Warp: Multi-Key Transactions for Key-Value Stores

Robert Escriva[†], Bernard Wong[‡], Emin Gün Sirer[†]
[†] *Computer Science Department, Cornell University*
[‡] *Cheriton School of Computer Science, University of Waterloo*

Abstract

Implementing ACID transactions has been a long-standing challenge for NoSQL systems. Because these systems are based on a sharded architecture, transactions necessarily require coordination across multiple servers. Past work in this space has relied either on heavyweight protocols such as Paxos or clock synchronization for this coordination.

This paper presents a novel protocol for coordinating distributed transactions with ACID semantics on top of a sharded data store. Called linear transactions, this protocol achieves scalability by distributing the coordination task to only those servers that hold relevant data for each transaction. It achieves high performance by serializing only those transactions whose concurrent execution could potentially yield a violation of ACID semantics. Finally, it naturally integrates chain-replication and can thus tolerate faults of both clients and servers. We have fully implemented linear transactions in a commercially available data store. Experiments show that the throughput of this system achieves 1-9× more throughput than MongoDB, Cassandra and HyperDex on the Yahoo! Cloud Serving Benchmark, even though none of the latter systems provide transactional guarantees.

1 Introduction

Big Data needs have catalyzed radical change in the architecture of data stores over the last decade. A new class of data stores, called NoSQL systems, have emerged to meet the performance and scalability challenges posed by large data. While the precise definition of NoSQL systems remains mired in controversy, the defining characteristic of these systems seems to be their distributed architecture, where the data is sharded across all hosts in the cluster. And it is partly because of this distributed architecture that has NoSQL systems have found it difficult to support ACID transactions.

Distributed transactions are inherently difficult, because they require coordination among multiple servers. In traditional RDBMSs, transaction managers coordinate the clients and servers, and ensure that all participants in multi-phase commit protocols run in lock-step. Such transaction managers constitute bottlenecks, and modern NoSQL systems have eschewed them for more distributed implementations. Scatter [20] and Google's Megastore [5] map the data to different Paxos groups based on their key, thereby gaining scalability, but incur the latency of Paxos. An alternative approach that incurs comparable costs, pursued in Calvin, is to use a consensus protocol and deterministic execution to determine an order, though Calvin uses batching to improve throughput at further latency cost. Most recent work in this space, Google's Spanner [13], relies on tight clock synchronization to determine when an operation is safe to commit. While these systems are well-suited for the particular domains they were designed, a completely asynchronous, low-latency transaction management protocol, in line with the fully distributed NoSQL architecture, has not yet emerged.

This paper introduces Warp, a NoSQL system that provides support for efficient, one-copy serializable ACID transactions. Warp's design combines optimistic client-side execution with a novel server-side commit protocol called *linear transactions*. In line with the NoSQL design philosophy, linear transactions involve solely those servers that hold the data affected by a transaction, and eliminate the need for transaction managers and clock synchrony. The coordination among these servers is performed by a modified single-pass chaining protocol that is fault-tolerant, non-blocking, and serializable.

Three techniques, working in concert, shape the design of linear transactions and account for its advantages. First, linear transactions arrange the servers in dynamically-determined chains, where transaction processing is performed in an efficient two-way pipeline. Traditional consensus protocols, such as Paxos and Zab, require a designated server to perform a broad-

cast followed by a quorum-incast, which divides overall throughput by the number of servers involved. In contrast, each server involved in a linear transaction can pump messages through the pipeline at line rate. This improvement comes at the cost of a potential increase in latency, a cost we address with careful client implementation, and quantify with microbenchmarks.

Second, linear transactions further reduce transaction overheads by not explicitly ordering concurrent but independent operations with respect to each other. Traditional approaches to transaction management compute a total order on *all* transactions, which necessitates costly global coordination. Such over-synchronization is a significant source of inefficiency, which some systems target by partitioning the consensus groups into smaller units [5, 20]. In contrast, linear transactions leave unordered the operations belonging to disjoint, independent transactions. This enables the servers to execute these operations in natural arrival order, saving synchronization and ordering overhead, without leading to any client-observable violations of one-copy serializability. Linear transactions determine a partial order between all pairs of overlapping transactions that have data items in common, and also detect and order transitively interfering transactions, thereby ensuring that the global timeline is always well-behaved.

Finally, linear transactions improve performance by taking advantage of the natural ordering imposed by the underlying data store. Specifically, they avoid computing a partial order between old transactions whose effects are completely reflected in the data store, and new transactions that cannot have observed any state of the system prior to fully committed transactions. Traditional approaches, especially those that involve Paxos state machines, would require the assignment of an explicit time slot, and perhaps couple it with garbage collection. In contrast, linear transactions can avoid these overheads because the happens-before relationship is inherently reflected in the state of the store and no reordering can lead to a consistency violation.

It is impossible to achieve ACID guarantees without a consensus protocol or synchronicity assumptions, and linear transactions are no exception. Our approach relies on a replicated state machine called a coordinator to establish the membership of the servers in the cluster, as well as the mapping of key ranges to servers. A crucial distinction from past work that invoked consensus on the data path, however, is that linear transactions involve this heavy-weight consensus component only in response to failures.

Overall, this combines these insights, which have been observed independently in past work, into a cohesive, unique transaction system, and makes three contributions. First, we outline a novel protocol for providing efficient, one-copy serializable transactions on a distributed, sharded data store. The protocol can withstand up to a user-specified threshold of faults, guarantees atomicity and provides isolation. Second, we describe our implementation of the Warp key-value store, including the design of the client. The system has been fully implemented, supports C/C++, Python, Java, and Ruby bindings, and has been deployed in production at one large company and a few startups. Third, we report macro- and microbenchmarks that examine the performance characteristics of linear transactions. Our experiments with the YCSB macrobenchmark compare Warp to popular high-performance NoSQL data stores, specifically MongoDB, Cassandra and HyperDex, and show that it achieves throughput that 1-9× that of other systems.

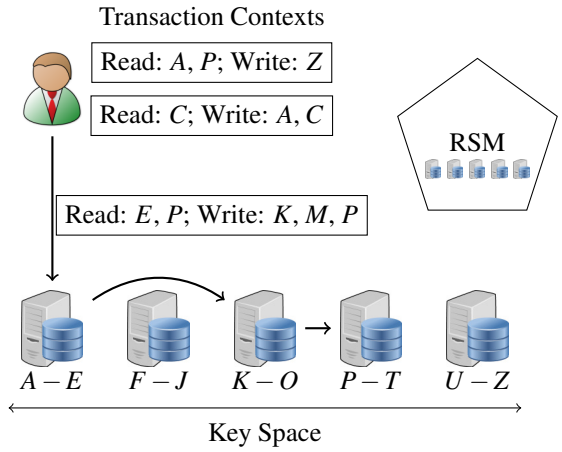

Figure 1: The architecture of Warp. A replicated state machine (RSM) maintains metastate about cluster membership and the mapping from keys to servers. Storage servers are assigned partitions of the key-space by the RSM. Clients communicate with Warp through a client library, which transparently retrieves the mapping from the RSM, maintains a cached copy of the mapping, and contacts the storage servers to issue operations.

To our knowledge, Warp is the first asynchronous, fault-tolerant, fully distributed key-value store that supports multi-key transactions without a shared consensus component on the data path. It represents a new design point in the continuum between NoSQL systems and traditional RDBMSs.

