Evaluating 2PL Lock Policies in a Distributed Key-Value Store

Rajavardhan Reddy Siriganagari
University of Utah
u1592037@umail.utah.edu

Jose Alexander Garcia
University of Utah
u0857155@umail.utah.edu

Siddharth Kapoor
University of Utah
u1592376@umail.utah.edu

Neha Bhat
University of Utah
u1592534@umail.utah.edu

***Abstract*—This work explores how alternative lock management and deadlock avoidance strategies affect performance in a distributed key-value store with Two-Phase Locking (2PL). We built a distributed transaction layer on top of a high-performance key-value store and began with a no-wait baseline, in which lock conflicts trigger immediate aborts. While being simple, the no-wait policy performs poorly under contention. To provide a richer comparison, we add two classical timestamp-based deadlock avoidance algorithms: wait-die and wound-wait, which allow transactions to wait selectively based on age. The client protocol stays the same to ensure a fair comparison. This system design supports a focused study of how different deadlock avoidance mechanisms influence throughput, abort behavior, and overall robustness in distributed transaction processing.**

***Index Terms*—Two-Phase Locking, Deadlock Avoidance Algorithms, Distributed Key-Value Transactions**

I. Introduction

Distributed transactional systems depend critically on effective lock management, especially when using Two-Phase Locking (2PL) to ensure serializability across multiple servers. However, many practical deployments rely on simplistic locking policies such as no-wait that immediately abort transactions upon encountering a lock conflict. While easy to implement, this approach often performs poorly under high contention, leading to excessive aborts and reduced throughput across different transaction workloads. The central problem is to understand whether more advanced deadlock avoidance strategies can improve performance and fairness without fundamentally changing the client-facing transaction API.

Our project addresses this problem by implementing and evaluating three classic 2PL lock management strategies: no-wait, wait-die, and wound-wait, within a distributed key-value store. The core idea is to extend the system with timestamp-based waiting mechanisms so that certain transactions may wait for conflicting lock acquisitions rather than being forced to abort immediately. By integrating these mechanisms, we aim to analyze how different lock management strategies influence throughput, abort rates, latency and fairness, particularly as workload contention and transaction length vary. This exploration provides insight into the practical trade-offs involved in deploying deadlock avoidance algorithms in distributed transactional systems.

We first describe each of the three deadlock avoidance lock management strategies considered in this project.

A. No-Wait

No-wait is a very simple 2PL deadlock avoiding wait strategy, where a transaction is aborted as soon as it tries to acquire a lock on a key which is already held by another transaction. As the name suggests, the transaction requesting the contested lock is not allowed to wait, and is aborted immediately. On abort, all the keys held by the transaction are immediately released, making them available for other transactions in the system. In essence, the invariant guaranteed by this strategy ensures that no transaction can wait on another transaction currently holding the lock on a key. The Waits-For Graph (WFG) of all running transactions in this case has no edges, i.e., no transaction is waiting on any other transaction. Since the graph has no edges, it also has no cycles (presence of cycles in the WFG indicate a deadlock).

B. Wait-Die

The wait-die locking strategy is a 2PL deadlock avoidance strategy where transactions are allowed to wait for a contested lock to be available, instead of being aborted immediately. Unlike no-wait, based on certain criteria, the transaction requesting a conflicting lock may be allowed to wait for the lock to be available and retry its acquisition.

Transactions in wait-die require to be timestamped. The wait-die invariant guarantees that a younger transaction may never wait on an older transaction, but an older transaction can wait for a lock held by a younger transaction. If a transaction attempts to acquire a conflicting lock, then the requester is aborted if it is younger than the current holder, otherwise, it is allowed to wait and joins a wait queue if it is older than the current holder. This prevents deadlocks as no cycles are feasible in the WFG. This can be proven by the fact that while traversing along any path in the WFG, the next node will have a strictly greater timestamp (younger) than the previous node (older). Hence, a cycle in the graph will violate this property.

C. Wound-Wait

In the wound-wait deadlock avoidance strategy, a younger transaction requesting a conflicting lock held by an older transaction will wait for the holder to release the key. An older transaction requesting a conflicting lock held by a younger transaction will wound (kill/abort) the holder. The wait direction in wound-wait is opposite of the wait-die rule. A cycle in the WFG violates the above invariant and hence

deadlocks are not possible.

