SCALABLE DISTRIBUTED TRANSACTIONS ACROSS HETEROGENEOUS STORES

TECHNICAL REPORT 696

AKON DEY, ALAN FEKETE, UWE RÖHM

JULY 2014
Scalable Distributed Transactions across Heterogeneous Stores

Akon Dey ¹  Alan Fekete ²  Uwe Röhm ³

School of Information Technologies, University of Sydney
NSW 2006 Australia

¹ akon.dey@sydney.edu.au ² alan.fekete@sydney.edu.au ³ uwe.roehm@sydney.edu.au

June 19, 2014

Abstract

Modern cloud computing systems usually provide a highly scalable and fault-tolerant data store that sacrifices other features. Often, these systems may not support transactions at all or else restrict transactions to one data item each. Recently techniques to support multi-item transactions in these types of systems have been successfully developed but have focused on transactions across homogeneous data stores. However, applications often need to store different data in different storage systems perhaps for legacy or interoperability reasons. We propose an approach that enables multi-item transactions across multiple heterogeneous data stores using only a minimal set of commonly implemented features such as single item consistency, conditional updates, and the ability to store additional meta-data. We define an client-coordinate transaction commitment protocol that does not rely on a central coordinating infrastructure. We implement this as a Java library, we call Cherry Garcia (CG), that supports data store abstractions to Windows Azure Storage (WAS), Google Cloud Storage (GCS) and our own high-performance key-value store called Tora.

1 Introduction

Cloud computing infrastructures have emerged as the preferred deployment platforms for a wide variety of applications including web-based applications. Increasingly, desktop and mobile applications are using cloud infrastructure such as data stores to take advantage of proven high-availability and scalability characteristics. In the past, these type of systems used local or centralized, mostly relational databases to store information and application state. There are many new applications that share some or all their data with applications running on other hosts or in the cloud and use these cloud-based data stores for data persistence.
The data management infrastructure available in the cloud such as Google Cloud Storage (GCS), Windows Azure Storage (WAS), Amazon SimpleDB, and others are simple to setup, access, and require little system administration. They scale in and out seamlessly and are highly available and fault-tolerant.

However, typical cloud-based distributed data stores have some limitations. Firstly, there is limited capability to query the data, often restricted to access via the primary key. Complex queries that involve other fields or joins may not be possible. Secondly, the services often provide no transactions or only transactions that involve access to a single record.

The limited query capability is usually worked around by designing new data structures and using different data access patterns. However, the weak transaction guarantees are a severe hindrance to application development and requires major application refactoring and may ultimately prevent the application from being deployed using these technologies.

There are a few ways to support multi-item ACID [12] transactions when using a data store that provides only single-item transaction guarantees. The obvious choice is for every application to manage transactional access to the data store. This is complex, prone to programmer error, and almost certainly results in incorrect behavior as the application evolves.

One way to resolve this is to implement transaction support in the data store itself. This approach is complicated and difficult to implement without sacrificing scalability and availability.

Another approach is to use middleware to coordinate transactional access to the data store. This approach has similar implementation complexities but is suitable for situations where the applications are deployed in a controlled environment.

A different approach is to define a transactional access protocol to each data store and provide a transaction and data store abstraction API to enable the client applications to access the data with transactional semantics while continuing to take advantage of scalable and reliable access to the data store.

We use the latter to implement a library, we call Cherry Garcia, with an easy-to-use API that defines a client coordinated transaction management protocol with a pluggable data store abstraction layer enabling it to handle transactions across more than one heterogeneous data store.

In this paper we make the following contributions:

- We define a client coordinated transaction protocol to enable efficient multi-item transactions across heterogeneous key-value store by distributed applications.
- We describe the implementation of our library in Java that uses the protocol to perform multi-item transactions across Windows Azure Storage (WAS), Google Cloud Storage (GCS) and our own implementation of a high-throughput key-value store with an optimized RESTful interface over an HTTP extension we call Tora.
2 System Design

Distributed NoSQL data stores do not support multi-item transactions. They typically support lower consistency guarantees often with support for single-item transaction in the form of test-and-set operations. This is difficult to write programs using such an Application Programming Interface (API) without being prone to making errors. Further more, maintenance of applications using these systems is often plagued by difficulty due to ad-hoc use of these features.

