# Nami: a Partitioned, Replicated, Transactional KV Store with Optimistic Concurrency Control

Lillian Ma
*Stanford University*

Juan M. Tamayo
*Stanford University*

## Abstract

Nami is a partitioned, replicated key value store with support for optimistic transactions. It uses RocksDB for durable storage, and Ratis (an open source Raft implementation) for consensus and log replication. In Nami, a transaction is committed to the distributed log before determining whether it conflicts with previous transactions. Nodes ignore transactions irrelevant to their data. When a transaction is relevant, each node independently determines whether it should be applied, potentially querying other nodes that have the data needed. Our hope with this approach is to provide higher throughput and reliability than a single RocksDB instance, while providing useful transactional semantics.

## 1 Introduction

Google's MapReduce [6] paper was published in 2004. Two years later, Hadoop's MapReduce [1] was made available, allowing many engineers and institutions across the world to easily process and generate large data sets. Unlike MapReduce, twelve years after the publication of Spanner's paper, there is no widely adopted distributed SQL database.

One reason for this discrepancy is the difficulty in implementing two-phase commit across a distributed dataset. In this paper we explore a different approach to distributed transactions, inspired by Calvin [9] and FaunaDB [5].

In this approach, every transaction is committed to a single global distributed log before it is processed by various partitions in the dataset. The API and data model are designed such that any node with access to the distributed log can determine whether a transaction conflicts with other transactions that came before it.

Although it may seem counterintuitive to put every transaction in the same log, implementations like Ratis [2] and Kafka [3] are known to saturate disk and network bandwidth, which can be much higher than the actual write bandwidth of a transactional workflow.

## 2 Design

### 2.1 Data model

Nami is a key-value store. There is a single key space. Keys are restricted to printable ASCII strings of up to 127 characters; values are arbitrary byte arrays. In Nami, every write is part of a transaction. Transactions are assigned a monotonically increasing `tid` at commit time. Every row is annotated with the `tid` of the transaction that wrote it. The system keeps track of multiple versions of each row for concurrency control.

### 2.2 API

In Nami, all operations are part of a transaction. Transactions are snapshot isolated: when a transaction begins, the system picks a `snapshot_tid` as of which all reads will be executed.

Because of our architecture choices, we distinguish between the fat client API used by applications and the server API used by the fat client.

#### 2.2.1 Fat client API

An application using Nami can perform the following operations:

**begin(min_tid): ClientTx** Starts a new snapshot isolated transaction. The fat client picks a snapshot tid as of which all reads will be executed. The snapshot is guaranteed to be after `min_tid`. Returns a `ClientTx` object.

**ClientTx.get(key): value** Fetches the value stored for the given key. The `<key, value>` pair is stored in memory to detect conflicts at commit time.

**ClientTx.put(key, value)** Stores a `<key, value>` pair in memory so it is written at commit time.

**ClientTx.commit()** Submits the transaction to the Raft leader. Returns the `tid` assigned to the transaction, and its *transaction outcome* (see 2.3): `APPLIED` if the transaction was applied to the KV store, or `CONFLICT_IGNORED` if any of the values read has changed since `snapshot_tid` and the transaction was discarded. Applications are expected to retry in case of conflicts.

### 2.2.2 Server API

The server API is designed to support the fat client. It provides the following endpoints:

**getRecentTid(): tid** Returns the most recent `tid` seen by this server.

**getAsOf(tid, key): value** Returns the value of the given `key` as of `tid`. Server will block until it has caught up to `tid`.

**commit(transaction): <tid, TxOutcome>** Attempts to commit a transaction. Returns the transaction's assigned `tid` and its outcome (see 2.3).

## 2.3 Implementing transactions

One of Nami's distinguishing features is the support for ACID transactions across a distributed dataset. It does so using an approach inspired by Calvin [9] and FaunaDB [5].

When the leader receives a Transaction, it commits it to the Raft log without determining whether the transaction conflicts with any prior ones. Every transaction is assigned a `tid`, which is simply the index of the log entry that represents the transaction in the replicated Raft log. This establishes sequential order for all transactions. Once a transaction is "raft-committed", it is durable and will eventually be processed by the system.

