Experimental Evaluation of ABD and Blocking Protocols

CSE 511 Project 2

Abstract

This project implements and evaluates two protocols for a replicated, linearizable key-value store: the classic ABD algorithm for atomic read/write registers and a lock-based blocking quorum protocol. Both run on a cluster of servers that communicate via TCP sockets, with a multi-threaded workload generator issuing mixed GET and PUT operations. We compare throughput and latency as we vary the number of replicas ($N \in \{1, 3, 5\}$), number of concurrent clients, and workload mix (GET-heavy and PUT-heavy). Our measurements show the expected trade-offs: the blocking protocol can achieve higher throughput at low replication or low contention thanks to exclusive locks, but it degrades under high concurrency due to lock contention and lease interactions. ABD scales more gracefully with clients and replicas because it avoids locks at the cost of additional read and write phases.

1 Introduction

Replicated key–value stores are a fundamental building block in distributed systems. Clients expect strong semantics such as linearizable reads and writes even in the presence of concurrency and partial failures. Implementing such semantics over a set of unreliable servers requires careful coordination to ensure that every operation observes a consistent order of updates.

In this project, we implement and experimentally compare two such coordination mechanisms:

- **ABD** (Attiya–Bar-Noy–Dolev) style atomic read/write register using versioned tags and quorum reads/writes.
- **Blocking quorum protocol**, where clients acquire per-key locks on a quorum of replicas before reading or writing.

Both protocols are built on the same common networking and type definitions, and both expose a simple GET/PUT interface from the perspective of the workload generator.

2 System Model and Common Infrastructure

2.1 Servers and Communication

Each server runs as a standalone TCP server process listening on a configurable port. Clients connect over TCP using helper routines provided in the common networking library. A `ServerInfo` structure describes each replica (IP and port), and the workload generator maintains a vector of such servers to target.

Requests and responses are exchanged as single-line strings terminated by `\n`. The common `Network` module provides:

- `connect_to_server`: establishes a TCP connection to a replica.
- `send_message`: sends a full line over the socket.
- `recv_line`: reads one line until newline.

2.2 Key State and Tag Representation

Both protocols share a logical representation of each key. On the server side, a `KeyState` structure maintains:

- `tag_lamport`: integer Lamport-like counter.
- `tag_cid`: client identifier component.
- `value`: the stored string value.

In the blocking protocol, `KeyState` additionally tracks locking state:

- `locked_by`: client id holding the lock (or -1 if free).
- `lock_expiry`: time point implementing a lock lease.

On the client side, read responses are captured in a `ReadResp` structure containing the tag components, value, and a `valid` flag. These are used during quorum aggregation to select the maximum tag.

3 ABD Protocol

The ABD client implements two high-level operations `get` and `put` that operate against a set of replicas and run the two-phase quorum logic.

3.1 Client-Side Read and Write Phases

For each operation, the client chooses a read/write quorum size

$$R = \left\lfloor \frac{N}{2} \right\rfloor + 1,$$

where N is the number of replicas. It uses the first R servers in the configuration as its quorum set.

The *read phase* sends `READ_REQ key` to each server in the quorum in parallel threads and collects `READ_RESP` messages, each returning a tag and a value. The implementation uses a vector of threads that are joined at the end of the phase, and stores results in a shared vector of `ReadResp` entries. The function `find_highest_tag` scans these responses to compute the maximum tag according to the lexicographic order first on `tag_lamport` then on `tag_cid`. If no valid responses are received, the operation fails.

For `get`, after the read phase determines the highest tag and associated value, the client performs a *write-back phase* to ensure that a majority of replicas hold at least this latest value. It sends `WRITE_REQ key tag_lamport tag_cid value` to each quorum server in parallel and waits for acknowledgments. The client then returns the value to the caller.

For `put`, the client first runs a read phase to obtain the current maximum tag, increments the Lamport component, sets the client id as the second component, and then runs the write phase with the new tag and the user-provided value. This ensures that each write has a globally unique, totally ordered tag.

3.2 ABD Server

The ABD server maintains a per-key `KeyState` map protected by a mutex. Upon receiving `READ_REQ key`, it returns the current tag and value. For a `WRITE_REQ`, it parses the incoming tag and value, compares the new tag against the stored one, and conditionally updates the key if the incoming tag is newer. It then replies with `ACK`. Any unknown command results in an `ERR` response.

