Madaba: Starvation free, Scalable Transactions for Sharded Key-value Stores

by

Jeffrey DallaTezza

Advisor: Dr. Yair Amir

A project report submitted to The Johns Hopkins University in conformity with the requirements for the degree of Master of Science.

Baltimore, Maryland

November, 2015

Abstract

Sharded key-value stores are widely used as the storage system for big-data applications, largely due to their ability to scale horizontally. These systems often achieve scaling by sacrificing ACID transactions that would involve multiple shards, and systems which provide ACID transactions across multiple shards often introduce the potential for starvation of some processes.

This report introduces Madaba, a system which provides linearizable ACID transactions on top of a sharded key-value store, with the guarantee that each transaction will eventually be completed. Madaba uses the existing open-source key-value store, HyperDex, for storage, and implements transactions entirely in the client library.

1 Introduction

Key-value stores are storage systems that use an associative array, or map, as the fundamental data model. Data is modeled as a set of key-value pairs such that each key is unique, and users can read the value associated with a key or update the value associated with a key. Examples of widely used key-value stores are BerkeleyDB and LevelDB, databases designed to store data which fits on a single machine. Recently, key-value stores like BigTable, Dynamo, and Cassandra have been able to utilize clusters of machines to provide high availability and scalability.

BigTable is a proprietary key-value store that was designed for storing petabytes of data across clusters of thousands of servers [2]. BigTable allows users to read, write, or delete objects which are accessed by row key. To divide load across multiple servers, the keys for each table are divided into ranges called tablets, and each server is assigned a set of tablets. When a client performs an operation on an object, it contacts the server responsible for the object's tablet. Because all of the operations on an object go through the same server, this server can ensure that every operation is executed atomically, including read-modify-write operations. Each operation is committed to a replicated log before it is applied, so the system can recover from the failure of a server by replaying the log on another server. BigTable maintains a consistent view of tablet assignment by using a strongly consistent locking service based on Paxos [1]. Because operations of different tablets are mostly independent, the system can scale linearly with the number of machines. Maintaining tablet assignment requires centralized coordination, but this approach has been shown to scale to thousands of machines.

A notable open-source, sharded key-value store is HyperDex [5]. Users can read, write, or delete objects stored in HyperDex, as well as execute atomic read-modify-write operations, such as compare-and-swap or atomic-increment. All of these operations are linearizable, which means concurrent operations appear to occur atomically between then the operation begins and completes. Tables in HyperDex are partitioned by key, and each partition is stored on a set of replicas which form a logical server. The mapping from partition to replicas is maintained in a centralized replicated state machine.

Objects with linearizable read-modify-write operations are very powerful tools for building concurrent systems. In fact, the compare-and-swap operation can be used to implement a concurrent version of any data structure on multiprocessing architectures [6]. Despite this, implementing data structures with these primitives can be complex, inefficient, and error prone, especially when considering the possibility of machine failures. To implement more complex logic, many systems offer the ability to perform atomic transactions which can affect multiple items. ACID transactions are common in databases which operate on single machines, but not as common on distributed key-value stores. This is largely because the scalability of shared key-value stores comes

from the independence of different shards, and this property is broken by transactions that affect objects from different shards.

Recently, several systems have added transactions to sharded key-value stores while maintaining the scalability that makes them attractive. An example of such a system is Percolator [7], a proprietary system which implements transactions in a client library which uses BigTable for storage. Percolator provides atomic transactions with snapshot-isolation, meaning all of the reads within a transaction reflect a consistent snapshot of the data. To ensure consistent reads, each operation receives a monotonically increasing timestamp, and multiple versions of each object are stored. One can read a consistent view by getting a timestamp, then reading the latest version of each object which was committed before that timestamp. In Percolator, write transactions are performed by acquiring a lock on each item and committed once all of these locks have been acquired. An issue is that in the case of a write-write conflict between transactions, the transaction which reaches the object first will have priority. A long running or slow transaction may have to abort and retry repeatedly if it must perform writes on objects that are frequently updated. This can starve some processes, meaning they are not able to complete a transaction.

Another system which provides transactions in addition to the normal operations of a key-value store is Warp [4]. Warp implements strictly serializable transactions on top of HyperDex. Transactions start by getting the current version of each object which is read by the transaction with no regard for snapshot isolation. When a transaction tries to commit, it atomically performs any write operations and verifies that the objects it read were not changed by other transactions. As will be described in Section 2.5, Warp manages to scale by limiting coordination in the validation step to the servers that are responsible for the affected objects. An issue with this approach is that if a transaction reads an object and does not commit before that object is modified by another process, the transaction will fail. This can lead to starvation if a process is slow, or must read frequently updated objects. Starvation can be an issue in concurrent systems, because some transactions may never complete.