Notice, however, that a "raft-committed" transaction might conflict with prior transactions, and thus should be ignored. Nodes in the system must decide the outcome of every transaction that is relevant to them, and any two nodes must always come to the same conclusion. We call this decision the *transaction outcome*.

To understand how any nodes can determine a transaction's outcome, consider transaction `T` with assigned transaction id `tid`; let `prior_tid` be the id of the transaction right before `T`. When a node participating in the Raft protocol determines that `T` is safe to apply to its state machine, it reads the values for all keys listed in `T.in_tx_gets` as of `prior_tid`. Some of these values might be available in its own partition of the data, but some of them might require RPCs to other nodes. These RPCs must wait until the remote nodes have processed `prior_tid`. Once all values are collected, the node compares them with the values included in `T.in_tx_gets`, which were read as of `T.snapshot_tid`. If any discrepancy is found, the transaction outcome is `CONFLICT_IGNORED`. If all values match, the transaction outcome is `APPLIED` and any writes in `T.in_tx_puts` are applied.

Applying transactions efficiently is critical to the write performance of the system. Nami implements two optimizations to reduce the time it takes to determine transaction outcome. First,

1. If none of the keys in `T.in_tx_puts` are stored in a node, the transaction is ignored.

2. The strict semantics around commit allows each node to cache recently committed values. With this *commit cache* (see 2.3.1) we can determine most outcomes locally.

For simplicity, the Raft leader always determines the transaction outcome and sends it back to the client. That being said, any system with access to the raft log could determine the outcome of a transaction, which could alleviate the load on the leader.

### 2.3.1 Commit cache

To reduce the overhead of remote communication between nodes when determining the outcome of a transaction, we created a caching layer on top of the KV store. This cache tracks a sliding window of the last 10k transactions successfully committed on node (configurable), storing only the writes in those transactions that don't already live in the local KV store. The structure of the cache consists of an "index" mapping a non-versioned key to its most recent `tid` that committed it, and a "cache", mapping a `tid` to a list of all the non-local writes consisting of non-versioned keys and their value. This allows for quick lookup when fetching values given a non-versioned key. Note that this cache only contains transaction writes that the node had to process. Typically this means that the transaction had local writes or the node was the leader. However, if the node also has all of the reads in the cache, it processes the transaction anyway and writes the non-local writes to its caching store, as this has a negligible performance cost but could save a future remote lookup. In the case where the node skips a transaction, it will invalidate all the writes in that transaction in the caching layer by removing all the relevant entries. This ensures that the cache store will always have the most update-to-date transaction data processed by the node, if they exist.

Eviction occurs when the size of the transactions in the store reaches the configured threshold. It is done by tracking the oldest possible `tid` in the cache and locating that entry for removal, if it exists. This id will be continuously incremented until an entry is found, and incremented again once all the relevant data for that entry is removed.

By enabling this caching layer, we hope to dramatically reduce the number of remote data fetches, as we don't expect too many clients to keep transactions open long enough where more than 10k transactions would have passed between the `snapshot_tid` it used to start the transaction and the time it commits.

#### 2.3.2 Choosing a `snapshot_tid`

When starting a transaction, the fat client need to determine the `snapshot_tid` it'll use for reading data. Choosing one is a tradeoff between recency and latency. Picking the most recent snapshot seen by the raft leader is likely to cause higher latency, as some of the replicas may not have caught up; picking an old snapshot may cause conflicts at commit time. We solve this by picking the most recent snapshot from a random server.

This approach could cause transactions to "go back in time" to an earlier snapshot. Applications that need "read-my-writes" consistency can defend against this by keeping track of the largest `tid` returned by the server and using it when starting a transaction, or by retrieving the latest committed index from the raft group.

## 3 Implementation

### 3.1 Software stack

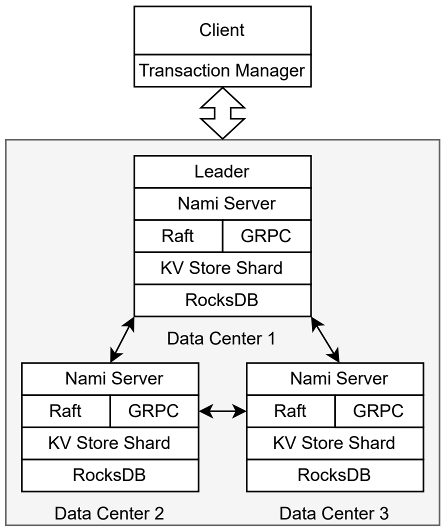

Figure 1: A Nami cluster and client application.

