Homework 4

CS425/ECE428 Spring 2019

Due: Wednesday, April 24 at 11:55 p.m.

1. (a) (4 points) Modify the token ring mutual exclusion algorithm to support reader/writer-style mutual exclusion. As a reminder, a read lock can be shared with other nodes, but conflicts with a write lock.

> **Solution:** Let a single token get passed around the virtual ring. To add reader/writer mutual exclusion constraints, say multiple nodes can read at the same time if the token being passed around has no message attached to it or a 'CAN READ' message. If a node wants to write, they can attach a write message to the token and pass it to all the other nodes asking them to 'STOP READING'. When the node receives the token back, it can acquire the lock and start writing. Once it is done writing, it can attach a 'CAN READ' message and forward it to all the nodes. When a node sees a 'STOP READING' message in the token being passed around, if it has the read lock, it can hold the token until it finishes reading and then pass it on. If a node sees a 'CAN READ' message and it wants to read, it can go ahead and acquire the read lock to read the resource.

(b) (4 points) Analyze the best- and worst-case client and synchronization delays for N nodes. Note that there are two types of client delays: read and write client, and three types of synchronization delay: read/write, write/read, and write/write. Assume that the one-way communication delay between two nodes in the ring is T and there is no processing delay.

Solution:

Delays	**Best Case**	**Worst Case**
Read client delay	0	N*T
Write client delay	0	2N*T
Read/Write Synchronization delay	(N+1)*T	(2N-1)*T
Write/Read Synchronization delay	1*T	(N-1)*T
Write/Write Synchronization delay	(N+1)*T	(2N-1)*T

(c) (2 points) Does your algorithm create a possibility of reader or writer starvation? Justify your answer. (Note: you will not lose any points for an algorithm that can lead to starvation, nor gain any points for an algorithm that does not.)

> **Solution:** It can happen that every node in the ring that receives the token wants to write. In that case, if a node wants to read later on, it will have to wait for all writes to finish before it but there won't be an indefinite wait because the token will eventually be received by the node that wants to read it. So there will not be reader/writer starvation.

2. (a) (4 points) Describe a modification of the Bully protocol that elects a leader and a vice-leader, which should be the two nodes with the highest and second highest identifiers among all the live nodes. Note that all nodes should know which is the leader and vice leader. Write down the pseudocode for the algorithm, similar to what was contained in the lecture slides for Bully.

> **Solution:** When a processor find a leader or vice-leader failed:
> - If a node known its id is the highest, it elects itself as leader.
> - If a node known its id is the second highest, it elects itself as vice-leader.
> - Else it initiates a leader election.
>
> it sends an `Election` the message to all nodes with higher ids.

- If it received no answer calls itself leader by sending `Leader` message.
- If it received one answer calls itself vice-leader by sending `ViceLeader` message.
- If it received more than one answer waits for the new leader and vice-leader.

A process that receives an `Election` message from a lower ID replies with `OK`.

(b) (2 points) Analyze the worst-case time for an election among N nodes, assuming no nodes fail after an election has been called. Assume that the one-way communication delay between any nodes is T and there is no processing delay.

Solution:
Best case: Old Vice-Leader starts the new election, the election ends in One round so T.
Worst case: Lowest id starts the new election, the election will take $(N-1)$ iterations so $T(N-1)$.

(c) (3 points) How would you modify Raft to use a Bully-style election algorithm? What might be an advantage and disadvantage of this approach as compared with the Raft election?

Solution: Everytime that the leader heartbeat times out, the node starts a new election with Bully. A big advantage is that at most one leader will be present at any given time. The main disadvantage is that the nodes should be aware of the network partitions.

3. Consider an implementation of a bank account transaction participant. It supports five RPCs: deduct, which is called during a transaction, and canCommit, doCommit, and doAbort, which are called during two-phase commit. Assuming a transaction always uses the correct txid and only calls deduct once per transaction.

   Below is Python-like code implementing the RPCs:

```
def deduct(txid, amount):
  account.lock()
  will_commit[txid] = account.balance > amount
  account.unlock()

  saved_amounts[txid] = amount

def canCommit(txid):
  return will_commit[txid]

def doCommit(txid):
  account.balance -= saved_amounts[txid]

def doAbort(txid):
  pass  # do nothing
```

(a) (3 points) Which of the ACID properties does this implementation violate? Explain your answer.

Solution: Consistency. This may cause inconsistent system state. Consider the following two transactions:
T1: 1. `deduct(1, 100);` 2. `doCommit(1);`
T2: 1. `deduct(2, 100);` 2. `doCommit(2);`

When an interleaving of T1-1, T2-1, T1-2, T2-2 happens an account with balance of 150, both transactions will manage to commit, but as a result leave a negative balance.

Partial credits will be given to solutions which make sense to some extent.

(b) (3 points) Consider an alternate implementation:

```
def deduct(txid, amount):
  account.lock()
  account.balance -= amount
  will_commit[txid] = account.balance > 0
  account.unlock()

  saved_amounts[txid] = amount

def canCommit(txid):
  return will_commit[txid]

def doCommit(txid):
  pass # nothing

def doAbort(txid):
  account.lock()
  account.balance += saved_amounts[txid]
  account.unlock()
```

Which of the ACID properties does this implementation violate? Explain your answer.

Solution: Isolation. This may cause system results of certain interleaving different from serial execution. Consider the following two transactions:
T1: 1. `deduct(1, 100);` 2. `doAbort(1);`
T2: 1. `deduct(2, 100);` 2. `doCommit(2);`

When an interleaving of T1-1, T2-1, T1-2, T2-2 happens an account with balance of 150, T2 will falsely fail due to modification of `account.balance` by T1.

Partial credits will be given to solutions which make sense to some extent.

(c) (4 points) Write pseudocode that correctly implements all ACID properties

Solution: One possible solution is shown as follows.

```
def deduct(txid, amount):
    saved_amounts[txid] = amount
def canCommit(txid):
    account.lock()
    ok = account.balance > saved_amounts[txid]
    account.unlock()
    return ok
def doCommit(txid):
```

```
    if canCommit:
        account.balance -= saved_amounts[txid]
    else:
        doAbort(txid)
def doAbort(txid):
    pass # do nothing
```

4. Consider a system with 5 replicas. The latency to access each replica is given in the table:

Node	Latency
A	1 ms
B	3 ms
C	5 ms
D	30 ms
E	35 ms

(a) (2 points) What would you set the sizes of a read quorum (R) and write quorum (W) to be if you wanted to minimize write latency?

Solution: To minimize write latency, write quorum should be set to be minimum, $W = 3$. This requires $R = 3$.

(b) (2 points) What would be the read and write latency using those quorum sizes, assuming optimal choice of nodes for the quorum?

Solution: Read and write latency would be 5 ms by picking replicas A, B, C for the quorum.

(c) (2 points) What would be the read and write latency using those quorum sizes, assuming optimal choice of nodes for the quorum, if A has failed?

Solution: Read and write latency would be 30 ms.

(d) (4 points) Imagine D and E are located in a remote data center. How would you change the number of votes each node gets, and the quorum sizes, to ensure that each write is replicated outside the data center, while still minimizing write latency? What would be the write and read latency in your configuration?

Solution: Let A, B, and C have 1 vote each and D and E have 2 votes. This means that the smallest quorum has to be 4 and the smallest read quorum has to be 4. Note that the only way to get 4 votes would be to use at least one of D and E. This would result in 30 ms read and write latency.

Another solution is to allow all nodes to have 1 vote and require a write quorum to be 4. The write latency will remain at 30 ms, but the read quorum could be set to 2, allowing for a read latency of only 3 ms.