The goal of this work is to address the problem of starvation, while maintaining the scalability characteristic of a sharded key-value store. A secondary goal is to build the transactional system as a client library of a key-value store, so that the approach could be used with existing systems and key-value stores which are offered exclusively as services. We introduce Madaba, a system which provides strictly serializable transactions, and guarantees each transaction is eventually completed.

Madaba builds on HyperDex, retaining the single object operations of a key-value store, while adding the ability to perform atomic transactions which can perform operations on multiple objects. Madaba is implemented as a client library which makes calls to HyperDex. All data is stored in a HyperDex cluster, and the HyperDex server code is not modified. Each object is stored as a record in HyperDex which contains two versions of the object and additional fields to represent locks. Madaba

performs transactions with a form of two-phase-locking; clients use HyperDex's atomic read-modify-write operations to acquire a lock on each object in the transaction, and commit once all of these locks have been obtained. Because acquiring locks in Madaba requires writing to the lock field of an object, reads in Madaba are relatively more expensive than systems which use optimistic concurrency control or only ensure snapshot isolation. In the event that transactions conflict, priority is given to the older transaction. Because clients may crash, it is possible for a newer transaction to abort an older transaction after a timeout, but the length of this timeout is increased each time a transaction must be retried. By giving priority to older transactions and increasing the timeout required to abort an older transaction, Madaba can ensure that transactions will eventually be completed, despite contention or differences in process speeds.

Madaba is able to maintain the horizontal scaling of the underlying key value store by ensuring that two transactions will only operate on the same key-value pair if they have objects in common. This allows Madaba to exploit the parallelism of transactions which affect disjoint sets of objects, and avoids the introduction of any global bottlenecks.

This report discusses the algorithm used in Madaba in further detail and evaluates its semantics and performance.

2 Related Work

2.1 Sharded Key-Value Stores

BigTable is a proprietary key-value store which is used by Google for many large scale applications. Objects stored in BigTable are accessed by key, and may have multiple secondary attributes. BigTable supports the ability to read the value of an object, write the value of an object, or perform a checkAndPut, which updates the object only if the provided condition is satisfied. All of these operations are linearizable, and this guarantee is maintained in the event of failures and partitions. Tables are partitioned by key, into units called tablets. For each tablet, one server is assigned to handle client requests for that tablet. Because these servers (tablet servers) can fail, BigTable uses the distributed Google File System (GFS) to store data redundantly. Tablet servers can maintain a cache in memory to serve client request quickly, but writes must be committed to a GFS log before they can be completed. If a tablet server crashes, its tablets are reassigned to other servers which can replay the log file. GFS provides operations to read and modify files on a cluster of machines with a relaxed consistency model. This relaxed consistency model is sufficient for BigTable, because each file should only be modified by one tablet server at a time. To manage the assignment of tablets to tablet servers, BigTable uses a distributed

lock service called Chubby, which uses Paxos to remain strongly consistent. A tablet server must maintain an exclusive lock for each tablet it is assigned, and must ensure that it still holds the lock before servicing a client requests. Chubby uses timed leases to allow servers to verify that they still hold a lock efficiently. Chubby is not designed for high throughput or volume, but BigTable has been demonstrated to scale to thousands of machines and tens of thousands of clients.

HBase is a popular open-source key-value store which was modeled after Google's BigTable. The architecture of HBase is very similar to BigTable, and it also depends on a distributed file system and a distributed lock service. Instead of using GFS and Chubby, HBases uses the Hadoop Distributed File System for redundant storage and Zookeeper for maintaining tablet assignment.

Recently, distributed key-value stores have been offered as a service, through systems such as Google's Cloud BigTable and Amazon's DynamoDB. Managed services have gained popularity in practice largely because users are saved the operational overhead of maintaining such a system. The decision to implement Madaba as a client library was motivated by the ability to use a similar approach for fully managed services.

2.2 HyperDex

The sharded key-value store used for storage in Madaba is Hyperdex. Hyperdex splits the key-space into regions, and assigns each region to a set of servers which use chain replication. The assignment of regions to chains is managed by a replicated state machine based on paxos. Objects stored in Hyperdex have primary keys and can have multiple secondary attributes. Clients can perform reads, writes, and conditional updates, and all operations are linearizable. The ability to perform atomic conditional updates, which will perform a write operation only if certain conditions are met, is used heavily in Madaba for ensuring correctness and starvation freedom.

