Q: Could a received InstallSnapshot RPC cause the state machine to go backwards in time? That is, could step 8 in Figure 13 cause the state machine to be reset so that it reflects fewer executed operations? If yes, explain how this could happen. If not, explain why it can't happen.

A: No, a received InstallSnapshot RPC cannot cause the state machine to go backwards in time and reflect fewer executed operations. The reason for this is rooted in how the Raft consensus algorithm is designed to maintain consistency and ensure that the state machine's execution progresses monotonically forward.

In the Raft algorithm:

1. Monotonicity of Log Indexes:
   Log entries in Raft are identified by their index, and each log entry has a unique index that increases monotonically over time. When a follower receives an InstallSnapshot RPC, it discards all log entries up to a certain index specified in the snapshot metadata. However, it does not reset its log index to a lower value; instead, it advances its log index to the next entry after the snapshot.

2. Consistent Log Matching:
   Raft ensures that the logs of the leader and followers are consistent. When a follower applies a snapshot and discards old log entries, it ensures that the log entries following the snapshot are received and applied in the correct order. This ensures that the follower's state machine reflects the same sequence of operations as the leader's state machine.

3. Safety Property:
   Raft ensures the safety property, which guarantees that a committed log entry will never be overwritten or removed from the log. Once a log entry is committed by a majority of nodes, it is considered safe and cannot be lost, overwritten, or reverted.

Therefore, even if a follower applies a snapshot and discards old log entries, it will not cause the state machine to go backwards in time or reflect fewer executed operations. Instead, the follower's state machine will reflect the same sequence of operations as the leader's state machine, ensuring consistency and safety in the distributed system.

Q: Suppose we have the scenario shown in the Raft paper's Figure 7: a cluster of seven servers, with the log contents shown. The first server crashes (the one at the top of the figure), and cannot be contacted. A leader election ensues. For each of the servers marked (a), (d), and (f), could that server be elected? If yes, which servers would vote for it? If not, what specific Raft mechanism(s) would prevent it from being elected?

A: Server (A) can become the leader in case where its election times out before server D, because it will get votes from, only server B, E, and F would vote for it which will count as majority(4/7) but it can not become leader if server D's election clock starts first as server D will get more votes.

Server (D) will win it because it has the highest term number and longest log, so every server will vote for server D to become the leader.

Server (F) cannot be elected as a leader because, as it has stale data, every other follower has a higher term number than server F.

Q: With a linearizable key/value storage system, could two clients who issue get() requests for the same key at the same time receive different values? Explain why not, or how it could occur.

A: In a linearizable key/value storage system, two clients who issue `get()` requests for the same key at the same time cannot receive different values. This is because linearizability ensures that operations appear to be executed atomically and instantaneously at a single point in time, providing the illusion of a single, centralized copy of the data.

Here's why two clients cannot receive different values for the same key in a linearizable system:

1. Real-Time Ordering: Linearizability guarantees that the order of operations observed by clients reflects the real-time order in which they were performed. When two clients issue `get()` requests for the same key simultaneously, the linearizable system ensures that these requests are processed in a consistent order.

2. Atomicity of Operations: Each `get()` operation in a linearizable system is atomic, meaning it retrieves the value associated with the key at a specific point in time. Once a `get()` operation is completed, the value returned to the client is consistent with the state of the key/value storage at the time of the operation.

3. Single Point of Truth: Linearizability provides the illusion of a single, centralized copy of the data. Even though the key/value storage may be distributed across multiple nodes, linearizability ensures that clients see a consistent and up-to-date view of the data, regardless of which node they interact with.

However, it's essential to note that while linearizability prevents clients from receiving different values for the same key simultaneously, other consistency models, such as eventual consistency or weak consistency, may allow for such scenarios. In systems employing weaker consistency models, clients may observe temporary inconsistencies due to replication delays or concurrent updates.