Simulating MP Programs with Unreliable Communication on Shared Memory

CS403/534 - Distributed Systems
Programming Assignment #1, Fall 2023

Deadline: November 5, 2023 (23:55 UTC+03:00)

For message passing (MP) programs, there are basically two kinds of communication: reliable and unreliable. Under reliable communication, message losses never occur and order of messages are preserved. TCP sockets and nodes with FIFO mailboxes are some examples for reliable communication. On the other hand, messages might be lost and/or order of messages might not be preserved under unreliable communication. UDP sockets and nodes with set mailboxes can be examples of unreliable communication.

In the lectures, we have seen how to represent reliable communication on shared memory using linearizable concurrent queues. You have a similar task for this programming assignment (PA). You will implement a linearizable concurrent set to represent unreliable communication. Then, you will implement a distributed leader election algorithm in which your concurrent set will simulate unordered message boxes of the nodes.

1 A Linearizable and Concurrent Set Implementation

For this PA, your set implementation will be specific to PYTHON programming language. You will implement a class called *ConSet*. The state of the set will be kept using `dictionary` primitive type [1] of PYTHON. Basically, keys of your dictionary will be the elements of the set and the values will be either `True` or `False` depending on the existence of the element in the set. For instance, if your set consists of only elements a and c, then the state of your implementation could be as follows:

$$\{a : True, b : False, c : True, d : False\}$$

You are asked to implement two of the set operations for this PA: `insert` and `pop`. The `insert` operation takes an object `elt` as an argument and first checks if this `elt` is an existing key in the dictionary. If so, it modifies `elt`s value to `True` in the dictionary. Otherwise, it inserts a new key, value pair $elt : True$ to the dictionary.

[1] See Section 5.5 of this tutorial for more info on PYTHON dictionaries.

The second operation `pop` just returns a random element from the set. To achieve this, the operation iterates over the keys of the dictionary. At each iteration, it reads the corresponding value of the current key. If it is `True`, the `pop` operation changes its value to `False` and returns the current key. Otherwise, it passes to the next key. If it cannot find a key with `True` value until the end of the dictionary's keys, it restarts the search hoping that some other node has inserted an element to the list in the meantime. Note that the `pop` operation is blocking when the set is empty and it might cause deadlocks when it is run concurrently with other set operations.

Above details describe the behaviour of operations when they are executed sequentially. If they are used concurrently, they do not guarantee linearizability correctness condition that we have seen in the lab session. So, they must be properly synchronized using mechanisms like locks, semaphores and/or condition variables. It is your duty to place synchronization mechanisms properly to ensure linearizability. You might use them directly inside the operation implementations or create another class called `SeqSet`, that implements sequential operations and `ConSet` keeps a sequential set inside it to represent the state and wrap sequential set's operations using synchronization primitives. Choice is totally yours. However, you need to ensure linearizability and that your operations never deadlock.

2 A Distributed Leader Election Algorithm

In this part, you are asked to simulate a leader election protocol using your concurrent set implementation. For this protocol, assume that there are n nodes. Each node has a mailbox that keeps its messages without respecting the order of their arrival but it does not allow message losses. So, you will use your concurrent set to simulate mailboxes of nodes. Mailboxes will be represented by a global list that stores n *ConSet* instances.

To simulate nodes, you need to implement a $nodeWork$ function that will be run by different threads. Node identifier (a non-negative integer) and n will be arguments of this function. It will generate a random number between 0 and n^2, create a tuple with this number and its id and will put this tuple into the mailboxes of all nodes (including its own) using the `insert` operation. Then, it will start receiving messages from its own mailbox using the `pop` operation. Since a node is aware of n it knows how many messages it is expecting. Among the received messages, its aim is to find the node with maximum generated value. If this value is unique, it prints this node's id and declares it as a leader. If there are more than one nodes with this maximal generated number, the whole process starts again by producing a new random number and putting it into mailboxes. Unfortunately, it is possible that nodes can create a livelock while choosing a leader meaning that they can keep generating the same maximal numbers and then restarting indefinitely. In the scope of PA, livelock is not a problem. We will ignore it hoping that eventually they will reach to a unique maximum.

3 Submission Guidelines

You need to submit at least two PYTHON files:

- `ConSet.py` contains the class with the same name that implements the concurrent set described in the first part of this document.
- `Leader.py` contains the distributed leader election algorithm described in the second section.
- (optional) `SeqSet` contains the sequential set implementation if you implemented the concurrent version by using the sequential implementation as the underlying class.

For this PA, you are not allowed to work in groups. So, every one needs to submit his/her own implementation. Required files explained above should be put in a single zip file named as `CS_403-534_PA01_name_surname.zip` and submitted to PA1 under assignments in SUCourse+. Submissions will end on 23:55 on the day of the deadline. There is no late submission but you can use a grace day if necessary (see the syllabus).

4 Grading

- Concurrent Set Implementation (60 Points): Your class *ConSet* must contain either a dictionary or a sequential set class you have implemented. If you prefer the latter, your sequential set class *SeqSet* must contain only a dictionary instance as its state. You are not allowed to use existing sequential or concurrent data type implementations like sets, lists, vectors, queues, stacks etc. Except this, your *ConSet* can only have synchronization variables like locks, semaphores and condition variables. You are advised to use `Threading` library of Python for this purpose. You are allowed to use other native libraries that provide shared memory synchronization mechanisms, however, then you might not get partial credit if your code does not work correctly or fail to run in our machines. Grading of your implementation will be done with respect to the following criteria:
  - (30 Points) Your implementation works as a correct sequential set i.e., its methods are not called by concurrent threads.
  - (10 Points) Your implementation is linearizable. Your set behaves correctly when called by concurrent threads.
  - (20 Points) Your set implementation does not allow deadlocks. Please be careful about how your code is synchronized.
- Distributed Leader Election (40 Points): We will check this part with our own correct set implementation. So, if your first part is not totally correct, it will not affect grading of this part. We will check a single condition for this part. We will initiate leader election with different n values and see if eventually all nodes agree on a leader. For this part, we again advise using `Threading` library of Python. If you want to use another library, you are free to do so but you take the responsibility described in the previous part.

6. Sample Runs

In this section, we provide sample runs for the program implemented in Leader.py. Assume that n = 4 for all cases. In the first example, the leader is elected in the first round:

```
Node 0 proposes value 1 for round 1.
Node 1 proposes value 6 for round 1.
Node 2 proposes value 7 for round 1.
Node 3 proposes value 1 for round 1.
Node 1 decided 2 as the leader.
Node 2 decided 2 as the leader.
Node 0 decided 2 as the leader.
Node 3 decided 2 as the leader.
```

In the second example, a decision could be made in the second round:

```
Node 0 proposes value 3 for round 1.
Node 1 proposes value 3 for round 1.
Node 2 proposes value 0 for round 1.
Node 3 proposes value 3 for round 1.
Node 0 could not decide on the leader and moves to round 2.
Node 1 could not decide on the leader and moves to round 2.
Node 2 could not decide on the leader and moves to round 2.
Node 3 could not decide on the leader and moves to round 2.

Node 0 proposes value 2 for round 2.
Node 1 proposes value 3 for round 2.
Node 2 proposes value 0 for round 2.
Node 3 proposes value 2 for round 2.
Node 1 decided 1 as the leader.
Node 2 decided 1 as the leader.
Node 0 decided 1 as the leader.
Node 3 decided 1 as the leader.
```