2.3 Concurrency Control Systems

In the sharded key-value stores we discussed, consistency guarantees are maintained for operations on single objects. This means performing an operation on an object only requires running a replication protocol between the replicas who store that object. Performing general transactions on multiple objects is more difficult to scale, because ensuring ordering and atomicity involves the replicas for all of the objects in the transaction.

Concurrency control is responsible for ensuring the correctness of transactions while maximizing performance. Two widely used mechanisms for concurrency control are two-phase locking (2PL) and optimistic concurrency control (OCC).

2PL is a common approach to concurrency control that can be used to ensure serializability. First is the expanding phase, in which a lock is acquired on each object, such that no other process can access the object until the lock is released. Once all of the locks have been acquired, locks can be released in what is known as the shrinking phase. Once a lock has been released, no new locks can be acquired. Informally, this ensures safety because no effects of a transaction are exposed until the shrinking phase, and by this point all of the objects being modified were already locked.

As described, two-phase locking can potentially lead to deadlock because transactions could mutually block. One system to avoid this problem is wound-wait [8]. In wound-wait, each transaction is given a unique number, which represents its priority. If a transaction requires a lock that is held by another transaction with lower priority, it wounds that transaction, sending it a message which forces it to abort and retry. Once the abort occurs, the transaction with higher priority can take the lock and continue. This ensures that the transaction with highest priority will be able to make progress.

Optimistic concurrency control (OCC) refers to approaches which use resources without acquiring locks, then validate that the objects were not modified by other transactions before committing. While some form of locking may be used to validate and commit atomically, OCC can reduce the overhead of managing locks if contention is low. With a high conflict rate, this approach can be less efficient than locking, since transactions could be repeated many times. In the worst cast, OCC approaches can starve processes if data is repeatedly modified between when it is read and when it is validated.

2.4 Transactional Distributed Databases

As previously mentioned, Percolator is a proprietary system which implements transactions in a client library which uses BigTable for storage. Percolator stores multiple versions of each item and accesses them by timestamp to achieve snapshot isolation for reads. Writes within transactions are performed by acquiring a lock on each item being written. These locks are implemented by writing to an additional field stored on each item. One lock is considered the primary lock, and once all of the write locks have been acquired the primary lock is updated to signify that the transaction has been completed. In the event a client fails during a transaction, the status of the primary lock is used to determine whether to roll the transaction back or forward. If a transaction attempts to write to an item that is already locked by other transaction, it will abort itself immediately. This approach to concurrency control can cause starvation if contention is high, because a transaction can keep aborting repeatedly. Percolator is an example of a system which chose to implement transactions in the client library of a sharded key-value store for simplicity and scalability.

Spanner is a distributed database deployed in Google. Spanner provides general transactions with strict serializability [3]. Similarly to BigTable, tables in Spanner are partitioned into tablets. Each tablet is assigned to a Paxos group, which is a group of servers that use Paxos for consistent replication. To perform a transaction, a client acquires a lock on each object that is read, while storing writes locally. Once the client has completed this, it initiates two-phase commit between leaders of the different Paxos groups. Write locks are obtained by the group leaders during the prepare phase of two-phase commit. Spanner uses wound-wait to avoid deadlocks. To tolerate client crashes, locks held by clients can timeout if the client is unable to send a keepalive message in time. Paxos groups use timed leases to support long lived leaders. Because leaders are not frequently changed, locks can be managed by the leaders without coordinating with other replicas. Spanner also makes use of highly accurate clocks to support efficient read-only transactions and paxos leader lease management, but that is beyond the scope of this work.

2.5 Warp

Warp is a system built on HyperDex. It maintains the operations of HyperDex, but also allows ACID transactions which can perform multiple HyperDex operations atomically with strict serializability [4]. To perform a Warp transaction, clients start by optimistically reading objects and storing writes in a local cache. To commit a transaction, the set of read objects are validated and the writes are performed atomically using a protocol called linear transactions. All of the servers involved in the transaction are ordered by id to form a chain. Starting with the head of the chain, each server validates the transaction by verifying that the values which were optimistically read were not modified by another transaction which was previously validated or committed, and the transaction does not write any values that were read by a validated transaction. If every server is able to validate the transaction, it is safe to commit. The tail of the chain sends a commit message which goes in reverse through the chain to the head. If two transactions conflict on any objects, the chains necessarily overlap, and the last server which is a part of both chains is used to decide the ordering for those transactions.

This protocol maintains scalability by limiting coordination to the servers involved in the transaction, and limiting the ordering constraints on transactions to those which have conflicts. This protocol avoids mutually blocking transactions by requiring that each transaction acquires locks in the same order.

3 Data Model and Service Semantics

3.1 Data Model

Records in Madaba have a primary key and one or more named fields. Madaba can store multiple tables, and each table is defined with a schema that declares its fields. All read operations return all of the attributes associated with a key. Write operations can update any number of the fields in a record (other than the primary key).

