Assignment 4 - TDT4225

Olaf Rosendahl

October 31, 2022

1. **Kleppmann Chap 5**

   a) *When should you use multi-leader replication, and why should you use it in these cases? When is leader-based replication better to use?*

   A multi-leader replication is beneficial for some spesific use-cases. When using datacenters in multiple regions, having a leader in each region ensures that not every write-request must go to the same region which increases performance. It does also increase the tolerance of network and datacenter outages. A multi-leader replication is also beneficial when having clients which must function while offline and therefore keeps a local database which can be written to.

   Leader-based replication is simpler and one avoids write-conflicts.

   b) *Why should you use log shipping as a replication means instead of replicating the SQL statements?*

   There are various ways were replicating SQL statements may go wrong. If the statement that calls a nondeterministic function, such as $NOW()$ or $RAND()$, each replica will likely generate different values. If statements use an autoincrementing column or depend on the existing data in the database, each replica must execute them in the exactly same order. Side effects such as stored procedures may also result in different effects on each replica.

2. **Kleppmann Chap 6**

   a) *What is the best way of supporting re-partitioning? And why is this the best way?*

   The best way of supporting re-partitioning is using a fixed number of partitions according to Kleppmann. Here one creates many more partitions than there are nodes and then assigns several partitions to each node. If a node later is added to the cluster, it can take a few partitions from the existing nodes until the partitions are fairly evenly distributed once again. This is the best way since it doesn't move data around unnecessarily, which hash mod N for example does.

   b) *Explain when you should use local indexing, and when you should use global indexing?*

   Local indexing is quicker to write and therefore preferred when there is a lot of writes compared to reads. Global indexing makes writes slow and more complicated and is therefore more preferred when there is more reads which is very efficient.

3. **Kleppmann Chap 7**

   a) *Read committed vs snapshot isolation. We want to compare read committed with snapshot isolation. We assume the traditional way of implementing read committed, where write locks are held to the end of the transaction, while read locks are set and*

released when doing the read itself. Show how the following schedule is executed using these two approaches:
$r1(A); w2(A); w2(B); r1(B); c1; c2;$

Read committed:

i. Transaction 1: Reads initial value from DB A
ii. Transaction 2: Writes new value to DB A
iii. Transaction 2: Writes new value to DB B
iv. Transaction 1: Reads initial value from DB B (since Transaction 2 has write-lock and the new value isn't commited yet)
v. Transaction 1: Commits
vi. Transaction 2: Commits

Snapshot isolation:

i. Transaction 1: Reads initial value from DB A
ii. Transaction 2: Writes new value to DB A
iii. Transaction 2: Writes new value to DB B
iv. Transaction 1: Reads initial value from DB B (since there is no locks and the Transaction 2 is executed on a copy)
v. Transaction 1: Commits
vi. Transaction 2: Commits

b) *Also show how this is executed using serializable with 2PL (two-phase locking)*
Transaction 2 have to wait until Transaction 1 have committed since it tries to write to the same DB's as Transaction 1 have read from:

i. Transaction 1: Reads initial value from DB A
ii. Transaction 1: Reads initial value from DB B
iii. Transaction 1: Commits
iv. Transaction 2: Writes new value to DB A
v. Transaction 2: Writes new value to DB B
vi. Transaction 2: Commits

4. **Kleppmann Chap 8**

a) *If you send a message in a network and you do not get a reply, what could have happened? List some alternatives.*

i. The request may have been lost
ii. The request may be waiting in a queue and will be delivered later
iii. The remote node may have failed
iv. The remote node may have processed your request, but the response has been lost on the network
v. The remote node may have processed your request, but the response has been delayed and will be delivered later

b) *Explain why and how using clocks for last write wins could be dangerous*
It could be dangerous since each machine has its own clock which all could slightly faster or slower then other machines. This can in practice lead to a write that performed later in time can overwrite an existing write if the clock is slower and it therefore believe that its write was done first.

5. **Kleppmann Chap 9**

a) *Explain the connection between ordering, linearizability and consensus.*

Linearizability is about making a system appear as if there is only one copy of the data and all operations is atomic. Ordering preserves causal dependency between the question and the answer. If a system follows the ordering by said causality, it's causally consistent. Causal consistency do also provide a consistency which doesn't slow down due to network delays. Consensus is about having separate nodes agree on something. Election of a leader and atomic commits are examples where this is necessary.

b) *Are there any distributed data systems which are usable even if they are not linearizable? Explain your answer.*

The CAP Theorem describes who applications which doesn't need linearizability possibly can be more tolerant of network problems. If an application are created in a way where each replica handles requests independent from other replicas, even when disconnected, the application can also be available in the event of a network problem, without linearizable behavior. CAP is no longer used with new solutions superseding it, but it do provide insight to linearizable system-alternatives.