Application developers are faced with the following problems when programming against data stores:

- Lack of or limited transactional guarantees.
- Limited record locking capability without an ability to avoid or handle deadlocks.
- Lack of a standard API, making it hard for application developers to implement transactions across different, heterogeneous data stores.

```java
public void UserTransaction() {
  Datastore cds = Datastore.create("credentials.xml");
  Datastore gds = Datastore.create("goog_creds.xml");
  Datastore wds = Datastore.create("mstfcreds.xml");
  Transaction tx = new Transaction(cds);
  try {
    tx.start();
    Record saving = tx.read(gds, "saving");
    Record checking = tx.read(wds, "checking");
    int s = saving.get("amount");
    int c = checking.get("amount");
    saving.set("amount", s - 5);
    checking.set("amount", c + 5);
    tx.write(gds, "saving", saving);
    tx.write(wds, "checking", checking);
    tx.commit();
  } catch (Exception e) {
    tx.abort();
  }
}
```

Listing 1: Example code that uses the API to accesses two data stores

It is evident that current systems do not address the problem of transaction coordination across heterogeneous data stores. In this section, we describe the design of our client-coordinated transaction commitment protocol that enables transactions involving multiple data items that span multiple heterogeneous data store instances.

The protocol depends on the following capabilities of modern distributed key-value data stores:

1. Lack of or limited transactional guarantees.
2. Limited record locking capability without an ability to avoid or handle deadlocks.
3. Lack of a standard API, making it hard for application developers to implement transactions across different, heterogeneous data stores.

2.1 Client-coordinated transaction protocol

The protocol is implemented in a library which exposes an API that abstracts data store instances using a Datastore class that is can be accessed through a transaction coordinator class called Transaction. Each data record is addressable using a string key and its value is accessed using the Record class.

An example of a program that uses this API to access data records residing in two heterogeneous data stores instances, one in Windows Azure Storage (WAS) and one in Google Cloud Storage (GCS), is show in Listing 1.
- single-item strong consistency (always read latest version of the data item)
- atomic conditional update and delete on single items, similar to Test-and-Set
- ability to add user-defined meta-data to the content of a data item; used to tag each version with the information about the creating transaction, and also to include both current and preceding version within the item
- a data store that can support global read-only access to records, to make transaction state visible to all clients

In essence, our protocol calls for each data item to maintain the last committed and perhaps the currently active version for the data and meta-data. Each version is tagged with the information pertaining to the transaction that created it. This includes the transaction commit time and transaction identifier that created it pointing to a globally visible transaction status record (TSR) using a Universal Resource Identifier (URI). The TSR is used by the client to determine which version of the data item to use when reading it, and so that transaction commit can happen just by updating (in one step) the TSR. The transaction identifier, stored in the form of a URI allows any client regardless of its location to inspect it the TSR in order to determine the transaction commitment state. Using the status of the TSR, any failure can be either rolled forward to the later version, or rolled back to the previous version. The test-and-set capability on each item is used to determine a consistent winner when multiple transactions attempt concurrent activity on a conflicting set of items. A global order is put on transactions, through a consistent hash of the record identifiers, in order to prevent deadlocks. This approach is optimized to have parallel processing of the commit activity. We describe the details of the algorithms in the remaining part of this section.

### 2.1.1 Transactions

The transaction start time, $T_{start}$, is set at the start of the transaction and is used to select the correct version of the records read from the data stores. The transaction commit timestamp, $T_{commit}$, to tag all the records that belong to the transaction write set. At transaction start, the transaction start time, $T_{start}$, is recorded using a reliable source of time.