3.2 Interface

As Madaba is a key-value store, it retains the ability to read and write single keys. The basic single-key operations are GET(table, key) and PUT(table, key, updates). GET returns the value associated with the provided key as a map from field name to value. PUT performs the given updates atomically on the requested key. The updates parameter is a map from field name to value. Each field name in the map will be updated, and any fields not included will remain unchanged.

Madaba also allows users to define transactions, which can combine multiple single-key operations on different keys. Users define a transaction by creating a python function that takes a transaction object, which is used to perform database operations. Within a transaction function, the code can perform multiple GETs and PUTs, as well as run external code. The calls to GET and PUT can access any tables and keys. The PERFORM(func) call is used to run a transaction. PERFORM takes a transaction function and executes it, applying any GET and PUT operations atomically on the database. In the event of conflicts with other transactions, the transaction function may be stopped and retried multiple times before the transaction is successfully completed. Once the call to PERFORM returns, the effects of the transaction function are guaranteed to have taken place.

Any example using these operations is shown in Figure 1. transfer is a function which reads the balance associate with two accounts and transfers the balance of one account to the other. transfer takes a parameter, t, which is the transaction object. Within the function, the transaction object is used to make Madaba calls.

3.3 Safety Semantics: Atomicity, Consistency, Isolation

Transactions in Madaba are strictly serializable, meaning that the execution of concurrent transactions will be equivalent to some serial execution in which each

```
client = MadabaConnection()
client.put("accounts", "jack", {"balance": 10})
client.put("accounts", "jill", {"balance": 10})
def transfer(t):
  jack = t.get("accounts", "jack")["balance"]
  jill = t.get("accounts", "jill")["balance"]
  t.put("accounts", "jack", {"balance": 0})
  t.put("accounts", "jill", {"balance": jack + jill})
  return (jack, jill)
client.perform(transfer)
```

Figure 1: Performing a transaction with Madaba.

transaction is atomically completed at some time between when it was started and when it returns.

Madaba also guarantees the consistency of reads within an aborted transaction attempt. During a transaction, if a value which was previously read is modified by another transaction, no future calls to read will complete. This means that even if a transaction attempt is not able to be committed, the part of the function that is executed is provided a consistent view. This property is typically not found in systems based on OCC, which generally provide inconsistent reads which are then validated before committing. This gives Madaba a simpler programming model, as the user need not reason about how the code would be executed with inconsistent results.

Madaba trivially ensures durability, because everything stored in HyperDex is durably stored.

3.4 Liveness Semantics: Starvation Freedom

An interesting property of Madaba is that 'perform' is guaranteed to eventually commit the transaction, regardless of relative client speeds and other client crashes, as long as the code of any transaction function can be performed in some finite amount of time. It is important to note that the amount of time required to complete a transaction is not known in advance, it is only guaranteed that the transaction will be completed.

Field	Use
lock	Id of the last transaction which attempted to read or write this object
block	Transaction which wants to acquire the lock on this object
block ts	Timestamp of the waiting transaction if there is one
new	The value of this object being proposed by the transaction which holds the lock
old	The value which was committed before the transaction holding the lock

Figure 2: Fields of a data record.

4 Transaction Protocol

4.1 Storage System

Madaba uses HyperDex for storage. HyperDex provides the required level of consistency and the ability to scale to large clusters of machines. While Madaba depends on HyperDex, the algorithm used in Madaba could be implemented with any storage layer that provides consistent conditional updates. A notable example of a system which could have been used is DynamoDB, which is a fully managed cloud database service.

4.2 Data types

Madaba stores two types of objects in Hyperdex: data records and transaction records. A transaction record is created each time a transaction is attempted, and that record maintains the state of the transaction, which is either PENDING, ABORTED, or COMMITTED. The state of a transaction is always updated using atomic compare-and-swap operations.

Data records in Madaba represent an object which the client has stored. Two versions of the object are stored in the record, and the lock field stores the id of the transaction which last attempted to update the object. The value of the object from the clients perspective depends on the state of the transaction which holds the lock - if that transaction has been COMMITTED, the *new* field holds the current value, otherwise the *old* field holds the current value.

While the lock field on a data record is used to ensure the atomicity of transactions, the *block* and *block_ts* are used to ensure starvation freedom. The *block* field is the id of a transaction which is waiting to obtain the *lock* on that object, and the *block_ts* is the timestamp of that transaction. Any operation updating the 'lock' made by transaction T is a conditional update requiring that the *block_ts* of that object is not less than the timestamp of T. A transaction can always update the *block* and *block_ts*

Field	Use
timestamp	The timestamp of this transaction.
status	Either PENDING, ABORTED, or COMMITTED.
attempt	A counter of how many times this transaction has been attempted.

