1. **Scaling with event driven architecture**

We used an event driven approach for handling I/O and other asynchronous events, such as timer interrupts. Our architecture follows the reactor design pattern and is implemented with the java NIO framework. Our key-value store consists of two core components, the server component and the client component, the server component handles everything server related, such as handling key-value requests, migration, replication and node membership, while the client component handles sending requests to other nodes, ensuring that any requests sent gets a success response, otherwise it retries up to N times. Our key-value store also utilizes a thread pool for handling requests and other tasks, and a scheduler for handling delayed tasks.

The server has one single thread operating an event loop, which registers read and write handlers for the corresponding DatagramChannels, we have multiple channels, one channel receives key-value requests from the client and sends the key-value response, one channel for routing key-value requests to other nodes, one channel for migration and replication and one channel for the epidemic protocol. Each channel registers an event handler for both read and write events, when a channel has data to be read, the read event handler puts the read data into a buffer and creates the corresponding task to be run on a thread in a thread pool. When writing to a channel, a thread first tries to send the data right away, if it is not able to do so then it triggers the channel to listen for a write event while placing the data in a queue, once the channel is ready for writing the write event handler sends the data in the queue and if the queue becomes empty it stops listening for write events.

The client component which is in charge of sending requests to nodes uses timer interrupts to detect time outs, whenever a request is recorded to be sent, a task is scheduled with the scheduler to be executed after some delay, when the task executes it will check if any requests timed out, if there are still requests left that has yet to be received or still have retries left then the task is rescheduled.

The main feature in our architecture is that there are no blocking threads, except the event loop, and all other work are short lived and completed by either the thread pool or the scheduler.

2. **Sequential consistency using a modified vector clock approach**

To achieve sequential consistency, we started with the vector clock concept seen in class and modified it slightly to suit our server architecture, and applied some space optimization to make it more efficient. We decided to associate each entry with its own vector clock (this way nodes can test for causality of incoming put requests locally for each entry independently).

Our modified vector clock is a flattened int array, storing pairs of *nodeID* and *seqNum* (clock count) values (*nodeID* at even indices *i*, associated *seqNum* at *i+1*). In order to save space we don't use a more complex data structure such as ArrayList. To further optimize for memory usage, we start with a small array (size 2) and increase the length of the int array manually as more nodeIDs are added to a vector clock. If a nodeID is not found in a vector clock, this is implicitly understood as it having a clock count of 0.

The update logic for the vector clock is straightforward. Every put request sent by a server node has a vector clock associated with its particular key. However requests from clients do not have a vector clock. If a node receives a request from a client, the local vector clock for the entry in question is incremented directly (entry of self node ID is incremented). If the request was forwarded from another node, the vector clocks for the entry are compared, and the put is performed only if the incoming vector

clock is *larger*. If the two vector clocks are *uncomparable*, we randomly pick one, which eventually leads to a consistent version throughout the key store.

The performance numbers attest to this approach, as our server nodes seem to handle scaling up to 1024 clients quite well. The sequential consistency tests show that our approach is resilient even under high loads, with almost 100% get success rate in all rounds. We also note that the only unsuccessful responses were timeouts, not errors.

3. **Implicit identification of replicas and simplistic approach to membership**

We used consistent hashing to implicitly identify replicas via the hash ring so that no replica information needs to be stored or passed around. In our case, when a node is hashed onto the ring, its keys are replicated onto *N* successor nodes. *N being the replication factor - 1.*

When a node joins, all other nodes will add the new node to their hash ring and scan their key space to determine if any keys needs to be copied over, this simplifies the implementation and guarantees correctness in all node joining scenarios. If the joining node is also a successor to a node then all keys that hash directly to the node is replicated onto the joining node.

Similarly, when a node leaves, all other nodes will check whether the node leaving is within *N* successor to them. It will also check whether the node leaving is the direct predecessor to itself. In both cases, it replicates all keys that hash to itself to its *new* N successors on the ring.

This simplistic approach ensures correctness in different failure scenarios, but It is possible for this approach to replicate extra keys, so each node uses a periodic check-up task, scheduled to automatically run every 5 minutes (but can be tweak depending on the memory limit and number of keys), to remove extra replicas from its KVstore. This task can run independently without message passing as it also uses the implicit replica information from the hash ring.

4. **Performance**

1 Client random front end goodput: 5 reqs/sec
1024 Client random front end goodput: 5800 ~ 6000 reqs/sec
Our architecture is able to scale close to the ideal scaling.

5. **Testing**

We developed automated tests similar to the correctness tests executed by the evaluation client and we used these tests locally to test for correctness and aid in debugging.

We have also found it useful to create mock clients for local tests that target a specific stage in order to isolate potential bugs. For example, to mimic stage 3 we created a client that deterministically sends put and get requests to specific keys, while we shut down and restart local nodes. This approach means we can replicate and trace through deterministic cases that cause errors, allowing us to test for correctness much more precisely than if we just went blind with Planetlab nodes.