Data items are read from individual containing data stores and cached in an in-memory cache in Datastore abstraction provided by the library. The items modified during the course of the transaction are kept in the library cache until the transaction is committed.

Once the application has performed all the updates to the data items the changes are committed to their respective data stores. The transaction commit operation is performed in two stages of processing.

**Stage 1:** The current timestamp is obtained using the TrueTime API to set the transaction commit time, $T_{commit}$. Then every dirty record in the client’s record cache is stamped with the commit timestamp meta-data. Each record is marked with a PREPARED flag and the TSR URI then written to their
Stage 2: If all the records have been successfully written, a Transaction Status Record (TSR) is written (using a test-and-set operation) to a data store that is globally readable to indicate that the transaction is considered as committed. Any future failure will be recovered by rolling forward.

Once the TSR is created, the records in the write set are marked with a COMMITTED tag and written to their respective data stores. This operation can be happen in parallel to improve performance. The TSR may be deleted lazily once all records have been committed.

As is evident in the description above, this approach does not use any centrally managed infrastructure to perform transaction commits. The underlying key value store and its inherent features are used to implement transaction across multiple records that may reside on different individual stores. As there is no need to install or maintain any central infrastructure it is suitable for use across heterogeneous data stores that can span multiple data centres across geographical regions.

The system is implemented as a runtime that is linked to the client application. The architecture of the library is described in Figure 1. The library provides programming abstraction for transactions and data stores. This is done through the implementation of a Transaction and Datastore class in the library respectively.

In the remaining part of this section we describe the various components of the library and and their structure.

2.1.2 Time

Timestamps are acquired from a reliable source of time abstracted using the TrueTime API [7] now(). The API defines time as a timestamp, \( t \) and an error
margin, $\epsilon$ associated with it. The time at any point is captured as an interval, $t \pm \epsilon$. Two timestamps are compared using the `before()` and `after()` methods are used to compare timestamps. They return boolean values `true` or `false`. In the remaining part of this paper, all timestamps are obtained and compared using the TrueTime API.

### 2.1.3 Data record structure

The data record has a record header and data. The header is a collection of field names and their values. The fields are:

- **valid time start** ($T_{\text{valid\_start}}$): The timestamp after which the version of the record is considered committed if in the COMMITTED state.
- **valid time end** ($T_{\text{valid\_end}}$): The timestamp after which the record is considered invalid. This is used to indicate that a record is deleted.
- **lease time** ($T_{\text{lease\_time}}$): The transaction lease time to complete the transaction commit. The record state recovery is executed if the lease time expires and the record is in the PREPARED state.
- **transaction identifier** (**TxID**): Signifies the URI of the transaction that last updated the record. The transaction URI can be examined to determine the fate of the transaction.
- **transaction state** (**TxState**): The last update state of the record, whether PREPARED or COMMITTED.
- **last update time** ($T_{\text{last\_update}}$): The last update timestamp on the data store server.
- **version tag** (**ETag**): A data store generated version tag for the version of the record. This tag can be used to perform conditional writes and reads.
- **previous version** (**Prev**): A reference to the previous version of the record.

### 2.1.4 Data store abstraction

The Datastore class implements the following methods:

- **start()** - start a transaction
- **read(key)** - read a record with the specified key from the data store.
- **write(key, record)** - write a record with the specified key to the data store.
- **prev(key, record)** - return the previous version of the record passed and associated with the specified key from the data store.
- **delete(key)** - delete a record identified by the key from the data store.
- **prepare(key, record)** - if the version of the record matches the version on the data store or if the record is a new record, write the contents of the record and its previous version to the data store and mark it with the PREPARED state.
- **commit(key, record)** - update record identified by the key to the COMMITTED state.
• **abort(key, record)** - restore the record identified by the key to the previous **COMMITTED** state if it is in the **PREPARED** state.

• **recover(key)** - recover the record identified by the key if it not in the **COMMITTED** state.