This design is stateless with respect to clients: the server stores only the latest version of each key and does not track locks or leases.

4 Blocking Lock-Based Quorum Protocol

The blocking protocol uses explicit per-key locks on a quorum of replicas to ensure exclusive access while a client performs its read and write.

4.1 Client-Side Locking Workflow

The client begins each operation by acquiring locks on a quorum of servers. It again chooses $R = \lfloor N/2 \rfloor + 1$ and attempts to contact all N replicas in parallel using `LOCK_REQ key client_id`. Each replica replies with either `LOCK_GRANTED` or `LOCK_DENIED`. The client collects grants in a shared vector protected by a mutex. Once it has collected R grants, it sets a stop flag to avoid wasting TCP connections.

If, after all threads join, the client holds locks on fewer than R servers, it releases any acquired locks using `UNLOCK` and returns failure. Otherwise, it truncates the granted list to exactly R servers and proceeds.

With the locks held, `get` performs a read quorum similar to ABD: it issues `READ_REQ` to the locked replicas, finds the highest tag among valid responses, and returns the associated value. It then releases the locks via `UNLOCK` messages.

For `put`, the client with *locks held* first performs a read quorum to find the maximum tag, increments the tag, and then runs a *write quorum* by sending `WRITE_REQ` with the new tag and value to each locked server. The client counts successful acknowledgements and considers the write successful if at least R servers respond with `ACK`. Finally, it unlocks the quorum.

4.2 Blocking Server and Lock Leases

The blocking server extends the key state with lock information. Upon `LOCK_REQ key client_id`, it checks whether any existing lock has expired (by comparing against `lock_expiry`). If so, it releases the lock. If the key is currently unlocked, it records `locked_by = client_id` and sets a lease expiration time `lock_expiry = now + LOCK_LEASE_SEC`, replying with `LOCK_GRANTED`. Otherwise it replies `LOCK_DENIED`.

Unlock requests `UNLOCK key client_id` clear the lock if the requester holds it or if the lease has expired. Reads (`READ_REQ`) are allowed regardless of who holds the lock; they simply return the current tag and value. Writes (`WRITE_REQ key tag_lamport tag_cid value`) are allowed only if the writer's client id matches `locked_by` and the lock has not expired. The server updates the value if the incoming tag is newer and responds with `ACK`; otherwise it responds with `WRITE_DENIED`.

This design ensures that, as long as leases are long enough relative to operation duration, a client holding a quorum of locks has exclusive write access on that quorum and thus can implement linearizable operations.

5 Workload Generator and Measurement

5.1 Workload Structure

The workload generator is protocol-agnostic: it accepts a command-line argument specifying whether to use ABD or the blocking client, and then passes function pointers for `get` and `put` into each worker thread. It also parses:

- Number of clients (threads).
- Operations per client.
- GET fraction (e.g., 0.9 for GET-heavy).

- Number of distinct keys.
- List of server addresses (IP:port).

Each worker thread loops for a fixed number of operations. For each operation, it:

1. Randomly picks a key index from `[0, num_keys)` and constructs a key string.
2. Draws a uniform random number in $[0, 1)$ to decide whether to issue a GET or PUT based on the configured GET fraction.
3. For GET:
   - Records the start time, calls the selected `get` function, records the end time.
   - Appends the latency in microseconds to a shared vector of GET latencies (protected by a mutex).
   - Increments either the global successful GET counter or the failure counter.
4. For PUT:
   - Constructs a value string depending on the client id and a random component.
   - Measures latency and updates global PUT success/failure counters and latency arrays.

At the end of the experiment, the main thread computes:

- Total attempted operations = clients $\times$ ops per client.
- Total successful operations = successful GETs + successful PUTs.
- Elapsed time in seconds.
- Throughput in ops/sec computed using only successful operations:

$$\text{throughput} = \frac{\text{succ\_get} + \text{succ\_put}}{\text{elapsed}}.$$

- Median and 95th percentile latency for GET and PUT by sorting the recorded latency vectors and selecting the appropriate index.