The rest of this paper is organized as follows. Section 2 describes linear transactions protocol for NoSQL systems. Section 3 describes our full implementation of Warp. Section 4 evaluates the performance of Warp. Section 5 surveys existing systems and provides context and Section 6 concludes.

```
class WarpClient:
    def get(table, key):
        # return value
    def put(table, key, value):
        # store value
    def cond_put(table, key, check, value):
        # store value if and only if check
    def begin_transaction(table, key):
        return WarpTransaction()
```

(a) Standard Interface

```
class WarpTransaction:
    def get(table, key):
        # return value
    def put(table, key, value):
        # store value
    def cond_put(table, key, check, value):
        # store value if and only if check
    def commit(): # commit the transaction
    def abort(): # abort the transaction
```

(b) Transactional Interface

Figure 2: A subset of the Warp API that illustrates the core operations provided by this system. The full API permits a wide range of atomic operations that are separate from the API presented here. The non-transactional and transactional APIs intentionally present the same set of operations.

2 Linear Transactions

Warp builds on top of a linearizable NoSQL store [3, 18, 24, 36]. Our design keeps the core architecture of the system relatively unchanged, integrating the transaction processing directly into the storage servers rather than introducing additional components dedicated to processing transactions.

The Warp system comprises three components. The first and primary component is a data storage server. Each data server is responsible for a subset of keys in the system, generally chosen using consistent hashing [27]. Collectively, the storage servers hold all the data stored in the system. The data is sharded across servers so that each server is responsible for a fraction of the systems' data. While each data server is $f+1$ replicated to provide fault-tolerance for node failures and partitions that affect less than a user-defined threshold of faults, for simplicity, we treat each data server as a singular entity throughout the early discussion in this paper. To further simplify the discussion, we assume that all clients issue solely read and write operations and not Warp's complex operations.

A second logical component called a coordinator partitions the key space across all data servers, ensuring balanced key distribution and facilitating membership changes as servers leave and join the cluster. Since the coordinator is not on the data path, its implementation is not critical for the operation of linear transactions. Many NoSQL systems centralize this functionality at a single operations console, backed by a human administrator; our implementation relies on a replicated state machine [4] that maintains the set of live hosts, the key partitioning table and an epoch identifier in a replicated, fault-tolerant object known as a *mapping*.

The third class of components, the clients, issue requests to the data servers with the aid of this mapping. Since the mapping is pushed to all non-disconnected servers by the coordinator after every configuration change, and since every client request and server response carries the epoch id, out of date clients and servers can be detected and directed to re-fetch the mapping when necessary. We defer a full discussion of partitions and focus first on the general operation of linear transactions.

Clients issue operations, both directly to the data store, and indirectly within the context of a transaction. Non-transactional requests identify the object to store or retrieve using a single key, and immediately perform the request against the relevant back-end storage server. Alternatively, a client may begin a transaction, which creates a transaction context, and issue several operations within the context of the transaction. Operations executed within the transaction do not take place on the servers immediately. Instead, the client library will log the key and type of each access. For a read, the client will retrieve the requested data from the storage servers, and record the value it read in a cache kept within the transaction context. Subsequent reads within that transaction are satisfied from this cache, providing read isolation. For a write, the client stores all modifications locally within the transaction context without contacting any storage server. Multiple writes to the same key will overwrite the stored modifications table. At commit time, the client library submits the set of all read keys, their read values and all modified unique key value pairs to the storage servers as a single entity, known as a linear transaction. The data servers, collectively, will only commit the modifications if none of the values read within the transaction context have been modified while the transaction was being processed.

Figure 1 shows Warp's overall system architecture. In this example deployment, data is sharded across five storage servers. The replicated state machine locally maintains this mapping. Each server fetches a copy of the mapping and maintains contact with the RSM to be notified of updates. A client may perform transactions by directly contacting the storage servers. The arrows indicate the communication necessary for a linear transaction involving the indicated servers.

For the rest of this section, we'll examine a simpli-

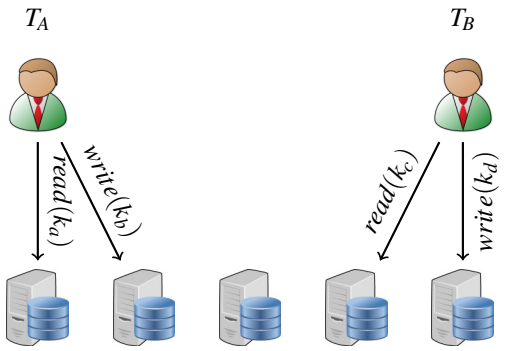

Figure 3: An example of two disjoint transactions. The clients read and write to entirely disjoint sets of keys.

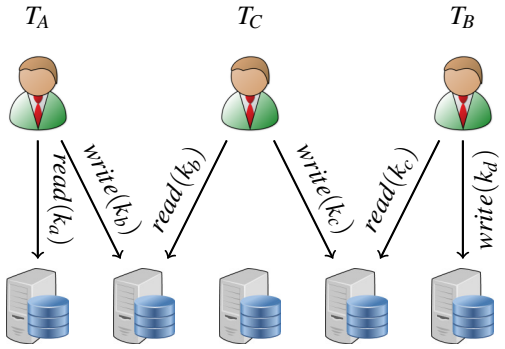

Figure 4: An example of three overlapping transactions. Transaction T_3 overlaps with T_1 and T_2, making all transactions overlap.

fied subset of Warp's actual transaction API, shown in Figure 2. This API captures the essential components of the interface to the NoSQL store. While clients may issue `get`, `put`, and `del` primitives either directly to the data store, or within the context of a transaction, for simplicity of the protocol description, we'll assume that all accesses are transactional and that each client has a single outstanding transaction. In the actual implementation, clients may begin any number of transactions simultaneously, may mix transactional accesses with direct `get`/`put` operations on the data store, and may create nested transactions. We revisit these issues following the basic protocol description.

2.1 Ordering Constraints

In order to provide one-copy serializability, the transaction management protocol needs to identify all required timing related constraints. In order to perform this, Warp identifies *overlapping* transactions. Formally, a transaction T_A is said to overlap a transaction T_B if they have an object immediately in common, or if T_B appears in the transitive closure of T_A's overlapping transactions. Non-overlapping transactions are said to be *disjoint*.

Intuitively, identifying overlapping transactions is critical for consistency because all of the operations involved in two overlapping transactions need to be ordered with respect to each other to ensure atomicity and serializability. At the same time, identifying disjoint transactions is critical for performance, as they can proceed safely in parallel, without restriction. Figures 3 and 4 respectively illustrate disjoint and overlapping transactions.

Operations performed within disjoint transactions may freely interleave without violating one-copy serializability because no matter what order the operations execute, the final state is, by definition, indistinguishable by clients. Had a client issued an operation (whether its own transaction or raw accesses directly against the key store) that could have distinguished between these states, that operation would cause the previously disjoint transactions to overlap, and thus would cause Warp to enforce strict atomicity and ordering between them. Linear transactions leverage this observation by executing disjoint transactions without any coordination.

Overlapping transactions require careful handling to ensure serializability. If two transactions T_A and T_B overlap, all operations $o_A \in T_A$ need to be executed either strictly before, or strictly after, $o_B \in T_B$. Implemented naively, such an ordering constraint may imply, in the worst case, establishing an ordering relationship between a newly submitted transaction and every previously committed transaction, yielding $O(N)$ complexity for transaction processing. But the astute reader will note that, if all the reads operations in a transaction T_B have read state that is subsequent to all the write operations in T_A, then the two transactions are already implicitly ordered with respect to each other. It would be redundant and wasteful to spend additional cycles on ordering transactions whose execution times differ so much that one transaction's state is already reflected in the read set of a subsequent transaction.

The Warp protocol, then, concerns itself with correctly identifying overlapping transactions, determining happens-before relationships only between those operations that need to be serialized with respect to each other, and enabling disjoint operations to proceed without coordination. The next section describes how the protocol accomplishes these goals.

2.2 Protocol

Warp operates by crafting a chain of servers to contact for each transaction such that the chain will identify all overlapping transactions and enable operations to be sequenced.