This section describes the structure of Nami and outlines the components and their interactions required for distributed transactions and data replication. Figure 1 shows the software stack. At the bottom, RocksDB is used for durable storage. Each RocksDB instance contains a single table, mapping versioned keys to values. The KV Store is responsible for tracking the latest `tid` that committed, as well as reading and writing data. It supports fetching data at an exact version, waiting for the data store to reach a specific version, and getting the latest version of the data or as of a version. It also allows clients to batch put and force physical write of data to disk (flush). Additionally, the KV store is aware of its own data partition.

To support transaction log replication, each Nami server is part of a Raft group and runs a Raft server responsible for log entry replication, voting, and leader election using Apache Ratis [2]. The server's `commit()` API is served through Ratis. Each server also implements a state machine that will determine whether a node will apply the changes in the transaction. For more details, see 2.3.

Read APIs, including `getRecentTid` and `getAsOf` are exposed through a separate gRPC server. This gRPC server is also used to fetch remote data from peers for transaction processing. Each Nami server is aware of how data is partitioned between peers, and chooses an available peer owning the data at random to query all the available data it might have in a batch request. In the case where the peer is non-responsive, it will be marked as unavailable based on the gRPC status and the server will retry on a different node. If no available nodes contain a requested key, transaction processing aborts. Additionally, each server can serve clients directly for snapshot reads and `snapshot_tid` queries required during the lifetime of a transaction.

## 3.2 Data partition and placement

Data partitioning is done by assigning a set of evenly distributed numerical ranges to each node in a configuration. The min and max of the range are 0 and $2^{15} - 1$ respectively. To determine whether a node owns a key, it will convert the first two bytes of the primary key to a number in that min/max range, then check to see if this number falls in any of the ranges that it is assigned. Data replication is done by assigning the same numerical range to multiple nodes. Note that this configuration is static and must be set manually based on the initial number of nodes in the cluster. Re-sharding data once the cluster goes live is not supported.

## 3.3 Client interaction

We provide a "fat" client that tracks the internal state of a transaction and submits it to the leader when the user commits. It consists of two parts: 1. The client responsible for submitting requests to the Raft server or gRPC server, 2. The transaction manager responsible for tracking all the reads and writes values over the course of the transaction. When a user starts a transaction, the transaction manager will either assign a `snapshot_tid` (see 2.3.2), or take one requested by the caller as the `tid` to use for all its reads. The client will select a peer at random to query for a `snapshot_tid` through the gRPC server. Similarly, snapshot reads will be assigned to any arbitrary peer that has the data, and may incur a delay if the peer has not yet caught up. The transaction manager also allows users to read values that they've written in the same transaction. When commit is called, all reads and writes values along with the `snapshot_tid` are submitted through Raft for processing.

## 3.4 Crash recovery

To support resuming after failures, snapshots of the Raft log are periodically taken along with its last log index, term and associated `tid`. This is done on each node every 1024 entries and is persisted to disk. Atomically, the RocksDB table is also force flushed to the disk from its in-memory buffers. When a failover occurs and the node recovers, it will restore its Raft log to the last snapshot, and restart RocksDB from the same storage files. It will also assign the KV Store the last processed `tid` to the one in the snapshot. This means that transactions applied after the last snapshot was taken will be re-applied to the KV Store when the node catches up. However, given that applying transactions in the same log order is idempotent, there is only a performance cost and can be tuned based on how frequently the snapshots are taken. The upside is that snapshots taken incrementally are lightweight and should only cause a small amount of delay in transaction processing.

## 4 Evaluation

We begin by describing a simple transactional application used to test the system. We then describe various benchmarks used to evaluate latency, throughput, and availability. All benchmarks were executed on a GCP cluster of up to 6 e2-medium machines with 2 vCPU and 4 GB of memory.

## 4.1 Banking app

Evaluation was performed using a simplified *Banking app*. The app stores "customer accounts" with a balance and description.