II. Design Decisions and Rationale

A. Single Global Mutex

The server state in all three strategies is guarded by a single mutex. Even though finer-grained locking could improve parallelism, for example, the wound-wait protocol requires that multiple shared structures like the lock table, wait queues, transaction table be updated atomically. A single coarse mutex ensures that wounding, rechecking lock availability, updating wait queues, pre-empting transactions all occur without data races.

B. Wait Queue

For each key, waiting transactions are maintained in a priority queue ordered by the timestamp. The oldest waiting transaction is always the next eligible to acquire a lock. Selecting exactly one waiter avoids the thundering herd problem.

C. Wait-Die Specific Decisions

1) Unbounded Wait Prevention: The wait-die strategy guarantees that a transaction can only wait on a key if it is older than the current holder. Even though this mechanism prevents deadlocks from occurring, it does not guarantee a bounded wait time for the transaction. Unlike no-wait, where a transaction has a random chance of acquiring a lock or getting aborted, in wait-die an older transaction can join a wait queue and has to stay there until the conflict is resolved.

In our KV store, we allow two main operations, *Get* and *Put*. The *Get* operation tries to acquire a shared lock (S-lock) on a key and reads its value before returning. The *Put* operation tries to acquire an exclusive lock (X-lock) on a key and writes to it before returning. As their names suggest, S-locks allow non-exclusive access to a key, i.e., multiple transactions are allowed to read the key concurrently, and X-locks require exclusive access to a key, i.e., no other transactions can hold an S-lock or X-lock on the key while the current transaction is writing to it. This is required to maintain consistency and serializability of the KV store.

An unbounded wait can occur in a scenario where a younger transaction holds an S-lock on a key, and an older transaction tries to acquire an X-lock on the key and is pushed to the wait queue. While waiting, additional newer transactions may come in and acquire S-locks on the key, which prevents the old transaction from acquiring the X-lock, and hence, it remains stuck in the wait queue indefinitely. This scenario may occur in high contention workloads where reads are significantly more common and writes (an example is the YCSB-B workload with $\theta = 0.99$).

To prevent this situation, we added an additional check before a transaction is allowed to acquire an S-lock on a key. If a transaction requests an S-lock, the system first checks if there are any transactions pending in the wait queue. If there are waiting transactions, the system checks if the requester is older than all the waiters, and aborts the requester if it is younger than any waiter. Otherwise, it is granted the S-lock.

2) Additional Queue for Handling Lock Upgrades: As a consequence of only selecting the oldest waiting transaction in a wait queue, a deadlock can occur in wait-die when a transaction tries to upgrade its S-lock to an X-lock. The scenario is described as follows:

Transaction T_1 holds an S-lock on a key with multiple other transactions. An older transaction T_0 tries to acquire an X-lock on the key but is added to the wait queue as it conflicts with the current readers. T_1 proceeds and then tries to upgrade its S-lock to an X-lock. However, due to the other concurrent readers, it is also added to the wait queue behind T_0. Once the other readers release their locks, T_0 is notified since it's the oldest queued transaction. However, T_0 is unable to proceed as T_1 is still alive and still holds an S-lock on the key, and is added back to the wait queue ahead of T_1. This results in a deadlock, as T_0 is unable to acquire the lock and T_1 isn't notified as it is behind T_0 in the queue.

To work around such a scenario, we maintain an additional wait queue per key, which exclusively holds transactions waiting to upgrade their S-lock to an X-lock. This new queue holds a higher priority than the original wait queue, and transactions which are part of this upgrade lock queue are the first to be notified about the availability of a released key, before the normal wait queue.

D. Wound-Wait Specific Decisions

1) Prepared Transactions Cannot be Wounded: A transaction in the Prepare phase is immune to being wounded. This is necessary for 2PC correctness, once a participant votes "prepare", it must guarantee durability of its decision.

2) Early Lock Release on Pre-emption: When a transaction is wounded, the system immediately releases all of its locks instead of waiting for the *Abort* RPC call. Releasing locks eagerly allows other transactions to reattempt acquisition without delay, improving throughput under high conflict workloads.

3) Lazy Abort: When a transaction is wounded, its status is updated to reflect that it is aborted. The shard that marks the transaction as aborted will vote "No" in the Prepare phase and the transaction will eventually be aborted.