5.2 Experimental Parameters

We swept the following parameters:

- Number of replicas $N \in \{1, 3, 5\}$.
- Number of concurrent clients $\in \{1, 2, 4, 8, 12, 16, 20, 24, 32\}$.
- Workload mixes:
  - GET-heavy: GET fraction $= 0.9$.
  - PUT-heavy: GET fraction $= 0.1$.

The number of operations per client was fixed to 2000 ops and the number of distinct keys to 10. For each configuration, we ran both protocols and logged the resulting throughput, success counts, failure counts (e.g., due to insufficient locks in the blocking protocol), and latency percentiles.

6 Results

This section presents the key graphs and interprets their trends. All figures use the CSV-generated PNG files listed in Section **??**.

6.1 Throughput vs. Clients

Figures 1–3 plot successful throughput as a function of the number of clients for each replication factor N.

Figure 1: Successful throughput vs. clients, $N = 1$.

Figure 2: Successful throughput vs. clients, $N = 3$.

For $N = 1$, the measurements show that *ABD achieves higher successful throughput than the blocking protocol for every client count we tested*. ABD scales almost linearly up to about 12 clients (reaching roughly 3.7×10^4 ops/sec) and then essentially saturates: from 12 to 32 clients the curve is nearly flat, so adding more threads does not increase capacity. In contrast, the blocking protocol grows much more slowly and saturates earlier: it peaks around 8 clients and then its successful throughput *drops* as we add more clients.

The key reason is how lock contention is handled in our blocking design. In the blocking client, if a thread fails to obtain locks on a quorum of replicas for a key (because those replicas already hold non-expired leases for another client), the operation is recorded as a *failure* and returns immediately *without retrying*. We made this choice deliberately:

- to keep the client logic simple and focused on the protocol itself rather than on retry policies,
- to make the impact of lock contention directly visible in the results (as increasing failure counts and lower successful throughput), and

Figure 3: Successful throughput vs. clients, $N = 5$.

- to avoid long retry storms that would complicate tail-latency and fairness analysis.

Because we only count *successful* operations in the throughput metric, once many clients contend on the same keys, more blocking operations fail outright instead of eventually succeeding, and the blocking curve flattens and then declines.

For $N = 3$, this pattern becomes even more pronounced. ABD throughput increases up to about 8 clients (around 2.0×10^4 ops/sec) and then saturates, with only a mild decrease beyond that point as the server CPU and network become the bottleneck. The blocking protocol, however, peaks very early (around 4 clients) and then steadily loses successful throughput as we add clients. The failure counts for blocking grow quickly with concurrency, reflecting the fact that more lock acquisition attempts collide and are not retried.

For $N = 5$, ABD again scales reasonably with clients but with a lower overall throughput level due to the higher replication cost. The curve increases from 1 to about 4–8 clients and then oscillates around a plateau (roughly 9k–11k ops/sec), with a slight bump near 20 clients but no sustained gains beyond that. The blocking protocol is consistently worse: it reaches its maximum successful throughput already around 4 clients and then degrades sharply as client count increases, because the probability that a given key is currently locked by someone else grows with concurrency and each such conflict turns into a failed operation.

Figure 4 summarizes this behavior for the GET-heavy workload across all N:

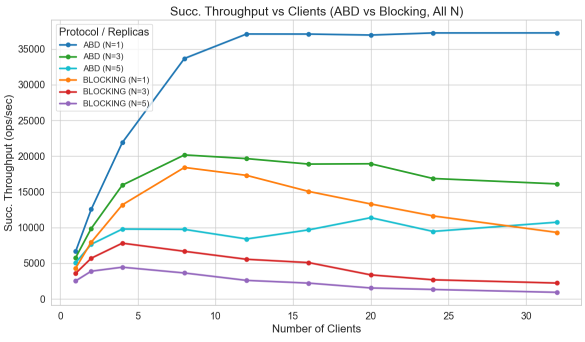

Figure 4: Successful throughput vs. clients for GET-heavy workload, ABD vs. Blocking, $N \in \{1, 3, 5\}$.