The chain for each linear transaction is uniquely determined by the keys accessed or modified within the transaction. The chain for a transaction is constructed by sorting a transaction's keys and mapping each key to a server using the consistent hashing of the underlying key-value

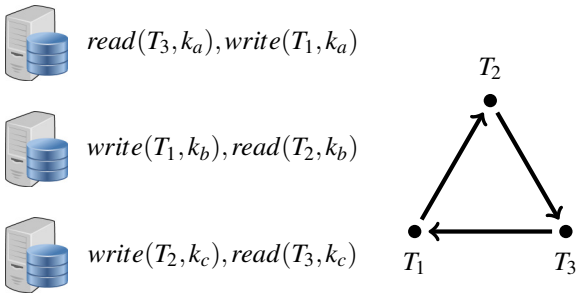

Figure 5: A dependency cycle between three transactions T_1-T_3 that read and write keys k_a-k_c. If the three data servers were to commit data out-of-order, the transaction dependencies would yield the cycle shown on the right, violating serializability. Linear transactions permit only those dependencies that do not introduce a cycle.

store. For example, the canonical chain for a linear transaction that accessed (read, write or delete) keys k_a and k_b is the two servers that hold the keys, in the order k_a, k_b. The servers are always arranged according to the lexical order of their respective keys. If a server is responsible for multiple ranges of keys, then it will occur in multiple locations in the chain.

The next step in linear transactions is to process a transaction through its corresponding chain. This is performed in two phases: a forward pass determines overlapping transactions, establishes happens-before relationships, and validates previous reads, while a backward pass either passes through an abort or commit response. Much like two-phase commit [21], the first phase validates the transaction before the second phase commits the result; however, unlike two-phase commit, linear transactions enable multiple transactions operating on the same data to prepare concurrently, tolerate failures of the client as well as the servers, and involve no data servers other than the ones holding the data accessed in a transaction.

The primary task of the forward phase is to ensure that a transaction is safe to be committed; that is, the reads it performed during the transaction and used as the basis for the writes it issued, are still valid. When a client submits a transaction, it goes through its transaction context and issues a `condput` with the old value it read for each object in its read set, where the new value is blank if the transaction did not modify that object. The rest of its modifications are submitted as regular put operations. The conditional part of the `condput` is executed during the forward phase, and if any conditionals fail, the chain aborts and unrolls.

The second critical task in the forward phase is to check each transaction against all concurrent transactions; that is, transactions that have gone through their forward, but not yet their backward phase. If the transactions operate on separate keys, they are *isolated* and require no further consideration. Transactions that operate on the same keys may either be *compatible*, in the case of a read-read conflict, or *conflicting*, in the case of read-write or write-write conflicts. Compatible transactions may be prepared concurrently. Of a pair of conflicting transactions, only one may ever commit. If a transaction conflicts with any concurrently prepared transaction, it must be aborted. On the other hand, if a transaction is compatible with or isolated from all concurrently prepared transactions, the server may prepare the transaction and forward the message to the next server in the chain.

Once a prepare message traverses the entire chain, the prepare phase completes and the commit phase begins. Commit messages traverse the chain in reverse, starting with the last server to prepare the transaction. Upon receipt of a commit message, each server locally applies writes affecting keys for which it is mapped to by the key-value store and passes the commit message backward to the previous server in the chain.

While the description above outlines the basic operation of the chain mechanism, the protocol as described does not achieve serializability because the overview so far omitted the third crucial step where compatible transactions are ordered with respect to each other. Figure 5 illustrates why ordering compatible, overlapping transactions is crucial with an example involving three transactions reading and modifying three keys held on three separate servers. If uncoordinated, these three servers may inconsistently apply the transactions, forming a dependency cycle between transactions. Under this hypothetical scenario, each server will see only two of the three transactions and will only establish one edge in the dependency graph with no knowledge of the other dependencies. To rectify this problem, compatible transactions must be applied in a globally consistent order that does not introduce dependency cycles. We discuss how linear transactions propagate dependency information in both phases to accomplish this.

2.3 Dependency Management

Linear transactions prevent dependency cycles between transactions by collecting and propagating dependency information. This dependency information comes in two forms. First, *happens-before* relationships establish explicit serialization between two transactions. To say that $T_1 \rightarrow T_2$ is to say that T_1 happens-before T_2 and must be serialized in that order across all hosts. The second dependency type is a *needs-ordering* dependency that indicates that two transactions will necessarily have a happens-before relationship in the future, but cannot be ordered at the current point in time. Conceptually, the dependencies may be modeled on a graph, where di-

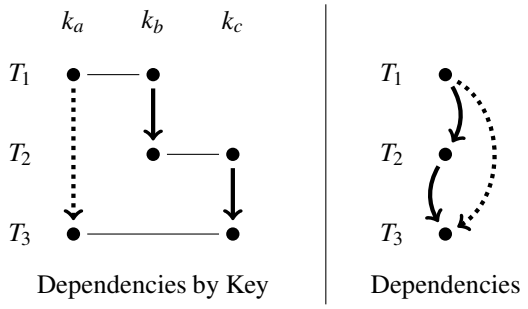

Figure 6: Linear transactions capture dependencies between transactions. This example shows three transactions, each of which touches two keys. The diagram on the left shows how happens-before relationships (arrows) are detected on a per-key basis. The dashed arrow is a transitively-defined dependency. The diagram on the right shows the overall acyclic dependency graph.

rected edges indicate happens-before relationships and undirected edges indicate needs-ordering relationships that will eventually become directed edges.

The protocol captures all dependency information as transactions traverse chains in the forward and reverse direction. Dependencies accumulate and propagate in the same messages that carry the transactions themselves. This embedding ensures that, for each transaction, the dependency information will be immediately available to every successive node without additional messaging overhead.

Servers introduce happens before relationships as they encounter previously committed transactions that pertain to keys appearing in the current transaction. Conceptually, whenever a server introduces a happens-before relationship, it also embeds all transitive relationships (in practice, garbage collection limits the size of these sets, a topic we discuss later). These implicit dependencies are added during both the forward and backward phases. Note that since all dependencies relate to compatible transactions, adding new dependencies during the backwards phase is a safe operation that cannot cause an abort.

Servers capture needs-ordering dependencies during the prepare phase of the transaction. For each concurrently prepared, compatible transaction, the server emits a needs-ordering dependency. The dependency specifies the two transactions and designates a server $\S_\omega$ that must translate the needs-ordering dependency into a happens-before dependency. S_ω is chosen such that it is the server responsible for the last key in common to both transactions. This server sees the commit message first, as it is being propagated in the backward direction, and thus assigns the order to the two transactions. Every other server in common to the chains must commit in accordance with this server's selected ordering.

A designated server S_ω needs to convert a needs-ordering dependency into a happens-before dependency in a manner that maintains serializability. If done incorrectly, the server could introduce a dependency cycle. For instance, Figure 6 illustrates a case where transactions T_1 and T_3 are ordered by the server holding k_a. If this server were to order $T_3 \rightarrow T_1$, the dependency graph would contain a cycle.

To avoid such failures to serialize, designated servers transform needs-ordering dependencies into happens-before dependency only when they have a complete view of the dependency graph. To obtain this, the server waits until it receives a `commit` message for every prepared-but-not-committed compatible transaction. Once a server has this information, it may consult the dependencies of all overlapping, compatible transactions, and compute the correct direction for the needs-ordering dependency. In the example above, the server holding k_a should order $T_1 \rightarrow T_3$ based on the embedded dependencies of all transactions, and lead to a serializable order. The next section discusses the invariants that ensure that such an order is achievable.

2.4 Anti-Cycle Invariants

The linear transactions protocol ensures correctness by ensuring that the dependency graph is acyclic. This section provides a sketch of why the dependency management maintains the anti-cycle invariant at all times.