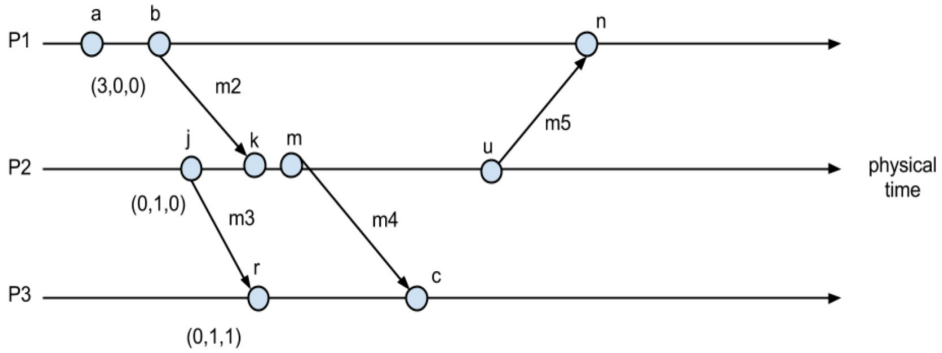
6. **Coulouris Chap 14**

a) *Given two events e and f. Assume that the logical (Lamport) clock values L are such that L (e) ¡ L (f). Can we then deduce that e "happened before" f? Why? What happens if one uses vector clocks instead? Explain.*

No we cannot. If the events happened in the same process, we can deduce that e happened before f. But if the events happened in different processes, Lamport times may differ between the processes and we can't with certainty deduce that e happened before f.

With vector clocks the implication of the ordering of events go both ways. This means that we with vector clock *can* deduce that e happened before f.

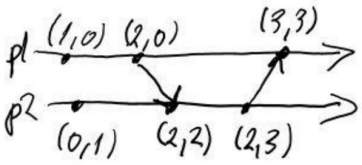
b) *The figure below shows three processes and several events. Vector clock values are given for some of the events. Give the vector clock values for the remaining events.*

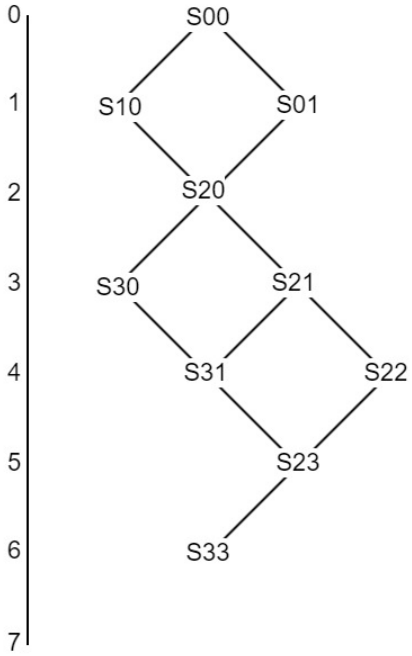


b: (4, 0, 0)
k: (4, 2, 0)
m: (4, 3, 0)

c: (4, 3, 2)
u: (4, 4, 0)
n: (5, 4, 0)

c) *The figure below shows the events that occur at two processes P1 and P2. The arrows mean sending of messages. Show the alternative consistent states the system can have had. Start from state S00. (Sxy where x is p1's state and y is p2's state)*





7. **RAFT**

RAFT has a concept where the log is replicated to all participants. How does RAFT ensure that the log is equal on all nodes in case of a crash and a new leader?

All interactions and log replications with clients are handled by the leader. After a change of leader, the previous leader may have some entries left which were only partially replicated. There isn't necessary to do anything with this, since it's the new leader's log that is considered correct and which the clients will copy.

In order to make sure that log entries will be present on future leaders, leaders never overwrite entries in their own log and only entries in the leader-log may be committed. Commits must also happen before applying logs to the state machine.

Another safety-measure is that when a new leader should be elected, an evaluation is done to find the most suitable candidate by picking the candidate which is most likely to contain all the committed log-entries.