The aggregated curves highlight two main points. First, ABD scales more smoothly with the number of clients for all replication factors we tested, with clear saturation points (around 12 clients for $N = 1$, 8–12 for $N = 3$, and roughly 4–8 for $N = 5$). Second, because the blocking protocol does not retry on lock failure and instead treats each failed lock acquisition as a permanently failed operation, its successful

throughput saturates much earlier (around 4–8 clients depending on N) and then declines as contention

6.2 GET Latency

Figures 5–10 show median and 95th percentile GET latencies as the number of clients grows.

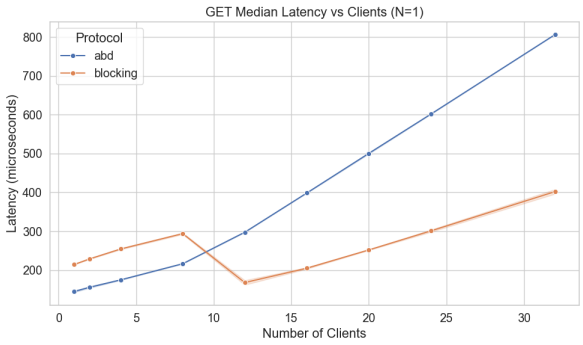

Figure 5: GET median latency vs. clients, $N = 1$.

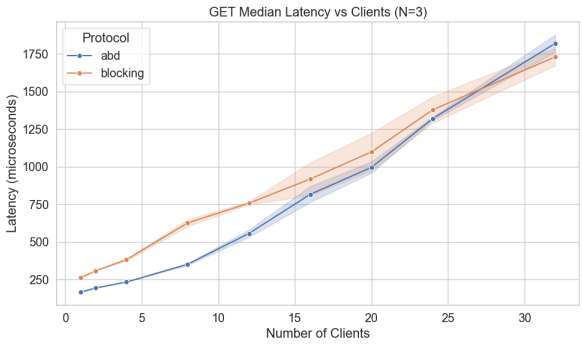

Figure 6: GET median latency vs. clients, $N = 3$.

At low client counts, both protocols deliver low median latency. For $N = 1$, the blocking protocol may have a slight latency advantage since ABD still performs a read and a write-back phase even though there is only one replica. As we increase the number of clients and replicas, contention effects become visible. The blocking protocol's latency can grow faster because clients occasionally fail to obtain locks immediately and must retry or experience failed operations. This is especially visible in the 95th percentile curves, which capture queuing delay and contention.

ABD's GET path, while heavier in terms of messages (read plus write-back), has more predictable latency because it avoids explicit locks and instead relies on quorum intersection.

6.3 PUT Latency

Figures 11–16 show PUT median and 95th percentile latency.

PUTs are more expensive than GETs for both protocols. In ABD, PUT performs a read phase to obtain the maximum tag and then a write phase to a quorum. In the blocking protocol, PUT must first acquire locks, then read, then write, then unlock, which introduces additional round-trips and potential contention. At low N and low concurrency, the blocking protocol can still perform competitively because lock acquisition is

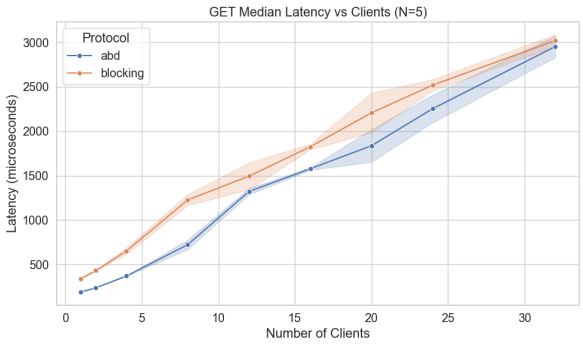

Figure 7: GET median latency vs. clients, $N = 5$.

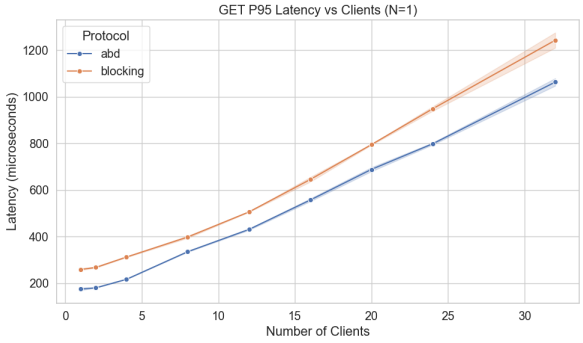

Figure 8: GET 95th percentile latency vs. clients, $N = 1$.

cheap. With more clients, however, the tails (p95) widen substantially due to lock contention and lease interactions.