III. Implementation

Our implementation is available on Github [1].

A. Client

This section explains how the client implements distributed transactions, communicates with sharded servers, handles retries and collects performance statistics.

1) Workload Generation: The client supports multiple workloads like (YCSB-A/B/C) and a bank transfer workload. A producer goroutine generates random YCSB transactions and places them on a work queue. Multiple worker goroutines dequeue and execute them. This provides realistic concurrent load for testing high contention behaviour. The workload can be configured at startup by setting the parameter *workload*.

[1] https://github.com/siddkapo/cs6450-labs/tree/final-submission

2) Worker Abstraction: Each client thread executes transactions using a *Worker* object.

Listing 1. Worker class

```
class Worker {
    rpcClients
    workerID
    txID
    txActive
}
```

The number of *Worker* objects is configurable by setting parameter *threads* at startup. The client maintains one RPC connection per server. A *Worker* also maintains the state of the in-flight transaction.

3) Automatic Transaction Retry Logic: Every client thread repeatedly:

- Receives a transaction from the work queue
- Executes each operation (*Get/Put*)
- If any operation returns error, abort the transaction and retry it

4) Latency and Throughput Measurement: Each commit calculates a per-transaction latency. A dedicated aggregator goroutine collects all latencies and computes the average, median, 95th and 99th percentile values. This does not interfere with transaction execution.

B. Wait-Die

As mentioned in I-B, the wait-die invariant guarantees that no cycles can occur in the WFG of a system, as a transaction which is younger than the current lock holder of key is aborted immediately. To achieve this, we need to ensure that no cycles are present in the WFG every time it is modified, either when a transaction enters a wait queue, or when it releases it locks on commit or abort. Maintaining a WFG and checking for cycles adds additional complexity to the implementation. To avoid this, we adopted a simpler approach.

1) Deadlock Avoidance Algorithm: Instead of checking for cycles whenever the WFG changes, a check is added that instead aborts transactions if they are not the oldest in the wait queue of a key. Since our wait queue orders transactions by timestamp and only notifies the oldest transaction when a lock is released, all other transactions in the queue will violate the wait-die invariant and have to be aborted when the top-most transaction in the queue acquires the lock. This check proactively aborts transactions and effectively reduces the wait queues to a size of one. Although this approach significantly increases the abort rate of our wait-die implementation, its simplicity was a crucial factor in our final decision.

C. Wound-Wait

The lock manager maintains the set of readers, writers, and waiting transactions for each key. The lock state for each key is created lazily on first access.

1) Lock Acquisition: The lock acquisition logic is as follows:

- If the lock is compatible, it is granted and the transaction proceeds without waiting.
- If the acquisition fails, the system checks all conflicting lock holders. If the requesting transaction is older than a conflicting holder and that holder is not in the prepared state, the holder is wounded. Wounding sets the younger transaction's state to TxAborted. It leads to the release of all the locks held by it and triggers wake-ups of waiting transactions for those keys.
- After wounding all eligible transactions, the system checks lock compatibility again. This makes wound-wait non-blocking for older transactions, which is its key advantage over wait-die.
- If the transaction still cannot acquire the lock (because another older transaction holds it), it must wait.

2) Wait Queue Wake-Up: When a lock is released or when a wounded transaction is aborted, the system dequeues the oldest waiter which belongs to an active transaction and unblocks it. No goroutine can miss a wakeup and phantom waiters are safely ignored.

IV. Results

A. Metrics measured and its Analysis

The primary metrics measured were:

- Throughput(Commit Rate).
- Abort Rate (efficiency/wasted work).
- Latency (Average, Median, 95^{th} and 99^{th} Percentile).

Experiments were run across five dimensions:

- Load: 10, 20, and 30 client threads.
- Transaction Length: Short (3 operations) vs. Long (20 operations).
- Data Skew(θ): Varying access distribution ($theta$ 0.0 to 0.99).
- Scaling: 1, 2, and 3 shards.
- Workload Type: YCSB-A and YCSB-B.

Each of these experiments were run independently for each workload and wait policy, and repeated three times. The following default values were set for experiments which did not affect a particular parameter:

- Total client threads were limited to 10 by default.
- The default transaction length was 3 operations.
- The skew was set to 0.99 by default.
- The KV store was shared across 2 servers by default.