Figure 3: Fields of a transaction record.

of a data record with one operation if it has a lower timestamp, which gives it priority over all newer transactions until one of those transactions waits for a timeout.

4.3 Timestamp Generation

Each transaction is assigned a unique timestamp, which is used to give priority to older transactions. While the generation of these timestamps has no effect on safety, it does affect liveness. To ensure each transaction finishes, we must guarantee that for each timestamp, T, that is generated in the system, there must be some time after which no new timestamps less than T are generated. This is because transactions may be aborted and retried indefinitely while a conflicting transaction with higher priority exists.

Madaba uses the system clock to generate a timestamp and uses the transaction id as a tiebreaker if two transactions have the same timestamp. This assumes that the system clocks are loosely synchronized, which is feasible in modern networked environments.

These timestamps could also be generated in a scalable way by using atomic counters, which are supported by many distributed key-value stores. Multiple counters could be stored in the data store. To generate a timestamp, one could randomly choose a counter, then perform an increment-and-get on that counter. Combining the value and the id of the counter would produce a unique timestamp. The throughput of that approach could be increased by storing more counters. While this approach would ensure liveness, it would increase overhead and likely result in less accurate timestamps.

4.4 Execution

At a high level, the execution of a transaction is as follows: create a new transaction record with a unique timestamp, lock each data record affected by a call to GET or PUT by updating the *lock* field of the specified data record, then commit by atomically modifying the transaction record. If the transaction is aborted at any time, restart the transaction with a new transaction record - using the same timestamp and incrementing the attempt number.

The procedure for obtaining a lock on a data object contains most of the logic for concurrency control. As described in 4.2, each data record contains additional fields for concurrency control - *lock*, *block*, and *block_ts*. The lock field contains the id of the transaction which holds the lock for that record, while the block and block ts fields hold the id and timestamp of a transaction which is attempting to acquire the lock. When attempting to lock a data record, the transaction must first acquire the *block* record. If the transaction holding the *block* has a higher timestamp, this will succeed in one operation. Otherwise, the client waits an amount of time based how many times the other transaction has been attempted before atomically replacing the *block* and *block_ts* with its own id and timestamp.

Once the block is obtained, we decide whether the transaction currently holding the lock will be aborted or committed by attempting to atomically set the state the ABORTED. If this succeeds, the transaction is aborted, otherwise the transaction must have been committed.

Once the lock has been decided, we can determine the current value of that data record, which will be used to set the *old* field. We can then update it with our lock and new value.. At this point, we make sure that our transaction is still alive by reading our transaction status and complete the operation. Simplified code for performing an update to a data record is provided below.

It is important to note that the lock field is updated for both GETs and PUTs, which means that GET and PUT operations both require performing the writes to Hyperdex. This makes read operations relatively more expensive than write operations when compared to the other transaction systems discussed previously. Percolator, which also uses client-managed locks, avoids this read overhead by offering only snapshot isolation. In Warp, values are read optimistically, then validated by the servers as part of the atomic commit protocol. Spanner reduces the overhead of read locks by only storing them on Paxos group leaders, though a group leader crash could result in clients seeing inconsistent reads within a failed transaction.

```
def obtain_block(txn, key):
  assignments = {"block": txn.id, "block_ts": txn.timestamp}
  while True:
    data = store.get(object_table, key)
    if data.block_ts < txn.timestamp:
      other_txn = store.get(txn_table, data.block)
      wait_for(txn, other_txn)
      conditions = {"block":other_txn.id}
    else:
      conditions = {"block_ts":GreaterEqual(txn.timestamp)}
    if store.cond_put(object_table, key, conditions, assignments):
      return store.get(object_table, key)
```

```
def wait_for(txn, other_txn):
  # There is no need to wait for a newer transaction or a completed
  ↪ transaction
  if other_txn.timestamp >= txn.timestamp or other_txn.status !=
  ↪ PENDING:
    return
  # The time to wait must increase with the number of times the
  ↪ other transaction has failed
  sleep(other_txn.attempt)
```

```
def decide(txn, other_id):
  if other_txn_id == "":
    return COMMITTED
  condition = {"status": NotEqual(COMMITTED)}
  assignment = {"status": ABORTED}
  # Attempt to update the state to ABORTED with the condition that
  ↪ the state is not already COMMITTED.
  # If the try_abort failed, it must have already been committed
  if store.cond_put(txn_table, other_id, condition, assignment):
    return ABORTED
  else:
    return COMMITTED
```

```
def update(txn, key, fun):
  if store.put_ine(object_table, key, {"block_ts":txn.timestamp,
  ↪ "lock":txn.id, "new":fun(""), "old":""}):
    if is_still_alive(txn):
      return ""
    else:
      return FAILED
  data = obtain_block(txn, key)
  # Check if we were overtaken already
  if data.block != txn.id:
    return FAILED
  other_state = decide(txn, data.lock)
  if other_state is COMMITTED:
    current_value = data.new
  else:
    current_value = data.old
  conditions = {"block_ts": GreaterEqual(txn.timestamp),
  ↪ "lock":data.lock} "readers":data.readers}
  if other_state is COMMITTED:
```

```
18      conditions["new"] = data.new
19    assignments = {"lock":txn.id, "new":fun(current_value),
      ↪  "old":current_value}
20    if not store.cond_put(object_table, key, conditions, assignments):
21      # If the operation failed because of another write by the same
        ↪  transaction, we can retry with the updated value
22      new_data = store.get(object_table, key)
23      if new_data.lock != data.lock or new_data.block_ts <
        ↪  txn.timestamp:
24        return FAILED
25      current_value = new_data.new
26      assignments["old"] = current_value
27      assignments["new"] = fun(current_value)
28      del conditions["new"]
29      if not store.cond_put(object_table, key, conditions, assignments):
30        return FAILED
31    # Before returning, ensure this transaction has not been aborted
32    if is_still_alive(txn):
33      return current_value
```