The observation to make here is that for any possible cycle that could exist, there is always one happens-before dependency that, if directed correctly, would prevent the cycle and preserve the anti-cycle invariant. The protocol does this by treating every needs-ordering dependency as a case that may introduce a cycle. Given sufficient information about other edges in the graph, it's always possible to make this decision.

The protocol guarantees that sufficient dependency information is available by first capturing all dependencies, and then making sure that all dependencies propagate through the whole system. All dependencies are inherently captured because each server checks local state for compatible transactions. The dependencies propagate because servers only add, and never remove, dependencies.

Note that servers must consult the embedded dependencies for both transactions in a needs-ordering relationship before a happens-before relationship may be established. We can see an example case where this is necessary in Figure 6. In this figure, the dependency $T_1 \rightarrow T_2$ may be introduced either as a happens-before dependency when T_1 commits before T_2 prepares at k_b, or as a needs-ordering dependency when T_2 prepares before T_1 commits at k_b. The former case will cause dependencies to propagate through the messages for T_2 and

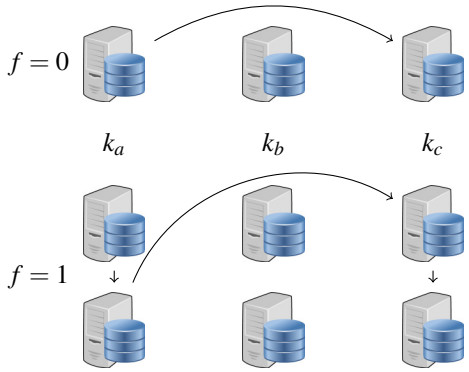

Figure 7: Fault tolerance is achieved through replication. The top set of servers shows an $f = 0$ configuration that tolerates no failures. By inlining replicas within the linear transaction's chain, the $f = 1$ deployment shown on the bottom can withstand one server failure for each key. The linear transaction is threaded through all relevant replicas.

T_3 while the latter case causes the server holding k_b to dictate the order and embed the dependency in T_1's commit message. In both cases, the server holding k_a has sufficient information to infer that $T_1 \rightarrow T_3$ using the relationships $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_3$.

2.5 Fault Tolerance

In a large-scale deployment, failures are inevitable. Linear transactions provide a natural way to overcome such failures. Specifically, linear transactions can easily permit a subchain of $f + 1$ replicas to be inlined into a longer chain in place of a single data server. This allows the system to remain available despite up to f failures for any particular key. Within the subchain, chain replication maintains a well-ordered series of updates to the underlying, replicated data. Operations that traverse the linear transaction chain in the forward direction will pass forward through all inlined chains. Likewise, operations that traverse the chain in reverse will traverse inlined chains in reverse. Figure 7 shows a linear transaction that traverses an $f = 0$ configuration and the same transaction under an $f = 1$ configuration.

This fault tolerance mechanism naturally tolerates network partitions as well. Servers that become separated from the system during a partition will not make progress because they are partitioned from the cluster, and any transaction that commits is guaranteed to have traversed all servers in the chain. To ensure liveness during the partition, the system treats servers that become partitioned as if they are failed nodes. After the partition heals, these servers may reassimilate into the cluster. Epoch identifiers in messages prohibit the mixing of messages from different configurations of the system.

Note that the notion of fault-tolerance provided by linear transactions is different from the notion of durability within traditional databases. While durability ensures that data may be re-read from disk after a failure, the system remains unavailable during the failure and recovery period; in contrast, fault tolerance ensures that the system remains available up to a threshold of failures.

2.6 Atomicity

The protocol ensures that transactions execute atomically; either all operations take effect, or none do. Since servers can never convert a `COMMIT` message into an `ABORT` or vice-versa, all nodes on a chain will unanimously agree on the outcome by the time an acknowledgement is sent to the client. In the event of a failure, the chain will be reconfigured and queued messages will be re-sent, enabling the chain to continue in unison.

2.7 Consistency

The consistency of the data store is preserved by linear transactions. With each commit, the system is taken from one valid state to the next. All invariants that an application may maintain on the data store are upheld by the linear transactions protocol.

Transactions are fully consistent with non-transactional key operations issued against the data store. Upon receipt of a key operation for a key that is currently read or written by a transaction, the system delays the processing of the key operation until after the transaction commits or aborts. This renders non-transactional key operations compatible with the linear transactions.

2.8 Isolation

Clients' optimistic reads and writes are consistent with one-copy serializability. Over the course of the transaction, the client collects the set of all values it read. A committed linear transaction guarantees that the checks specified by the client are valid at commit time. Although the values read may change (and change back) between when the client first reads, and when the transaction commits, the client is unable to distinguish between this case and a case in which the client read the values immediately before commit.

2.9 Nonblocking

Linear transactions are non-blocking [25] and guaranteed to make progress in the normal case of no failures. A transaction does not spuriously abort; it will only be aborted or delayed because of a concurrently executed, conflicting transaction. For each aborted transaction, there always exists another transaction that made progress at the key generating the conflict. Because there are only a finite number of transactions executing at any given time, there will always be at least one transaction that commits successfully causing others to abort. This satisfies the non-blocking criteria.

2.10 Garbage Collection

Our protocol as described thus far seems to collect information about transactions without bound. A simple gossip-based garbage collector with predictable overheads keeps the size of these sets in check. Specifically, each transaction is identified by a unique 128-bit id assigned to it by the first storage server in its chain, created by concatenating the ip address and port of the server with a monotonic counter. These transaction identifiers are strictly increasing, allowing each server to broadcast the lowest-numbered transaction that has prepared but not yet committed or aborted. Each server periodically broadcasts the lowest transaction id that has prepared but not committed or aborted. Upon collecting such broadcasts from its peers, a server can completely flush all information related to previous transactions. This enables large numbers of transactions to be garbage collected using a constant amount of background traffic.

3 Implementation

We have fully implemented the system described in this paper. The codebase consists of 74,563 lines of code, approximately 5,500 lines of which are exclusively devoted to processing transactions. The Warp distribution provides complete bindings for C, C++, and Python and supports a rich API that supports string, integer, float, list, set, and map types and complex atomic operations on these objects, such as conditional `put`, string prepend and append, integer addition/subtraction/multiplication/division, list prepend, list append, set union/intersection/subtraction, and atomic string or integer operations on values contained within maps and search over secondary values. Warp supports nested transactions that allow applications to create an arbitrary number of transaction scopes, and commit or abort each one independently.

3.1 Programming Model

Clients connect to Warp via a WarpClient object, through which a client can issue immediate, non-transactional operations to the data store. Clients create transaction objects using the `begin_transaction` call from the WarpClient. The transaction object provides the exact same interface as the WarpClient, enabling applications to easily wrap operations within a transaction. Figure 8 shows an example banking application that was converted from the WarpClient to a transaction using just the three marked lines of code. Whereas non-transactional code issues operations immediately to the data store, the transaction object stores reads and writes in a per-transaction local key-value store. At commit time, the read and modified objects are aggregated by the client and sent en-masse to the datastore.

Warp natively supports transactions that cross schema boundaries. The linear transaction incorporates servers from different schemas into the chain just as it does for operations on different keys.

```
def transfer_money(client, src, dst, amt):
    t = client.begin_transaction()  # <-
    src_bal = t.get(src)
    dst_bal = t.get(dst)
    if src_bal >= amt:
        src_bal -= amt
        dst_bal += amt
        t.put(src, src_bal)
        t.put(dst, dst_bal)
        t.commit()                  # <-
        return True
    else:
        t.abort()                   # <-
        return False
```

Figure 8: An example function that transfers money from one account to another inside of a transaction. The marked lines indicate the transaction-specific code that a developer would add to convert an application to Warp.