As seen in Fig. 1, for short transactions (3 operations), no-wait and wait-die perform similarly, with wound-wait lagging behind in throughput. In Fig. 2 for longer transactions (20 operations), the wound-wait policy is the most effective, as it has the least number of aborts, and its throughput being just slightly behind wait-die. No-wait and wait-die demonstrate a poor ability to handle longer transactions and write-heavy workloads where the probability of lock conflicts is much

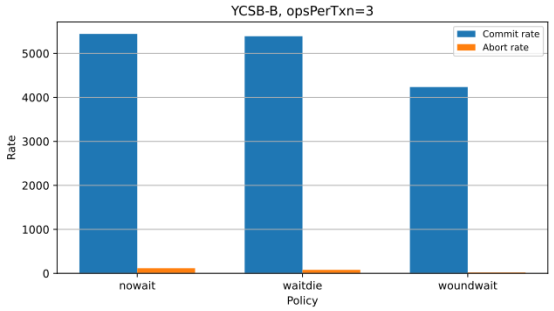

Fig. 1. Commit and abort rates by policy for short transactions

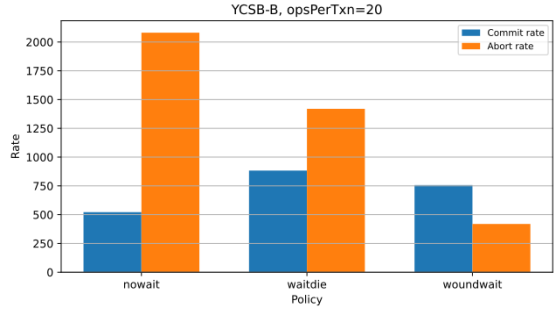

Fig. 2. Commit and abort rates by policy for long transactions

higher. Wound-wait has the lowest number of aborts in all scenarios.