4.5 Optimizations

There are several optimizations implemented in Madaba which achieve better performance than the simplified version described previously. One optimization is the use of heartbeats, which are implemented by adding a counter to each transaction record. This provides an obvious improvement for long-running transactions.

Another optimization implemented in Madaba is optimistically attempting to complete transactions without updating the *block* and *block_ts*. Transactions avoid the extra write operation required to set these fields until they have already attempted the transaction and failed multiple times.

Madaba also implements shared read locks by storing a set of transaction ids with each data record.

4.6 Correctness

4.6.1 Atomicity

In Madaba, the value of a data record is determined by the status of the transaction which holds the lock. The modifications proposed by a transaction are only accepted once that transaction has been committed. Since modifications are never read before a transaction is committed, we can show atomicity by showing that at the

time a transaction is committed, all of its modifications will be visible. Locks held by transaction T can only be removed by competing transactions if T has already been aborted or committed, so if T is able to commit, all of the records T attempted to modify must still be locked at the time of commit.

Since we lock each record before committing and a successful commit implies no locks have been revoked, the atomic operation of changing a transaction's status to committed applies all of its modifications atomically.

4.6.2 Strict Serializability

In Madaba locks are not released by a transaction until it has been aborted or committed. This complies with strong, strict, two-phase locking, which ensures strict serializability. Reads within failed transactions are also consistent, because a transaction verifies that is has not been aborted before the result of the read is returned to the client.

4.6.3 Starvation Freedom

Madaba ensures starvation freedom, meaning each transaction will be completed in finite time, regardless of contention or relative process speeds.

To show this, consider the transaction with the lowest timestamp in the system. This transaction can obtain a block on any object in exactly one operation, because this is implemented as a conditional update. While the block is held obtaining the lock on that object will also take a finite number of operations, so the transaction will be able to complete in a finite number of operations unless a competing transaction times out and aborts it. Because the timeout for overtaking a transaction is increased with the number of times that transaction has been attempted, the transaction with the lowest timestamp will eventually be given enough time to be completed.

The property we require of timestamps is that for each timestamp T that the system generates, there is a finite time after which no new timestamps will be generated that are lower than T. Once this time passes for a transaction, it only needs to wait for the finite number of existing transactions with lower timestamps to either complete or crash for it to have the lowest timestamp in the system. Because the transaction with the lowest timestamp is guaranteed to be completed, and each transaction must only wait for a finite number of transactions to complete before it has the lowest timestamp, each transaction will eventually be completed, unless the process performing that transaction crashes.

5 Performance Evaluation

We evaluate Madaba by comparing its performance to plain HyperDex on a set of benchmarks. The benchmarks consist of running transactions on a cluster of servers, but while Madaba performs these transactions with the transactional guarantees discussed above, plain HyperDex performs the same individual operations without any transactional guarantees.

With a low rate of conflicts between transactions, Madaba achieves up to 28% of the throughput of plain HyperDex. This shows the overhead managing locks from the client library - each logical operation requires multiple round trips between the client and the servers.

Benchmarks vary in the rate of conflicts, to show the increasing cost of synchronization when multiple processes contend for the same objects. With a higher conflict rate, more transactions are delayed or retried.