Overall, the results align with theoretical expectations: ABD trades extra message overhead for better composability and less lock contention, while the blocking protocol can be very fast when contention is low but suffers as concurrency rises.

7 Conclusion

We implemented and evaluated two protocols for a replicated linearizable key-value store: an ABD-style atomic register and a lock-based blocking quorum protocol. Using a unified workload generator, we measured throughput and latency across varying numbers of replicas, clients, and workload mixes on the Penn State W135 machines.

Our experiments show that:

- Across all replication factors $N \in \{1, 3, 5\}$ and for almost all client counts we tested, **ABD achieves higher successful throughput than the blocking protocol**. Even at $N = 1$, where we might expect the lock-based protocol to shine, ABD dominates in successful ops/sec because the blocking protocol suffers from lock conflicts that are treated as immediate failures.

- As the number of replicas and clients increases, **ABD scales more gracefully**: its throughput increases with concurrency up to a saturation point and then flattens. In contrast, the blocking protocol reaches its peak throughput at relatively low client counts and then *loses* successful throughput as concurrency grows, due to a rising fraction of operations that cannot obtain a quorum of locks.

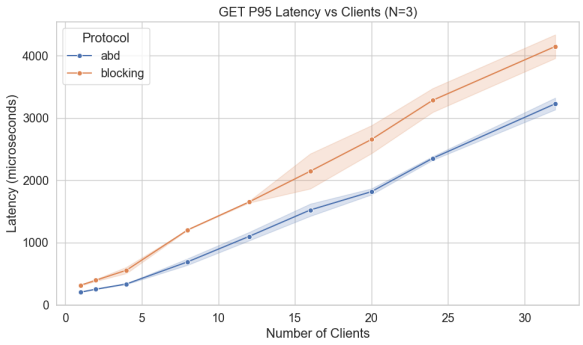

Figure 9: GET 95th percentile latency vs. clients, $N = 3$.

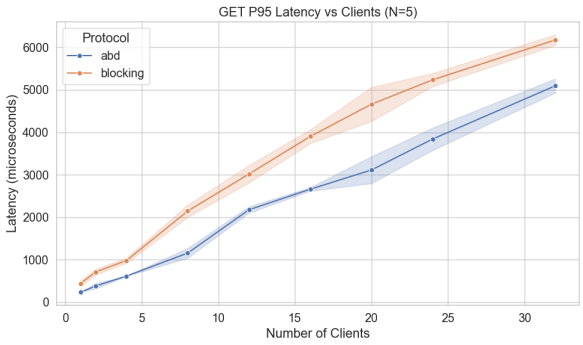

Figure 10: GET 95th percentile latency vs. clients, $N = 5$.

- In our blocking design, a failed lock acquisition is **not retried**. This design choice makes lock contention directly visible as failed operations rather than hidden behind client-side backoff. As a result, GET and PUT latencies for *successful* blocking operations remain comparable to ABD at low concurrency but the protocol exhibits a steeper increase in 95th percentile latency as contention grows, while at the same time many other operations fail quickly and do not contribute to the latency statistics.

These results highlight the trade-off between lock-based and quorum-based designs in a concrete implementation. Locks can be straightforward to reason about and work well at low contention, but without careful retry and backoff policies they are fragile under concurrency and can waste capacity on failed attempts. Quorum-based protocols like ABD pay an overhead in extra read/write phases, but in exchange they avoid explicit locks and provide more robust scaling of successful throughput as the system size and the number of clients increase.

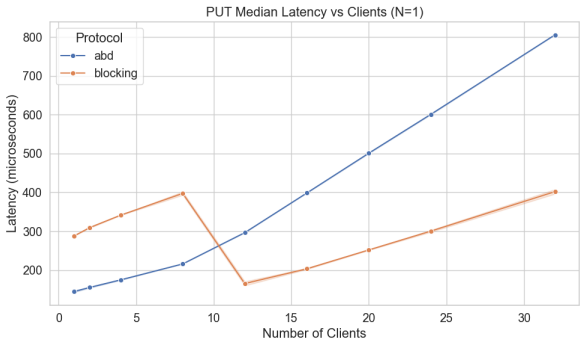

Figure 11: PUT median latency vs. clients, $N = 1$.