### 2.1.5 Transaction abstraction

The Transaction class implements the following methods:

- **start()** - start the transaction
- **read(ds, key)** - read a consistent version of the record specified key from the data store and cache it in the transaction cache.
- **write(ds, key, record)** - write the record to the transaction cache to be persisted in the data store specified by ds using transaction commitment protocol.
- **delete(ds, key)** - mark the record identified by the specified key as deleted in the transaction cache to be deleted from the data store at commit time.
- **commit()** - commit all the changes to the data stores using the commit-ment protocol.
- **abort()** - abort the transaction
- **recover(txid)** - return the status of the transaction. The returned state can be used to rollforward or abort the transaction.

The transaction context is available to the application in the form of an object of the Transaction class. It has a transaction record cache to cache data items that have been read from and/or are to be written to their originating data stores. The records in the cache are addressed using a combination of the record key and a data store identifier.

In addition to this, the transaction context must persist the TSR so that other clients can accessing the same data simultaneously can refer to it. The TSR is a special type of data record whose identifying key is its transaction identifier in the form of a Universal Resource Identifier (URI) whose key is Universally Unique Identifier (UUID) of the form http://⟨host⟩:⟨port⟩/⟨uuid⟩. It is persisted in a data store that is called the Coordinating Data Store (CDS) and must be globally readable. It must be noted that the choice of the CDS is determined by the application and is suitable as long as the global read visibility constraints are fulfilled.

**Start transaction:** The transaction is started by creating a unique transaction identifier using a UUID generator and setting the transaction start time \(T_{\text{start}}\) to the current time using the TrueTime API.

**Transactional read:** The record is read from the data store using the supplied key. The data store populates the record header and contents that can be used by the transaction code. The record state is checked to see if the transaction has been committed i.e. if it is in the **COMMITTED** state. If it is in the **COMMITTED** state, the \(T_{\text{valid~time}}\) record header is compared with the current transaction start time, \(T_{\text{start}}\). If the version of the record read is created
after the start of the current transaction, the previous version is obtained using
the Datastore *prev()* method. If the record is in the PREPARED state, the
transaction URI is used to inspect the state of the TSR. If the TSR exists, the
record version is considered committed. If the transaction lease time has expired,
the record is rolled forward and marked COMMITTED. If the transaction lease
time on the data record has not expired, the current application transaction is
aborted.

Once a valid version of the record is read from the data store, it is put into
the transaction record cache and then returned to the caller.

**Transactional write:** Transactional writes are simple. The record associ-
ated to the key is written to the Transaction object cache. If an earlier version
exists it is marked as the previous version. The record is written to the data
store at transaction commit time.

### 2.1.6 Transaction commit

The transaction commit is performed in two stages.

**Prepare:** The record cache is inspected and all dirty objects are inserted
into the write-set. Each The record in the write-set is marked with the transac-
tion status record URI, the transaction commit time, and the transaction state
is set to PREPARED then conditionally written to the respective data store in
the order of the hash values of the record identifying key. This is done by using
the *Datastore.prepare()* method which performs a conditional write to the data
store using the record version tag (ETag). The prepare phase is considered to
be successful if all dirty records are successfully prepared.

**Commit:** The TSR is written to the coordinating data store to indicate
that all the records have been successfully prepared. The records are then
committed by calling the data store *commit()* method in parallel. The record
commit method marks the record with the COMMITTED state. Once the
records are committed the transaction status record is deleted asynchronously
from the coordinating data store.

### 2.1.7 Transaction abort

If the transaction commit operation has not been initiated the abort operation
is trivial. The record cache is cleared and the transaction is marked as aborted.

### 2.1.8 Deadlock detection and avoidance

Concurrent conflicting transactions prepare the records in the same order as
they use the hash values of the record keys. Only one of the transactions will
succeed in performing a conditional write operation to the data store. The other
transaction aborts after rolling back its prepared records.
2.1.9 One-phase transaction commit optimization

If there is just one data item in the write-set of the transaction, the prepare-phase of the commitment protocol can be avoided and the commit operation can be performed on the single data item in one phase. The TSR is written before the data item is written to ensure that the transaction is marked as committed.