3.2 Nested Transactions

Warp supports arbitrarily nested transactions. Clients may perform a transaction within an ongoing transaction. Every nested transaction maintains its own locally-managed transaction context. Each read within a nested transaction passes through all parent transactions before finally reaching the key-value store, stopping at the first key-value store that contains a copy of the object. At commit time, the client atomically compares a nested transaction with its parent, and can locally make the decision to commit or abort. When the nested transaction commits, it atomically updates its parent's transaction context. When the root parent of all nested transactions commits, it includes all the checks seen by any nested transactions started within. The resulting linear transaction will commit the changes for both the parent transaction and all linear transactions.

3.3 Coordinator

Warp relies on a coordinator to keep track of metastate about cluster membership. A replicated state machine maintains and distributes a mapping that determines how objects are mapped to servers. Clients consult this mapping to issue reads and writes to the appropriate servers, while servers use the mapping to dynamically determine their next and previous servers for each linear transaction's chain.

Each time a server reports to the coordinator that a failure has disrupted one or more chains, the coordinator issues a new configuration acknowledging this report. Embedded within the configuration is a strictly increasing epoch number that uniquely identifies the configura-

tion. All server-to-server messages contain this epoch number, enabling servers to discard late-arriving messages from a previous epoch. Servers send each prepare/-commit/abort message at most once per epoch to ensure that other servers may detect and drop late-arriving messages. Because metadata about committed and aborted transactions persists on the servers until garbage collection, and garbage collection happens only after an operation completely traverses the chain, servers are guaranteed to be able to retransmit `prepare` messages for incomplete transactions and receive the same response. Any `commit` or `abort` message generated in the previous epoch will be ignored; only messages from current epochs will be accepted.

The coordinator is implemented on top of the Redacted replicated state machine library. Redacted uses chain replication [52] to sequence the input to the state machine and a quorum-based protocol to reconfigure chains on failure. The details of Redacted are beyond the scope of this paper; the function of the coordinator could easily be taken on by configuration services such as ZooKeeper [26] or Chubby [9].

4 Evaluation

We evaluate Warp transactions using both macro and micro benchmarks against two popular NoSQL systems, namely, Cassandra and MongoDB. We also include HyperDex in our evaluation, which serves as a representative for high-performance, second-generation NoSQL systems. The primary focus of our evaluation is on examining the performance and scalability – both as a function of transaction size and cluster size – of Warp transactions. Additionally, we illustrate the correctness of Warp transactions by building an example credit-card processing application on top of both HyperDex and Warp, and demonstrating that the Warp variant preserves all credit charges while the HyperDex variant leads to inaccurate final balances.

We performed our experiments on our dedicated lab-size cluster consisting of thirteen servers, each of which is equipped with two Intel Xeon 2.5 GHz E5420 processors, 16 GB of RAM, 500 GB SATA 3.0 Gbit/s hard disks, and Gigabit Ethernet. The servers are running 64-bit Debian 6 with the Linux 2.6.32 kernel. We deployed Cassandra version 1.2.0, MongoDB version 2.2.2, the latest HyperDex code from Git, and Warp on each server.

Each storage system was configured with appropriate settings for a real deployment of this size. This includes setting the replication factor to be the minimum value necessary to tolerate one failure of any process or machine. For HyperDex and Warp, this means that both the coordinators and the storage servers can each tolerate one failure. We used the default consistency settings for each storage system. Both MongoDB and Cassandra offer eventual-consistency [50] by default, whereas HyperDex and Warp default to strong consistency.

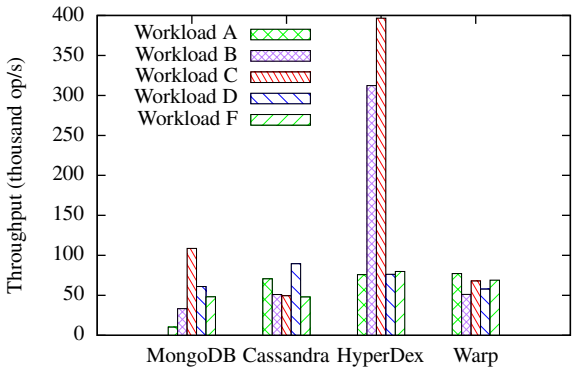

Figure 9: Warp achieves high throughput. This graph shows the average throughput for the YCSB benchmarks using 1 kB objects. MongoDB, Cassandra, and HyperDex are representative NoSQL systems that do not provide atomic transactions. Despite offering significantly stronger guarantees, Warp achieves throughput that is comparable to, or exceeds, the throughput of other systems. Warp achieves a throughput of more than fifty-thousand transactions per second for each workload.

4.1 YCSB

The Yahoo! Cloud Serving Benchmark (YCSB) [12] is the industry-standard tool for comparing the performance of NoSQL systems. YCSB provides a common platform for benchmarking systems through a set of abstractions that match most NoSQL systems' interfaces, and a set of workloads that resemble internal workloads within Yahoo!.

YCSB workloads involve only basic, non-transaction read or write operations on the primary key. We extend the YCSB benchmark for Warp to group these operations into transactions of eight operations each. For this experiment, we run the standard YCSB workloads for HyperDex, Cassandra, and MongoDB, and group key operations within `begin` and `end` operations for Warp. Figure 9 shows the overall throughput for the five key-based workloads in the YCSB benchmark. Warp provides comparable performance to MongoDB and Cassandra, outperforming both in workload A, B, F. For write-heavy workloads, Warp achieves 80% of the throughput of HyperDex, even though it offers transactional semantics. By offering comparable performance to these commonly used NoSQL systems, we demonstrate that Warp transactions have incredibly low overhead, and for a variety of workloads, Warp will actually offer increased throughput over MongoDB and Cassandra..

To understand why Warp and HyperDex offer significantly higher throughput than the other systems, we investigate the latency of the read-modify-write opera-

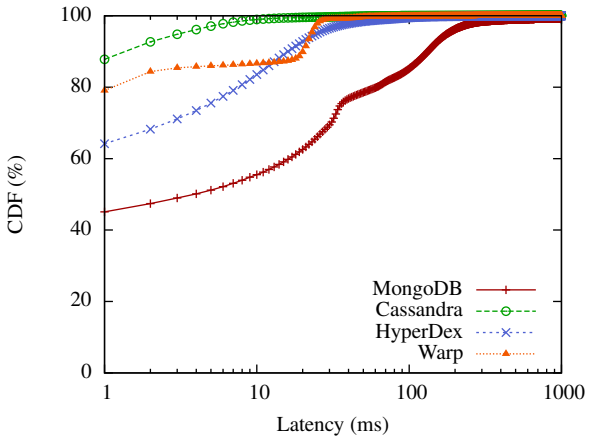

Figure 10: This graph shows the CDF of latency for write operations in Worload B. Warp and HyperDex have similar performance, and both have significantly lower latency than MongoDB.

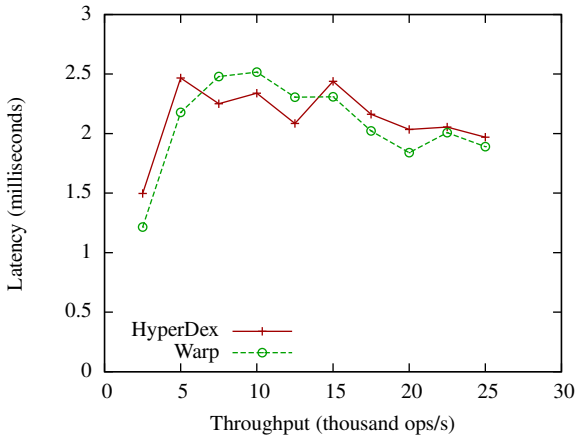

Figure 11: Warp's latency for transactions is comparable to a non-transactional workload under different fixed throughputs.

tions. Figure 10 shows a CDF of the latency of update operations for all four systems for Workload A. HyperDex and Warp complete 95% of read-modify-write operations in less than 1 ms while Cassandra takes 2 ms and MongoDB takes nearly 4 ms. The low latency explains the high throughput of Figure 9; as operations complete quicker, the clients may issue more operations per second.