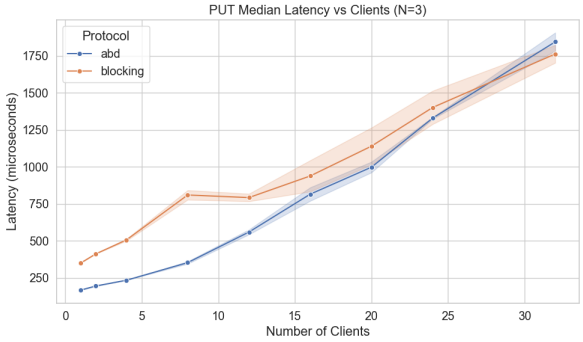

Figure 12: PUT median latency vs. clients, $N = 3$.

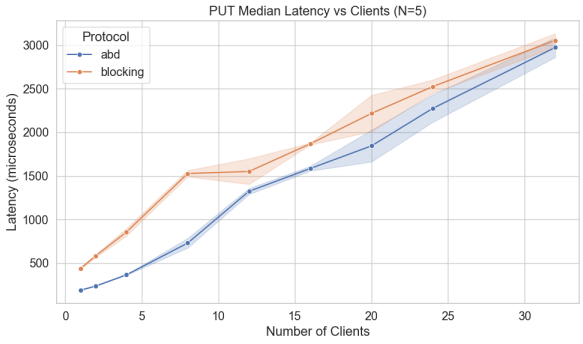

Figure 13: PUT median latency vs. clients, $N = 5$.

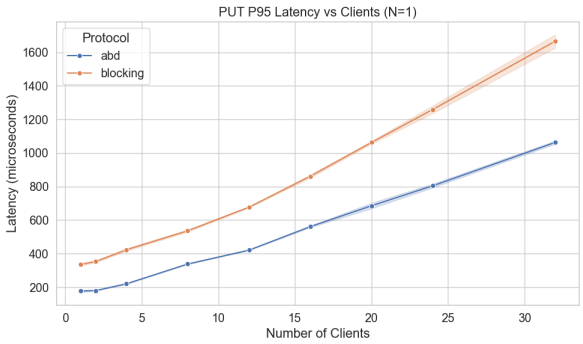

Figure 14: PUT 95th percentile latency vs. clients, $N = 1$.

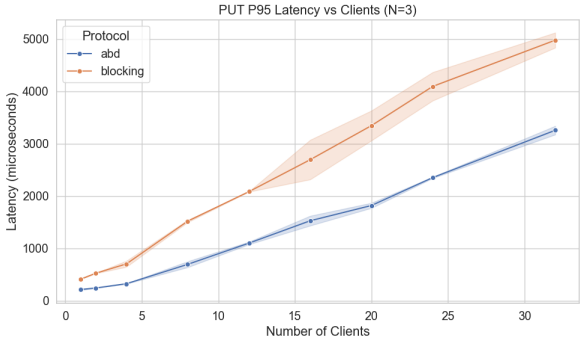

Figure 15: PUT 95th percentile latency vs. clients, $N = 3$.

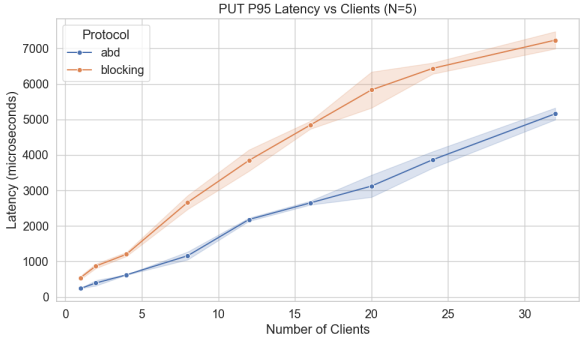

Figure 16: PUT 95th percentile latency vs. clients, $N = 5$.