If the transaction is partially committed the transaction is aborted if the TSR has not been written to the transaction coordinating data store. In this case, the abort is performed by undoing the record prepare operation by writing the previous version of the record to the data store. Once the transaction status record has been written to the coordinating data store the transaction cannot be aborted.

2.1.10 Transaction recover

Transaction recovery is performed lazily in case of application failure. The transaction state is inspected in the record header to see if the record is in the PREPARED or COMMITTED state. If the record is in the COMMITTED state the recovery is unnecessary. If the record is in the PREPARED state, the record is rolled forward and committed if the transaction status record (TSR) exists and rolled back otherwise. The rollforward is performed by marking the record with the COMMITTED state. A rollback is performed by overwriting the record with its previous versions.

There are a number measures we have used in order to improve the performance of the library and reduce TCP/IP connection overheads we use a connection pool for each data store endpoint. In addition, we use a thread pool to implement parallel asynchronous writes and asynchronous deletes. The current version does not perform one-phase commit optimization for transactions with only one object in the write-set. We are exploring ways to further improve the performance.

3 Implementation

In this section, we describe the implementation of the Cherry Garcia Java library and its various components. Additionally, we describe the Datastore abstraction for Windows Azure Storage (WAS), Google Cloud Storage (GCS) and Tora, a high-performance key-value store with a extended HTTP interface built using a WiredTiger [21] storage engine. We also give an overview of the implementation of Tora.

In this version of our implementation, timestamps are obtained using an implementation of the TrueTime API over local server time. This gives us the ability to switch to a more reliable time system in the future.

The Cherry Garcia library is implemented in Java using the standard Java classes available with JDK 1.6 and the Apache HTTP client library.

\[1\text{https://hc.apache.org/}\]
3.1 Datastore abstraction for WAS

The Datastore abstraction for WAS is implemented using the Apache HTTP client library version 4.3. An earlier implementation using the WAS Java client was much slower than using the REST API directly using the Apache HTTP client library version. In addition, the WAS Java Client performs it own data item caching making it harder to control data item life cycles.

We used the Apache HTTP client and commons library to implement the Datastore abstraction called AzureDatastore. It uses the WAS REST API to read (GET), write (PUT) and delete (DELETE).

Each record is stores along with its previous version which is stored as a serialized Java object for sake of implementation simplicity. The record headers are stored as meta-data headers supported by WAS. Each protocol-defined header is prefixed with the string `$x-msft-meta` HTTP header to enable the record header field to be stored.

The previous version of object is stored at the end of the object data and is pointed to by an additional header field called `$x-msft-meta-previous_offset`. In the current version of the code the previous version of the object is serialized using Java serialization. This is clearly not optimal but easy to implement. We plan to use a more light-weight and portable serialization format like Google Protocol Buffers or Twitter’s Thrift.

The `prepare()`, `commit()`, `abort()` and `delete()` methods use the conditional write operations supported by the REST API as described in the Appendix A. We take advantage of the conditional write capabilities supported using the `If-Match` and `If-Unmodified-Since` HTTP request headers supported by the REST API.

The `prepare()` method for WAS is implemented by performing a conditional write of a new version of the record along with a copy of the previous version to the data store with the `transaction state` header set to the string "PREPARED". This is done by setting the value of the `If-Match` HTTP header to the ETag of the previously read version of the record. This ensures that the record is written only if no other client had written a new version of the record. The PREPARED header setting ensures that any reader must ether recover the record or use the older version. If the object is a new object the `If-Unmodified-Since` header with a value set to "Thu, 01 Jan 1970 00:00:00 GMT". This ensures that no object version was written before the one being written preventing lost updates.