Figure 11 illustrates the latency of HyperDex and Warp for different fixed throughputs. It shows that latency of Warp transactions is largely independent of the system's throughput, as the latency stays just under 2.5 ms even with increasing throughput from additional load. We also see that Warp's transactional overhead is independent of the throughput, as the latency of HyperDex and Warp remain similar throughput the experiment.

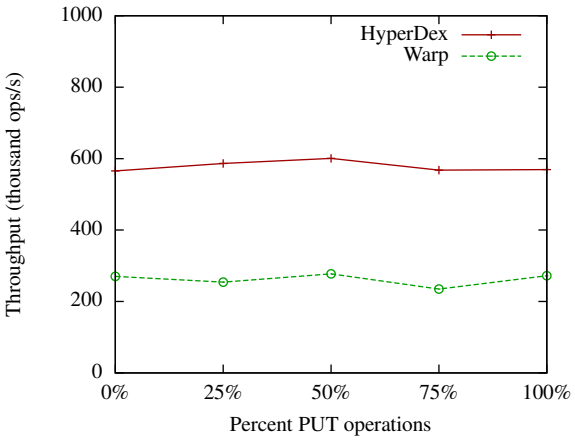

Figure 12: The ratio of read/write operations does not affect the throughput for transactions. This graph shows the per-operation throughput of transactions with two operations as the read/write ratio shifts from 100% read to 100% write.

4.2 Read/Write Ratio

YCSB Workload F specifies a mix of read and read-modify-write operations where each read-modify-write operation inherently represents a mix 50% read, 50% write within the context of a transaction. Intuitively, Warp should provide throughput that is independent of the read/write ratio within a transaction because both operations share the same dominant cost: one `get` to fetch the most recent object, possibly modifying it locally, and one pass through the linear transaction chain to validate the initial `get` and commit the result.

To test the performance impact of varying the read-/write ratios, we created a microbenchmark that creates transactions consisting of eight keys with fixed size keys and values – 8 B and 64 B respectively – chosen uniformly at random, with a configurable read/write ratio. As we can see in Figure 12, the throughput of Warp transactions is independent of the read/write ratio. Warp provides approximately 250,000 operations per second, or just over 31,000 transactions per second. Unlike our previous performance results from workload F, we see a constant performance difference between Warp and HyperDex. This is due to the significantly higher demand of our microbenchmark compared to workload F in YCSB, exposing even small performance overheads. The performance gap primarily stems from the extra round trips Warp performs to fetch data.

4.3 Transaction Size

The size of a transaction, which we define as the number of keys the transaction touches, determines the number of servers in a linear transactions chain. Therefore, increasing the size of a transaction decreases the number of transactions Warp can perform per-second, but does

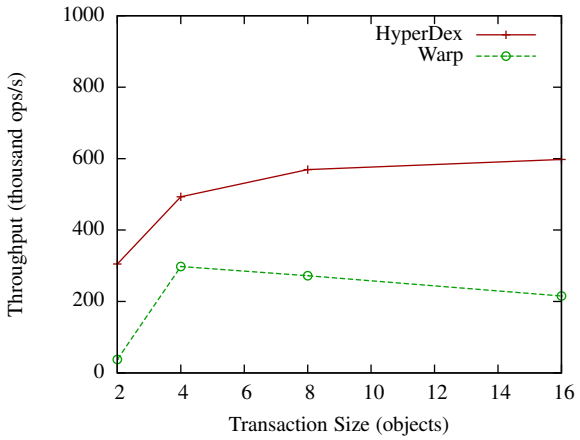

Figure 13: The total throughput of Warp is dependent on the per-operation throughput of the underlying key-value store, and largely independent of the transaction size. This graph shows the per-operation throughput of a 100% write workload as the number of keys in a transaction increases.

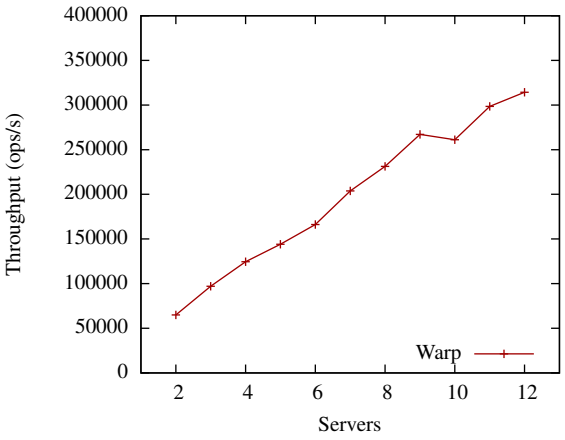

Figure 14: Warp is a scalable system. This graph shows the aggregate throughput of the system as servers are added. The number of clients and the workload remain fixed for all deployment sizes. Each point represents the average across three runs. With each additional server, the overall throughput increases proportionally, exhibiting linear scaling.

not affect the rate at which it performs key-value operations.

To test the performance impact of transaction size, we modified our previous microbenchmark to vary the number of keys in a transaction rather than the read/write ratio. In this experiment, the microbenchmark issues transactions with a configurable number of `put` operations on random keys. Figure 13 shows the results of this experiment. We can see that, as expected, the number of operations per second is relatively independent of the transaction size. This demonstrates that longer transaction chains do not introduce additional overhead, and that, for this workload, the transaction rate is a linear function of the transaction size.

4.4 Scalability

The performance of linear transactions should scale linearly with the number of servers in the cluster, as the number of servers that participate in a linear transaction is dependent only on the transaction size. Adding more servers to the cluster should therefore yield a proportional increase in performance by spreading the work across more servers. Figure 14 shows the aggregate throughput of our microbenchmark with different cluster sizes. As we had expected, Warp achieves perfect scalability in this experiment, with throughput increasing linearly with the cluster size.

4.5 Credit Card Application

To demonstrate the correctness of Warp transactions, we created a simulation of a credit card processing applications with strict atomicity, consistency, and isolation requirements. Our example application consists of merchants, users and banks, and simulates the process of a user charging her credit card. Each credit card transaction randomly selects a bank using a Zipf distribution, a user using a uniform distribution, and a merchant also using a uniform distribution. For each credit card transaction, the system picks an amount uniformly distributed between 10 and 100, and charges to the user, credits the merchant, and collects a processing fee for the bank.

Without transactional guarantees, this application is prone to lost-updates, which can lead to missing charges and even money disappearing from the system. Figure 15 shows the same bank application running on top of Warp and HyperDex to process 1,200,000 transactions. On average, using HyperDex as the storage system, the application neglected to charge users $100,000 in charges, did not credit merchants for that said $100,000, and cost the banks $12. Warp using linear transactions presented no such anomalies. Although this example is synthetic, the result generalizes to real-world applications.

4.6 Abort Rate

As with any optimistic concurrency control scheme, Warp transactions may occasionally abort. To determine Warp's abort rate, we performed read-write transactions on two keys, varying the size of the key set used to choose one of the keys. Intuitively a smaller key set will abort more often. Figure 16 shows the results of this experiment. For any write set consisting of more than ten thousand keys, the abort rate is less than 1% of all transactions.

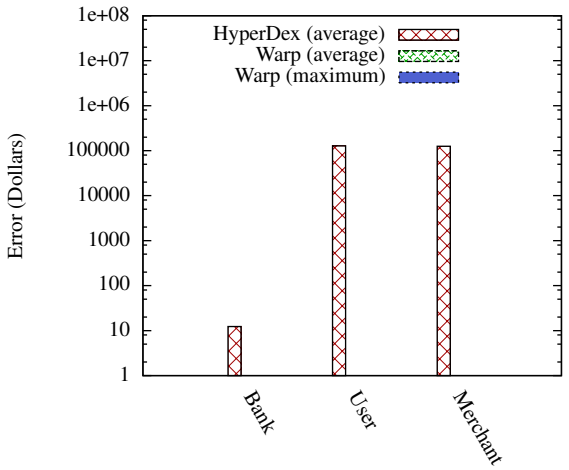

Figure 15: An example credit-card processing application built on top of HyperDex and Warp. The HyperDex application has a large discrepancy between expected and actual account balances, while the Warp application has no such discrepancies. After processing 10,000 transactions, merchants lost an average of $100,000.