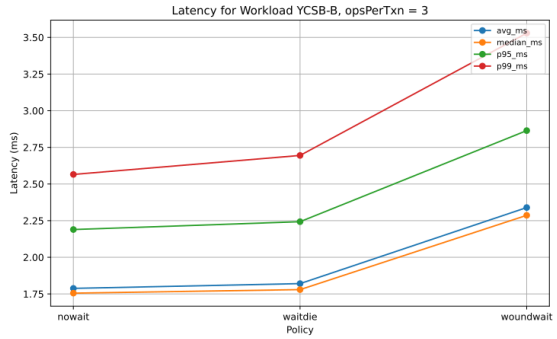

Fig. 3. Latency metrics by policy for short transactions

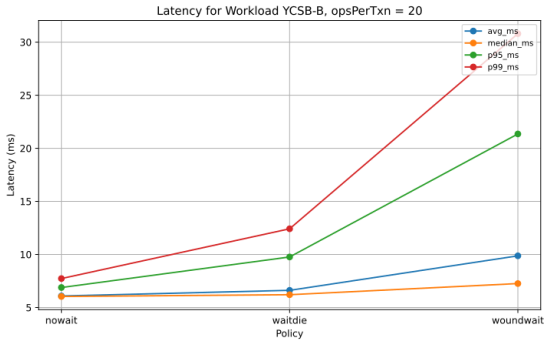

Fig. 4. Latency metrics by policy for long transactions

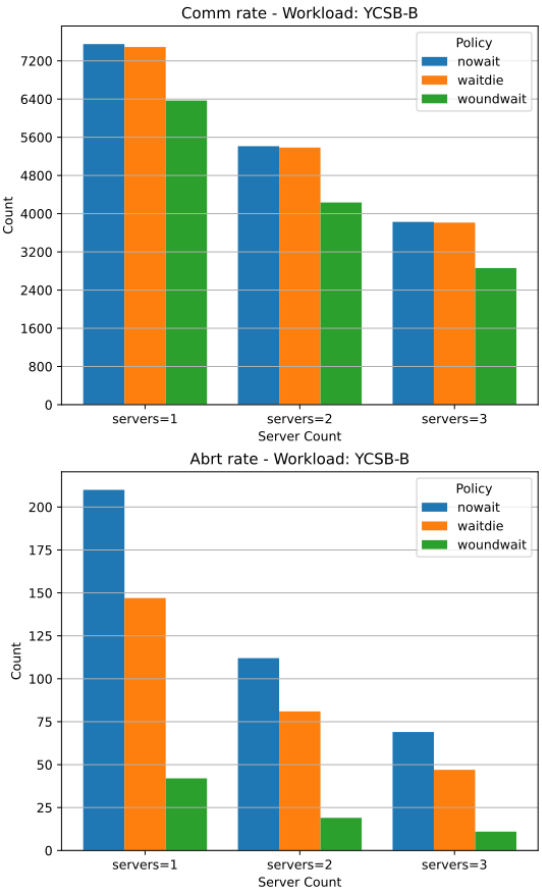

Fig. 5. Commit and abort rates by policy and sharding level

As seen in Fig. 3 and Fig. 4, in terms of latency, wait-die and no-wait perform almost identically in all scenarios, with wound-wait having slightly higher average and median latency in absolute terms, but significantly higher tail-end latency (p95 and p99). We observe similar results for YCSB-A for both throughput and latency metrics. If the primary goal is low responsiveness for the majority of transactions, no-wait is technically the ”fastest” policy, but this masks severe system inefficiency. If the primary goal is high throughput in a high-contention environment, wound-wait is superior. However, this comes at the cost of high tail-end latency, meaning a small portion of transactions will suffer long delays.

As observed in Fig. 5, the KV store has a diminished performance for all wait policies as we scale across multiple shards. This is true for all workloads. However, as noted earlier, wound-wait demonstrates the lowest abort rate for a small penalty on throughput.

In Fig. 6, as contention increases, the throughput remains relatively stable before crashing at $\theta = 0.99$, for wait-die and no-wait, while wound-wait shows a smaller drop. Consequently, the abort rate also increases significantly for no-wait and wait-die, while wound-wait manages contention

