A04 - Assignment 4

- **Deidier Simone** - *student number: 133020*

Theory questions

1. In the shopping cart example, **version numbers** capture the *happens-before* relationship by marking the sequence of updates, in fact when two concurrent updates to the cart are made (e.g., from different devices), version numbers act as **logical clocks**, recording the time of each action. These numbers help detect conflicts (i.e., concurrent changes needing a merge), ensuring that each device has a **consistent cart state** by deciding the most recent or conflict-resolved version.

2. According to Kleppmann, the best re-partitioning method is *consistent hashing*. This approach **minimizes the number of keys that need to move** when partition count changes, reducing re-partitioning overhead. *Consistent hashing* also provides scalable load balancing as nodes or partitions are added or removed.

   - **Local indexing** is ideal for databases where each partition has its index, enabling **faster, independent queries** on each shard, but it's less suitable for **cross-partition queries**.
   - **Global indexing** is needed for queries covering multiple partitions, providing a **global view**. While it's more complex and slower, it ensures *data consistency* across partitions.

3. Read Committed vs. Snapshot Isolation for the schedule `r1(A); w2(A); w2(B); r1(B); c1; c2`:

   - **Read Committed**: `T1` reads `A` before `T2` writes it, seeing the original value. `T1` reads `B` after `T2` writes it, seeing the **new value of `B`**. **Outcome**: `T1` might see *mixed old and new values*, leading to potential anomalies.
   - **Snapshot Isolation**: `T1` takes a *consistent snapshot* of `A` and `B` from the start. **Outcome**: `T1` sees the same versions of `A` and `B`, avoiding anomalies present in *Read Committed*.
   - With **2PL**, `T1` and `T2` adhere to the two-phase locking protocol: **locks prevent conflicting reads and writes** until transactions complete, this approach guarantees serializability by **preventing anomalies** from concurrent access.

4. Possible reasons for no reply to a network message are **network latency** or congestion, the receiver node may have *crashed or become unreachable*, the message may be *lost in transit* or dropped, acknowledgment failure or *firewall/security* rule blocking message return.

   - Using clocks for *last write wins* is dangerous due to **clock drift** between nodes, which can cause inaccurate conflict resolution. One

node's "newer" write may appear "older" on another, leading to inconsistencies.
- A node may not trust its judgment in cases like **network partitions** or **clock drift**, where it might believe it has the latest data but cannot confirm with other nodes.

5. **Ordering** ensures operations apply in a set sequence, **linearizability** guarantees a strong *consistency model*, where each operation appears to occur at a specific instant, and **consensus** (e.g., RAFT, Paxos) ensures all nodes agree on an operation order, achieving *linearizability* across a distributed system.
   - *Non-linearizable systems* like eventual consistency databases are widely used where strict consistency isn't necessary, in fact they offer high availability and partition tolerance, fitting **large-scale applications**.
   - Ensuring linearizability can reduce **system performance** and increase **latency**. The high cost of coordination often makes linearizability unsuitable for high-performance, highly available systems.
6. With **logical clocks**, `L(e) < L(f)` doesn't guarantee `e` happened before `f` due to possible concurrency. However, *vector clocks* can capture causality, so if `V(e) < V(f)`, then `e` indeed happened before `f`. Vector clock values for remaining events in the figure:
   - $b \to (4, 0, 0)$
   - $k \to (4, 2, 0)$
   - $m \to (4, 3, 0)$
   - $c \to (4, 3, 2)$
   - $u \to (4, 4, 0)$
   - $n \to (5, 4, 0)$

Alternative consistent states from state S_{00}:

- S_{00}: $(0, 0)$ for P_1 and $(0, 0)$ for P_2 (initial state)
- S_{10}: $(1, 0)$ for P_1, $(0, 0)$ for P_2 (P_1 has completed an event, but no messages exchanged yet)
- S_{11}: $(1, 0)$ for P_1, $(0, 1)$ for P_2 (P_2 has advanced independently of P_1)
- S_{20}: $(2, 0)$ for P_1, $(0, 1)$ for P_2 (P_1 advances again without communication)
- S_{21}: $(2, 0)$ for P_1, $(2, 1)$ for P_2 (P_2 receives the message from P_1, updating its clock)
- S_{22}: $(2, 2)$ for P_1, $(2, 2)$ for P_2 (P_2 performs an event and sends a message to P_1)
- S_{33}: $(3, 3)$ for P_1, $(2, 3)$ for P_2 (P_1 receives the message from P_2, and both clocks are synchronized)

7. RAFT ensures **log consistency** by requiring new leaders to synchronize logs with the majority before appending new entries. The leader **replicates its log to followers**, overwriting inconsistencies to maintain uniform logs across all nodes.

8. The main advantages of RAFT in MySQL are **improved consensus and fault tolerance**, **consistency across replicas** and **simplifies leader election and log replication**, enhancing MySQL's reliability for distributed transactions.

9. Stonebraker and Pavlo note that while **document databases** (e.g., MongoDB) are flexible for unstructured data, *SQL databases* have regained popularity for their **ACID guarantees** and **complex querying power**. A convergence trend shows SQL databases adopting schema flexibility, while NoSQL systems strengthen transaction guarantees, blending benefits across both types.