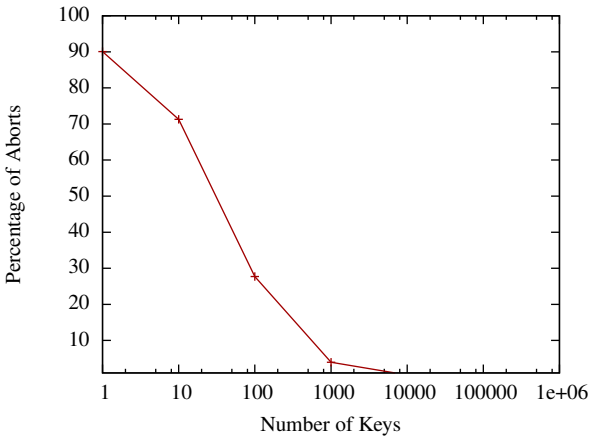

Figure 16: The abort rate decreases significantly as contention decreases. This experiment selects 2-key transactions where one key is selected uniformly at random from a set of 1 million keys and the other key is selected uniformly at random from a key set of varying size.

5 Related Work

Transaction management has been an active research topic since the early days of distributed database systems [7, 44]. Existing approaches can be broadly classified into the following categories based on the mechanism they employ for ordering and atomicity guarantees.

Centralized: Early RDBMS systems relied on physically centralized transaction managers []. While centralization greatly simplifies the implementation of a transaction manager, it poses a performance and scalability bottleneck and acts as a single point of failure. Warp, like many other systems, is based on a distributed architecture.

Distributed: The traditional approach to distributing transaction management is to provide a set of specialized transaction managers that serve as intermediaries between clients and back-end data servers. These transaction managers perform lock or timestamp management [8], and employ a protocol, such as two phase-commit (2PC), for coordination. Gray and Lamport [22] show that the classic two-phase commit algorithm is a special, $f = 0$ variant of Paxos that cannot tolerate a coordinator fault.

Some systems physcially separate and unbundle transaction management logic from the servers that store the data. Such a separation allows the design of the transactional component to be independent from the design of the rest of the system, such as data layout and caching [34]. ElasTraS [15] builds on this technique to provide an elastic layer of transaction managers for cloud services. Warp takes the opposite approach: instead of separating transactions from the underlying storage, it integrates transaction management with the underlying servers that hold the data and threads transactional updates through the storage components. This coupling refactors transaction management out of dedicated servers, distributes it across a larger set of hosts and leads to an efficient implementation.

Consensus-based: More recent work has examined how to use a general consensus protocol, such as Paxos [30] and Zab [26], to perform ordering while preserving fault-tolerance. Calvin [51] batches transactions, sequences them using a consensus protocol, and employs deterministic execution to arrive at a globally-agreed schedule of operations. Megastore [5] commits transactions within a single partition to a persistent log replicated by Paxos; cross-partition transactions employ two-phase commit. MDCC [28] durably commits transactions with one round trip using Fast Paxos, escrow transactions [37] and demarcation [6]. Scatter [20] is a strongly consistent DHT that uses Paxos for to provide linearizability and robust membership changes. Scalaris [46] builds strongly consistent multi-key transactions using Paxos. Like the consensus-based approaches, Warp relies on a fault-tolerant agreement protocol, inspired by chain-replication [52] and value-dependent chaining [18], to achieve strong consistency and atomicity. Warp does not partition the data or the consensus group, and does not place any restrictions on which keys may appear in a transaction. Warp uses no special, designated hosts to sequence transactions or to perform consensus; instead, only those servers that house the relevant data (plus transitive closure) partake in the agreement protocol. More importantly, Paxos-based approaches impose a signifi-

cant performance overhead, whereas Warp transactions are fast with minimal overhead.

Synchronized clocks: Some notable systems in this space take advantage of synchronized clocks to assign timestamps to transactions as well as determine when they are safe to commit. Adya et. al. [1] support serializable transactions and use loosely synchronized clocks as a performance optimization. Spanner [13] uses synchronized clocks to achieve high-throughput and external consistency for transactions. These techniques are complementary to Warp, which makes no assumptions about clock synchrony; processes' clocks may proceed at different rates without negatively affecting either performance or safety.

Client-managed transactions: Some systems have explored how to factor transaction management functionality to clients. CrSO [19] uses a centralized status oracle to check for read-write or write-write conflicts at commit time. Clients directly modify the underlying storage; consequently, CrSO must make provisions to garbage collect state when clients fail. Percolator [39] maintains Google's search index using distributed transactions. Clients of Percolator store both data and locks in BigTable. Periodic background processes clean up locks held by failed processes. Warp's transactions do not rely upon the client to remain available. Instead, transactions are fully fault-tolerant and do not require background processes to compensate for failures.

Alternative consistency models: Systems that operate across the wide area have explored consistency models weaker than linearizability. Real-time causal consistency [35] has been shown to be an upper bound for always-available, convergent data stores. COPS-GT [32] provides get transactions which retrieve a causal+-consistent view of a set of keys. Eiger [33] expands the guarantees of COPS-GT with write-transactions which commit atomically in accordance with the causal order of the respective writes. Both COPS-GT and Eiger build on top of strongly-consistent key-value stores. Walter [47] implements a consistency model called parallel snapshot isolation using counting sets for resolving conflicts. Similar to commutative data types [31], counting sets reconcile multiple concurrent writes by merging conflicting versions. Warp focuses not on low-latency geographically distributed transactions, but on providing fully-serializable transactions within a single datacenter.

Data Shipping G-Store [16] provides serializable transactions on top of HBase. Instead of using a separate transaction manager, G-Store changes the primary replica of all objects involved in a transaction to a single server which may then process the transaction atomically.

Limited Functionality Sinfonia [2] provides minitransactions which allow an application to specify sets of checks, reads, and writes and commit the result in two round trips. Granola [14] supports independent transactions which can be independently executed across multiple machines without coordination. Warp's transaction commit uses a set of checks and writes to validate and apply a client's changes and reduces coordination where possible.

H-Store [49] provides relational transactions and efficiently supports constrained tree applications through an optimization that guarantees that OLTP transactions are executed by a single server. Warp targets workloads that make use of key-value stores and is not designed for OLTP applications.

NoSQL Stores: NoSQL systems offer high performance and scalability, often obtained through tradeoffs involving consistency and transaction functionality. Amazon's Dynamo [17] and its derivatives [29, 40, 43] provides high write availability via sloppy quorums. Google's BigTable [10] manages structured data at the petabyte scale on top of commodity servers. Yahoo!'s PNUTS [11] stores data within structured tables using record-level replication on top of a guaranteed message-delivery service. More generally, these NoSQL systems have roots in Distributed Data Structures [23] and distributed hash tables [42, 45, 48, 53]. FAWN-KV [3] is a linearizable key-value store built on low-power servers. RAMCloud [38] provides strong consistency and fast failure recovery. Spinnaker [41] uses Paxos to provide atomic compare-and-swap operations on keys.

6 Conclusion

This paper described Warp, a key-value store that provides one-copy-serializable ACID transactions. The main insight behind Warp is a protocol called linear transactions which enables the system to completely distribute the task of ordering transactions. Consequently, transactions on separate servers will not require expensive coordination and the number of servers that process a transaction is independent of the number of servers in the system. The system achieves high performance on a variety of standard benchmarks, performing nearly as well as the non-transactional key-value store that Warp builds upon.