The first benchmark runs on 10 million different keys. Each transaction randomly selects 10 keys to read and 10 keys to update. Because the range of keys is large, the rate of conflicts for Madaba transactions is negligible. This shows the cost of acquiring locks via performing writes. Figure 4 shows the number of transactions per second achieved by the system on varying cluster sizes from 5 servers to 30 servers. Figure 5 shows the relative throughput of plain Hyperdex compared to Madaba. The graph is fairly flat, showing that Madaba is able to scale horizontally as well as HyperDex, albeit with an overhead of 4x. This overhead comes from the multiple HyperDex calls that must be made for an operation in Madaba. Additional reads of transactions records, and additional operations to manage locks increase the number of round trips between the client and servers. Even with the optimizations previously discussed, performing a write still requires 4 round trips in the normal case, reading the object, reading the transaction record which holds the lock, updating the object's lock, and reading the transactions record to ensure it is still alive. Performing a read

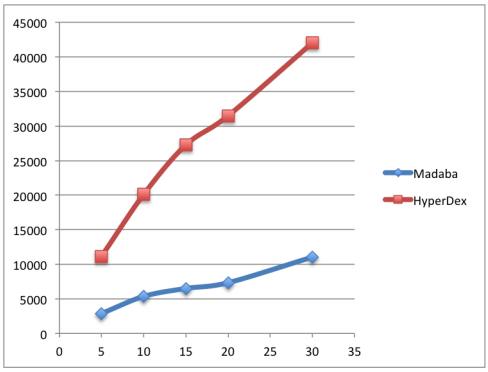

Figure 4: Transactions per second.

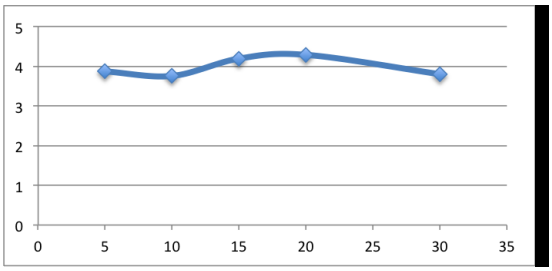

Figure 5: Relative throughput

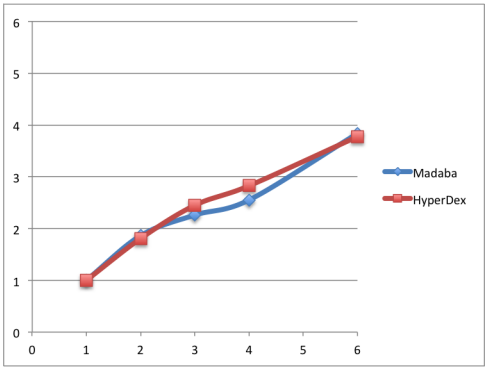

Figure 6: Scale-up

requires the same number of operations as a write in the normal case, though one round trip can be skipped if a read lock already exists for the object.

Both HyperDex and Madaba exhibit sublinear scale-up, as shown in Figure 6. Because operations were performed synchronously, maximal throughput on larger clusters was achieved by running 600 concurrent clients across 30 machines, each with one pending operation at a time. The system may have been limited by the number of clients that were required. Despite this, the scale-up of Madaba matches the scale-up of HyperDex, which was the goal of the system.

The second benchmark shows the performance of Madaba as contention increases. As in the first benchmark, each transaction randomly reads 10 keys and updates 10 keys. To increase contention, the range of keys used is decreased. With a smaller range of keys, the number of conflicts increases, so more transactions are delayed and/or restarted. Figure 7 shows the relative throughput as the range of keys is decreased. With 600 concurrent transactions running on a key range of 100000, each object has an average of 0.12 transactions contending on it at any given time. Since each transaction chooses 10 keys to write, the probability that the other 599 clients each choose 20 keys in a way that does not conflict is less than 31%. With this level of contention, the system achieves 85% of the system throughput without contention. With the key range decreased to 10000, there are an average of 1.2 transactions contending on each key, and the probability of choosing 10 keys unaffected by other transactions is significantly less than one hundredth of a percent. This lowers throughput to 22% of the system throughput without contention. In the most extreme case of a key range of 1000, there are an average of 12 transactions competing on each object, and the probability that any two transactions will have overlapping key sets is over 65%. The throughput in that case is less than 3% of the system throughput without contention, but the system remains live, and each client completes some transactions.

