtx.release()

        return res
```

5) In ‘printSet’, it prints all the keys of dictionary that have True as their values, which means they are existing elements of our Set. Since accessing dictionary is a critical section, I have wrapped in a mutex block.

```
    def printSet(self):
        self.mtx.acquire()
        items = [str(k) for k, v in self.set.items() if v == True]
        print(", ".join(map(str, items)))
        self.mtx.release()
```

Leader.py

1) Have imported needed libraries and modules.

```
from ConSet import ConSet
from threading import Thread, Lock, Semaphore
import random
```

2) Have defined needed global variables. ‘N’ is the number of nodes that we will have. Only this ‘N’ can be modified for testing purposes. Have defined ‘mtx’ with lock as a mutex so that the print statements, which are critical sections, will be wrapped with mutex blocks. ‘count’ is a semaphore with initial value of 0, this will be used for checking number of nodes that do not find any leader. ‘barrier’ is a semaphore with initial value of ‘N’. this will be used as barrier mechanism to make sure that all nodes are at the same round.

```
# Number of Nodes
N = 4

mtx = Lock()
count = Semaphore(0)
barrier = Semaphore(N)
```

3) In the main part of ‘Leader.py’, have created 2 lists, one will hold the mailboxes of each node, and the other is list of threads that will run. Then for ‘N’ many times I added a ConSet to

mailboxes that will represent that node's personal mailbox, and added a new thread to threads list. Then sequentially have started the threads and have waited for threads to finish.

```
59  if __name__ == '__main__':
60      mailboxes = []
61      threads = []
62
63      for i in range(N):
64          mailboxes.append(ConSet())
65          threads.append(Thread(target=nodeWork, args=[i, N]))
66
67      for t in threads:
68          t.start()
69
70      for t in threads:
71          t.join()
72
```

4) In the 'nodeWork' function, firstly have defined the local variables to be used, one for round number and the other for stopping purposes. Then inside an infinite loop the main task of sending messages and reading messages begins. First of all, each node waits for barrier, at first round already since barrier's initial value is 'N' all nodes will easily pass with acquired call. Then node creates his own message as 'tup' which is tuple of his random number from [0, n*n] and his node id. After creating message, sends it to all the mailboxes available inside 'mailboxes' list, and now sending process is done for this node.

```
16  def nodeWork(node_id, n):
17      round_ = 1
18      terminate = False
19
20      while not terminate:
21          barrier.acquire()
22
23          tup = (random.randint(0, n*n), node_id)
24          for mailbox in mailboxes:
25              mailbox.insert(tup)
```

5) After sending to everyone (including itself), it starts to read his own mailbox. Since, ConSet is linearizable, there is no need that each node has to wait for all nodes to finish their sending phase, as soon as each node finishes sending can directly go to reading phase. Firstly defined 'maxId' and 'maxVal' that will be used for finding leader from messages. For 'N' times each node will try to read (pop) message from his own mailbox, and will assign this popped value as data. This data is tuple of random number and node id, so it can be easily be checked for having maximum. And there is a special case when any node reads a message from himself it has to print that value and since printing is a critical section, I have wrapped it in mutex block.

```
        maxId = []
        maxVal = -1
        for _ in range(n):
            data = mailboxes[node_id].pop()

            if data[0] > maxVal:
                maxVal = data[0]
                maxId = [data[1]]
            elif data[0] == maxVal:
                maxId.append(data[1])

            mtx.acquire()
            if data[1] == node_id:
                print(f"Node {node_id} proposes value {data[0]} for round {round_}.")
            mtx.release()
```

6) The last task is to decide on leader or continue to next round. This part was critical section due to print statements and accessing to value of ‘count’ semaphore, so I wrapped it in a mutex block. In this part it first checks if there is only 1 id that had maximum value generated. If yes, then terminates infinite loop and prints the leader as his decision. But if there are multiple nodes with maximum value, then it increments round number and releases ‘count’ mutex, which will increment its value by 1, and will print that this node has moved to next round. Here comes the part why I used ‘count’ semaphore as a counter, if the value of semaphore is ‘N’ which means all running node could not decide on leader, it will call empty print so that an empty line will be printed. Then it resets barrier’s value to ‘N’ and will reset ‘count’ semaphore value to 0. By these resets, I made sure that in next round barrier and count semaphores can be reused.

```
        mtx.acquire()
        if len(maxId) == 1:
            terminate = True
            print(f"Node {node_id} decided {maxId[0]} as the leader.")
        else:
            round_ += 1
            count.release()
            print(f"Node {node_id} could not decide on the leader and moves to round {round_}.")

            if count._value == n:
                print()
                for _ in range(n):
                    count.acquire()
                    barrier.release()
        mtx.release()
```