At startup, the app creates 30k accounts with zero balance and a random description. It then proceeds to execute *money movement transactions* on several worker threads. Each money movement transaction picks pairs of accounts at random and moves a random amount between them. After all workers finish, the app validates that the net balance across all accounts is still zero.

Description length was chosen such that each row is 1 KB in size. Each transaction executes 5 movements affecting 10 accounts for a total of 10 KB in transaction size. We run 400 transactions per worker for a runtime of about two minutes. The number of workers varies per test. Note that the amount of data read and written on each transaction is identical.

## 4.2 Minimum latency

To evaluate best case latency, we ran the Banking app with one worker against two different cluster setups. The results are presented in Table 1.

| | 3 Nodes | | | 5 Nodes | | |
|---|---|---|---|---|---|---|
| Operation | p50 | p95 | p99 | p50 | p95 | p99 |
| `moveMoney` | 8.17 | 15.2 | 21.9 | 8.76 | 14.9 | 20.5 |
| `commit` | 4.83 | 9.86 | 13.1 | 5.28 | 9.99 | 4.19 |
| `getAsOf` | 0.95 | 2.66 | 4.22 | 1.05 | 2.43 | 15.9 |

Table 1: Latency in `ms` seen by the client under optimal conditions. On the left is a 3 node, no replication cluster; on the right a 5 node, 2-way replicated cluster.

## 4.3 Availability

To evaluate Nami's resilience to node failures, we set up a 5 node cluster with 2 way replication for each chunk of data. Data was placed such that, when any one node goes down, queries to randomly chosen keys are distributed evenly across the remaining nodes. Figure 2 shows the results of this experiment.

Two facts are noticeable from this chart. First, throughput slows down slightly when a node is killed, but quickly recovers. This is expected, as some of the requests time out and the system elects a new leader. The system quickly recovers. Second, transaction processing halts as soon as the killed node is brought back up. This is an unfortunate bug in our implementation where a node advertises itself as available before it has caught up with the replicated log. Any reads sent to that node are forced to wait until the killed node catches up.

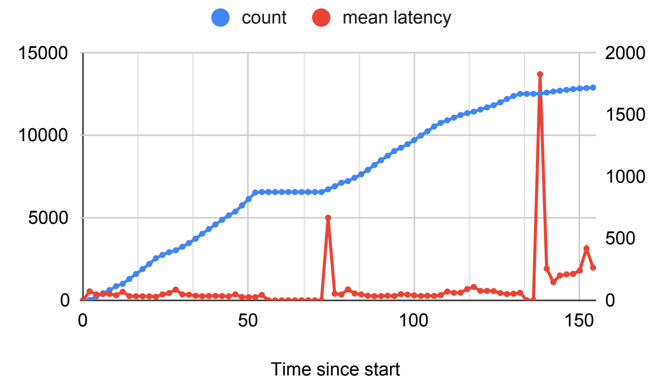

Figure 2: Cumulative number of money movements performed by the Banking app in the face of node failures. In the experiment, the Raft leader is killed 25 seconds into the benchmark, and restarted 25 seconds later. A follower is killed 110 seconds into the benchmark, and restarted 20 seconds later.

## 4.4 Comparison with RocksDB transactions

To establish a baseline comparison, we created a single node RocksDB server and fat client that implement the same API as Nami.

The server provides similar transaction APIs as Nami, including `begin`, `commit`, `getForUpdate`, and `put`. Internally, it delegates the transaction processing to RocksDB's `OptimisticTransactionDB` [7], which uses its in-memory buffer for conflict detection at commit time. The fat client implements the same transaction APIs as Nami, allowing the client to switch DB implementations transparently.

Table 2 shows the results of this experiment. With no load, Nami is faster than native RocksDB, which we attribute to distributed load across servers. As load increases, though, Nami saturates more quickly, supporting fewer worker threads.

Unexpectedly, both Native RocksDB and Nami plateau at around 130 ops/s. This number was consistent across other testing in the same VMs. We suspect there is a limit common to both setups, but are uncertain as to what it is.