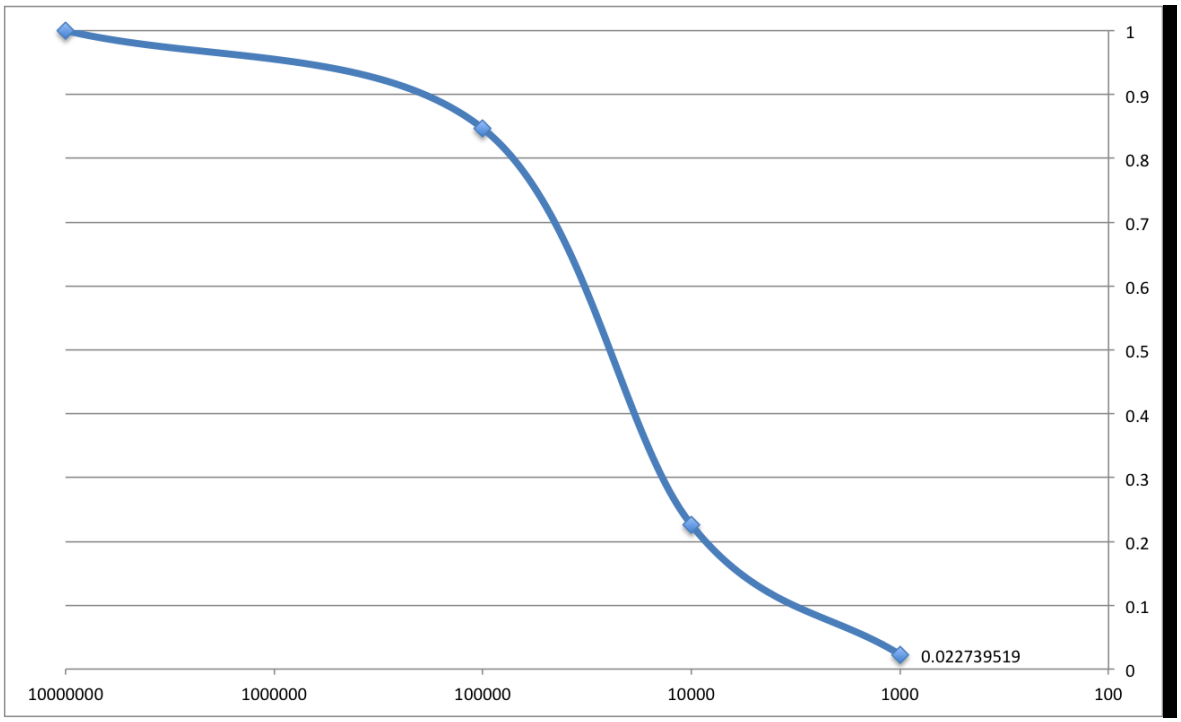

Figure 7: Relative throughput as the contention increases.

6 Conclusion

Madaba demonstrates that strong consistency and liveness guarantees can be achieved with transactions managed completely by the clients of sharded key value stores which only guarantee the consistency of single-key operations. Moreover, it demonstrates that this can be achieved with a decentralized approach to concurrency control which maintains the scalability of the underlying data store. Similarly to Percolator, the decision to manage transactions in the client library introduces a significant performance overhead, but the approach can be used to extends the semantics of existing database, especially those which are offered exclusively as a service.

When contrasted with optimistic approaches to concurrency control, Madaba would have the largest benefit for workloads in which transactions occasionally have read dependencies on objects which are frequently updated. Such transactions can be forced to retry indefinitely if an object is updated between when it is read and when it is validated.

Alternatively, Madaba is much less efficient for read operations, especially for reads of objects that are not frequently modified. Ensuring strict serializability of reads requires performing write operations in the underlying key-value store, and ensuring serializability of reads within failed transaction attempts requires performing them synchronously. The optimistic approach used in Warp achieves strictly serializable

reads with a validation step that takes place in two passes between the relevant servers.

All operations in Madaba are strongly consistent, which is not necessary for all workloads. Allowing for a more flexible consistency model could enable many performance optimizations. For example, if Madaba were to allow inconsistent reads within transactions that fail (which is similar in systems based on OCC), read locks could be deferred to a validation step immediately before committing, and they could be obtained in parallel. Exploiting that parallelism could significantly improve performance without affecting starvation freedom. Similarly, allowing for snapshot isolation of reads could also reduce the performance overhead for read operations. Extending Madaba with the option to enable both of those relaxations on a per-key basis could make the system much more practical for many workloads.

To the best of the author's knowledge, Madaba is the first system that provides a fully decentralized approached to client-managed transactions with both starvation freedom and a strongly serializable consistency model.

Bibliography

[1] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[3] 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. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[4] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Lightweight multi-key transactions for key-value stores. Technical report.

[5] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[6] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[7] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[8] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, June 1978.