References

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In Proceedings of the *SIGMOD International Conference on Management of Data,* pages 23-34, San Jose, California, May 1995.

[2] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. Sinfonia: A New

Paradigm For Building Scalable Distributed Systems. In Proceedings of the *Symposium on Operating Systems Principles,* pages 159-174, Stevenson, Washington, October 2007.

[3] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array Of Wimpy Nodes. In Proceedings of the *Symposium on Operating Systems Principles,* pages 1-14, Big Sky, Montana, October 2009.

[4] Anonymized for submission.

[5] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage For Interactive Services. In Proceedings of the *Conference on Innovative Data Systems Research,* pages 223-234, 2011.

[6] Daniel Barbará and Hector Garcia-Molina. The Demarcation Protocol: A Technique For Maintaining Constraints In Distributed Database Systems. In *VLDB Journal,* 3(3):325-353, 1994.

[7] Philip A. Bernstein and Nathan Goodman. Concurrency Control In Distributed Database Systems. In *ACM Computing Surveys,* 13(2):185-221, 1981.

[8] Philip A. Bernstein and Nathan Goodman. Concurrency Control In Distributed Database Systems. In *ACM Computing Surveys,* 13(2):185-221, 1981.

[9] Michael Burrows. The Chubby Lock Service For Loosely-Coupled Distributed Systems. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 335-350, Seattle, Washington, November 2006.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System For Structured Data. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 205-218, Seattle, Washington, November 2006.

[11] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the VLDB Endowment,* 1(2):1277-1288, 2008.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems With YCSB. In Proceedings of the *Symposium on Cloud Computing,* pages 143-154, Indianapolis, Indiana, June 2010.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ 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: Googles Globally-Distributed Database. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 251-264, 2012.

[14] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In Proceedings of the *USENIX Annual Technical Conference,* 2012.

[15] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An Elastic Transactional Data Store In The Cloud. In Proceedings of the *Workshop on Hot Topics in Cloud Computing,* San Diego, California, June 2009.

[16] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: A Scalable Data Store For Transactional Multi Key Access In The Cloud. In Proceedings of the *Symposium on Cloud Computing,* pages 163-174, Indianapolis, Indiana, June 2010.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In Proceedings of the *Symposium on Operating Systems Principles,* pages 205-220, Stevenson, Washington, October 2007.

[18] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In Proceedings of the *SIGCOMM Conference,* pages 25-36, Helsinki, Finland, August 2012.

[19] Daniel Gómez Ferro, Flavio Junqueira, Benjamin Reed, and Maysam Yabandeh. Lock-Free Transactional Support For Distributed Data Stores. Poster Session. Symposium on Operating Systems Principles, Cascais, Portugal, 2011.

[20] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas E. Anderson. Scalable Consistency In Scatter. In Proceedings of the *Symposium on Operating Systems Principles,* pages 15-28, Cascais, Portugal, October 2011.

[21] Jim Gray. A Transaction Model. In *Automata, Languages and Programming,* 85:282-298, 1980.

[22] Jim Gray and Leslie Lamport. Consensus On Transaction Commit. In *ACM Transactions on Database Systems,* 31(1):133-160, 2006.

[23] Steven D. Gribble. A Design Framework And A Scalable Storage Platform To Simplify Internet Service Construction. PhD thesis, U.C. Berkeley, 2000.

[24] HBase. `http://hbase.apache.org/`.

[25] Maurice Herlihy. Wait-Free Synchronization. In *ACM Transactions on Programming Languages and Systems,* 13(1):124-149, 1991.

[26] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-Free Coordination For Internet-Scale Systems. In Proceedings of the *USENIX Annual Technical Conference,* 2010.

[27] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent Hashing And Random Trees: Distributed Caching Protocols For Relieving Hot Spots On The World Wide Web. In Proceedings of the *ACM Symposium on Theory of Computing,* pages 654-663, El Paso, Texas, May 1997.

[28] Tim Kraska, Gene Pang, Michael J. Franklin, and Samuel Madden. MDCC: Multi-Data Center Consistency. In *The Computing Research Repository,* abs/1203.6049, 2012.

[29] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. In Proceedings of the *International Workshop on Large Scale Distributed Systems and Middleware,* Big Sky, Montana, October 2009.

[30] Leslie Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems,* 16(2):133-169, 1998.

[31] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. CRDTs: Consistency Without Concurrency Control. In *The Computing Research Repository,* abs/0907.0929, 2009.

[32] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle For Eventual: Scalable Causal Consistency For Wide-Area Storage With COPS. In Proceedings of the *Symposium on Operating Systems Principles,* pages 401-416, Cascais, Portugal, October 2011.

[33] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics For Low-Latency Geo-Replicated Storage. In Proceedings of the *Symposium on Networked System Design and Implementation,* Lombard, Illinois, April 2013.

[34] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwilling. Unbundling Transaction Services In The Cloud. In Proceedings of the *Conference on Innovative Data Systems Research,* 2009.

[35] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, Availability, Convergence. University of Texas at Austin, Technical Report TR-11-22, 2011.

[36] Memcached. `http://memcached.org/`.

[37] Patrick E. O'Neil. The Escrow Transactional Method. In *ACM Transactions on Database Systems,* 11(4):405-430, 1986.

[38] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, and Mendel Rosenblum. Fast Crash Recovery In RAMCloud. In Proceedings of the *Symposium on Operating Systems Principles,* pages 29-41, Cascais, Portugal, October 2011.

[39] Daniel Peng and Frank Dabek. Large-Scale Incremental Processing Using Distributed Transactions And Notifications. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 251-264, Vancouver, Canada, October 2010.

[40] Project Voldemort. `http://project-voldemort.com/`.

[41] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos To Build A Scalable, Consistent, And Highly Available Datastore. In *Proceedings of the VLDB Endowment,* 4(4):243-254, 2011.

[42] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A Scalable Content-Addressable Network. In Proceedings of the *SIGCOMM Conference,* pages 161-172, San Diego, California, August 2001.

[43] Riak. `http://basho.com/`.

[44] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis II. System Level Concurrency Control For Distributed Database Systems. In *ACM Transactions on Database Systems,* 3(2):178-198, 1978.

[45] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, And Routing For Large-Scale Peer-To-Peer Systems. In Proceedings of the *IFIP/ACM International Conference on Distributed Systems Platforms,* pages 329-350, 2001.

[46] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: Reliable Transactional P2p Key/value Store. In Proceedings of the *SIGPLAN Workshop on ERLANG,* pages 41-48, Victoria, Canada, 2008.

[47] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage For Geo-Replicated Systems. In Proceedings of the *Symposium on Operating Systems Principles,* pages 385-400, Cascais, Portugal, October 2011.

[48] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service For Internet Applications. In Proceedings of the *SIGCOMM Conference,* pages 149-160, San Diego, California, August 2001.

[49] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End Of An Architectural Era (It's Time For A Complete Rewrite). In Proceedings of the *International Conference on Very Large Data Bases,* pages 1150-1160, 2007.

[50] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing Update Conflicts In Bayou, A Weakly Connected Replicated Storage System. In Proceedings of the *Symposium on Operating Systems Principles,* pages 172-183, Copper Mountain, Colorado, December 1995.

[51] 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 *SIGMOD International Conference on Management of Data,* pages 1-12, Scottsdale, Arizona, May 2012.

[52] Robbert van Renesse and Fred B. Schneider. Chain Replication For Supporting High Throughput And Availability. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 91-104, San Francisco, California, December 2004.

[53] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: A Fault-Tolerant Wide-Area Application Infrastructure. In *SIGCOMM Computer Communications Review,* 32(1):81, 2002.