## 4.5 Impact of the commit cache

To evaluate the effect of the caching store on commit latency, we ran our banking app benchmark test on a

| | Native RocksDB | | Nami 3 nodes | | Nami 5 nodes | |
|---|---|---|---|---|---|---|
| Workers | Throughput | Latency | Throughput | Latency | Throughput | Latency |
| 1 | 37.9 | 20.1 | 46.3 | 16.5 | 40.0 | 17.9 |
| 5 | 96.4 | 23.0 | 119 | 21.4 | 85.0 | 24.6 |
| 10 | 116 | 29.5 | 131 | 37.7 | 92.8 | 44.6 |
| 20 | 121 | 49.0 | 131 | 82.4 | 94.8 | 85.3 |
| 30 | 118 | 71.2 | 132 | 118 | 94.4 | 138 |
| 40 | 133 | 87.6 | 119 | 154 | 98.5 | 183 |

Table 2: Comparison between the Native RocksDB baseline; a Nami 3 node, no replication cluster; and a Nami 5 node, 2-way replicated cluster. Mean throughput and p50 latency in ms are reported.

5 node cluster with 2 way data replication. The workflow used is as described in section 4.1. Table 3 shows the latency and throughput by percentiles for runs with the caching store enabled and without. With caching enabled, the median in latency saw a +10% reduction, while the p95 saw +35% reduction. These improvements are likely due to fetching data from the cache instead of remotely during transaction processing. p99, however, was worse by ∼29% when caching is enabled, and we attribute this degradation to possible slow cache eviction and invalidation when garbage collection is required. Throughput also improved slightly (∼4.4%), as more requests can be served per second due to faster transaction processing enabled by the cache. The effects of the store may be more dramatic in slower networks as the average ping between machines in our test cluster is ∼0.307ms.

| | Throughput | Latency | | |
|---|---|---|---|---|
| | | p50 | p95 | p99 |
| With cache | 109 | 46.0 | 132 | 305 |
| Without cache | 104 | 50.9 | 179 | 237 |

Table 3: Impact of the commit cache. We report mean throughput (in `ops/s`) and latency percentiles (in `ms`) as seen by the banking app.

## 5 Related work

FaunaDB [5] uses a similar technique for distributed transactions, where a replicated, distributed log (using Raft) is used to achieve consensus between replicas instead of 2PC. The position of the transaction in the log specifies its equivalent serial order relative to all other transactions being processed by the system. This system guarantees ACID transactions, and linearizable, consistent operations across replicas.

Calvin [9], like FaunaDB, is a geographically replicated, ACID compliant transaction database and also a distributed, replicated log for transactions before they are processed. Clients submit transactions to the preprocessing layer of their local region, which then submits these transactions to the global log via a cross-region consensus process like Paxos. Like FaunaDB and Nami, the order of the log dictates the actual transaction ordering.

Spanner [4], also a geographically replicated, ACID compliant transaction database, uses TrueTime for transaction ordering and requires write locks on all write data (due to 2PC), but no locks for read-only transactions.

CockroachDB [8], similar to Spanner except without TrueTime, will choose its transaction timestamps based on the maximum commit timestamp across all nodes it intends to read from. It has much looser clock synchronization requirements than Spanner and as a result may sometimes retry reads.

## 6 Conclusion

Our work shows that with careful selection of APIs, it is relatively easy to build a distributed transactional system with ACID semantics using a distributed transaction queue. The avoidance of 2PC and write locks have greatly simplified our implementation.

More work is required to understand whether this approach can achieve reasonable performance and horizontal scalability.

## References

[1] Apache Hadoop releases. https://archive.apache.org/dist/hadoop/common/?C=M;O=A, 2006. [Online; accessed 2024-06-05].

[2] High throughput data replication over RAFT. https://www.slideshare.net/slideshow/high-throughput-data-replication-over-raft/102689036, 2018. [Online; accessed 2024-06-05].

[3] Apache Kafka Performance. https://developer.confluent.io/learn/kafka-performance/, 2024. [Online; accessed 2024-06-05].

[4] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), aug 2013.

[5] Matt Freels Daniel Abadi. Consistency without Clocks: The Fauna Distributed Transaction Protocol. https://fauna.com/blog/consistency-without-clocks-faunadb-transaction-protocol, 2018. [Online; accessed 2024-06-05].

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[7] RocksDB maintainers. Transactions. https://github.com/facebook/rocksdb/wiki/Transactions, 2024. [Online; accessed 2024-06-05].

[8] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pages 1–12, 2012.