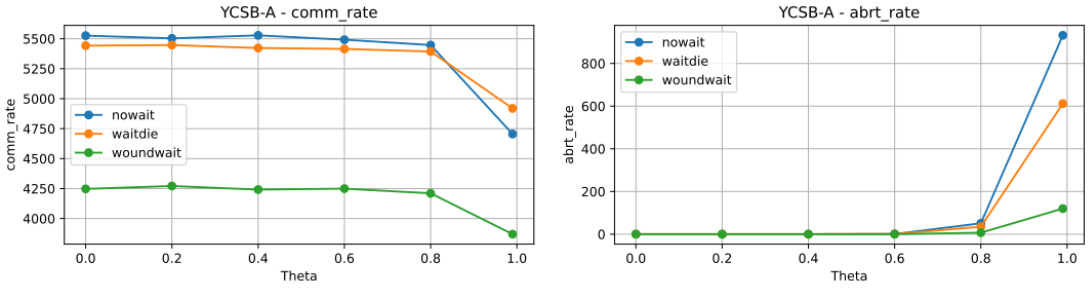

Fig. 6. Commit and abort rates by policy for different contention levels

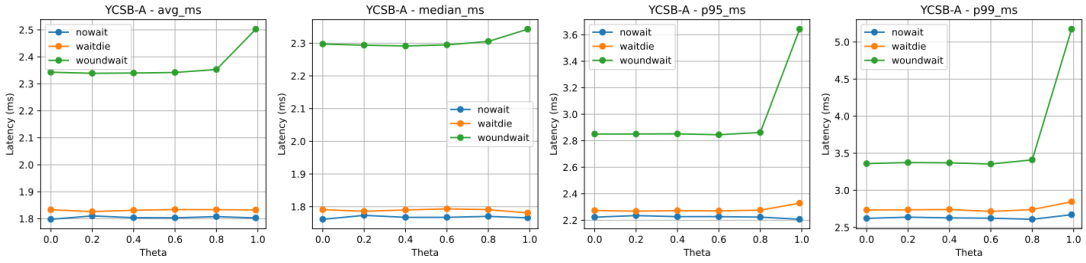

Fig. 7. Latency metrics by policy for different contention levels

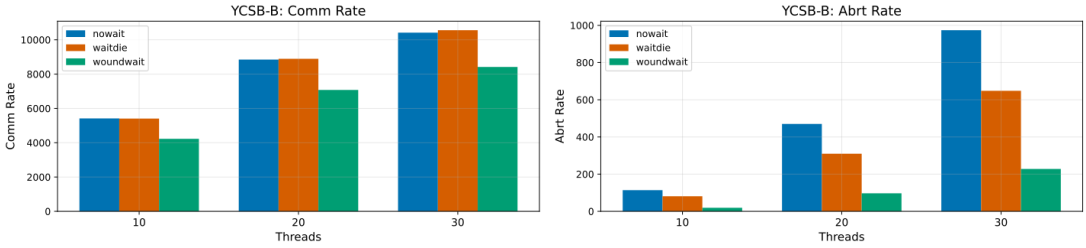

Fig. 8. Commit and abort rates for different client loads

better than the other two policies.

Fig. 7 shows that average and median latencies remain stable for all policies across all levels of contention, but wound-wait suffers from a notable increase in tail-end latency at higher contention levels.

Fig. 8 shows that throughput and aborts scale with the increase in client load across all three policies. Wound-wait has the lowest abort rate with a very small delta as the client load increases, unlike wait-die and no-wait. This demonstrates the higher efficiency of wound-wait for heavier workloads, with only a small penalty on throughput.

Overall, wound-wait demonstrates superior efficiency, stability and scalability for high contention and high load scenarios, with only a small penalty on overall throughput, but much higher tail-end latency. In scenarios with a more uniform key access pattern and read-heavy operations, wait-die and no-wait achieve significantly higher throughput and much lower latency than wound-wait.

B. Lessons Learned

- Implementing multiple 2PL locking strategies showed that even small changes in concurrency control can significantly affect system behavior. Even with the same client and transaction structure, no-wait, wait-die, and no-wait behaved very differently under contention showing how much lock semantics influence throughput and abort rates.
- We also learned that allowing transactions to wait adds complexity to the server. Supporting wait-die and no-wait forced us to change how the RPC handlers work so that lock requests could pause without blocking the entire server thread.
- The performance of each policy shifted noticeably depending on transaction length and workload skew. Short transactions saw little improvement from waiting while long transactions slowed down significantly under no-wait because they were repeatedly aborted. Keeping the client identical across all tests was important since it ensured that any changes in throughput, aborts, or latency came solely from the locking policy. These results gave

us a clearer understanding of how different lock management strategies behave under real workloads and why production systems often adopt more advanced deadlock avoidance algorithms rather than simple immediate abort approaches.

V. Comparison to Prior Work

The baseline PA2 system implements distributed transactions using Two-Phase Locking and Two-Phase Commit but relies exclusively on a no-wait policy. Under this approach, any lock conflict causes an immediate abort which avoids deadlocks but performs poorly under contention due to excessive retries and wasted work.

Our system extends PA2 by adding two deadlock avoidance algorithms wait-die and wound-wait which use timestamps to decide whether a transaction should wait or abort. Supporting these policies required redesigning the server side lock manager to maintain per key wait queues, evaluate transaction age, and delay RPC responses when locks are not immediately available. This introduces behavior that PA2 does not support, since the original system never allows waiting.

Overall, PA2 provides a clean but rigid locking model,whereas our extended system enables a more realistic exploration of distributed concurrency. By implementing no-wait, wait-die, and wound-wait under the same architecture, we can directly compare their impact on throughput, abort rates, latency, and fairness. These are insights that PA2 alone is not able to provide.

VI. Team Contribution

Rajavardhan implemented the wound-wait locking policy, including the server-side logic required to preempt younger transactions and the supporting changes to lock management. Alex contributed to the wait-die implementation and developed supporting scripts used for running experiments and collecting performance data. Siddharth also worked on the wait-die policy and assisted in building the automation scripts used to execute workloads under different locking strategies. Neha developed the statistics collection and reporting components, enabling measurement of throughput, abort rates, and latency across experiments and supporting the analysis presented in this report. All of us were involved in discussions and decisions to create the final design for each implementation.

References

[1] https://www.cs.emory.edu/~cheung/Courses/554/Syllabus/8-recv+serial/deadlock-waitdie.html

[2] https://www.cs.emory.edu/~cheung/Courses/554/Syllabus/8-recv+serial/deadlock-woundwait.html