The `commit()` operation is implemented using a PUT operation on the record using the `If-Match` header with the ETag returned during the `prepare()` or `recover()` operation. This ensures that the version prepared by the client was the one that is issued the commit request. The `transaction state` set to the string "COMMITTED" along with the `transaction commit time` and other headers respectively. Once, committed the record can be read and used by the data store client without needing recovery.

A prepared operation on a record is aborted using the `abort()` which is implemented by writing the previous version of the record read from the data store. In order to ensure that the version read is the one being aborted, the `If-Match`
header is used using the \textit{ETag} of the record previously read.

Similarly, the delete operation is implemented using the \texttt{delete()} method uses the DELETE HTTP method with the \textit{If-Match} header set to ensure that only the version read by the client is deleted.

WAS performs a form of request rate limiting to prevent abuse. This results in HTTP error code 503 (\textit{Resource temporarily unavailable}) when the number of concurrent connections and request rates exceed a preset limit. In addition to rate limiting, it appears as though there is some form of traffic shaping that takes place as well. We observed that before the actual request failures begin occurring, request responses slow down ultimately failing with error code 503.

The impact of rate limiting was significant in our experiments. It prevented us from attempting to scale beyond 16 threads for most microbenchmark tests due to the unreliable pattern of failures. This may be considered to be a testament to the design of the rate limiting system which happens to perform a reasonably good job of allowing us to use the system reasonably while also ensuring that we could not abuse it.

We used non-georeplicated containers for our tests to prevent the impact of georeplication from affecting the performance of the test application. The impact of latency was easily visible depending on the physical location of the client and the container.

### 3.2 Datastore abstraction for GCS

Google implements its own Java HTTP client that is similar to the Apache HTTP client library to access Google Cloud Storage and other Google Cloud services. The API is subtly different and the access control is handled quite elegantly by the library. This is why we use it to access GCS.

In order to enable consistent writes and allow conditional operations, we use the object versioning provided with the GCS 2.0 XML REST API for our implementation. With versioning set, every object is assigned a \textit{generation number} returned with the \texttt{x-goog-generation} response header when a record is read. This is essentially a high-resolution (nanosecond precision) timestamp of the record.

We use this \textit{generation} number as a unique record version identifier along with the \texttt{x-goog-if-generation-match} request header to perform conditional updates on data items. After an object is read, its \texttt{x-goog-generation} header is captured and treated like an \textit{ETag} in the case of WAS and passed as the value of the \texttt{x-goog-if-generation-match} request header to implement conditional PUT and DELETE operations.

As with WAS, the \texttt{prepare()}, \texttt{commit()}, \texttt{abort()} and \texttt{delete()} methods are implemented using these conditional HTTP operations supported by the GCS REST API to enable the algorithms described in the Appendix A to work.

Like WAS, the \texttt{prepare()} method for GCS is implemented by performing a conditional write of a new version of the record along with a copy of the previous version to the data store with the \textit{transaction state} header set to the string
"PREPARED". This is done by setting the value of the $x$-goog-if-generation-match HTTP header to the value returned in the $x$-goog-generation header when the record was previously read. This ensures that the record is written only if no other client had written a new version of the record. The PREPARED header setting ensures that any reader must either recover the record or use the older version. If the object is a new object the $x$-goog-if-generation-match header with a value set to 0. This ensures that no object version was written before the one being written preventing lost updates.

In GCS, the commit() operation is implemented using a PUT operation on the record using the $x$-goog-if-generation-match header with the value returned with the $x$-goog-generation header returned during the prepare() or recover() operation ensuring that the version prepared by the client is the one that is issued the commit request. The transaction state set to the string "COMMITTED" along with the transaction commit time and other headers respectively. Once, committed the record can be read and used by the data store client without needing recovery.

A prepared operation on a record is aborted using the abort() method which is implemented by writing the previous version of the record read from the data store. In order to ensure that the version read is the one being aborted, the $x$-goog-if-generation-match header is used as it is done with prepare() and commit().

Similarly, the delete() method uses the DELETE HTTP method with the $x$-goog-if-generation-match header appropriately set to ensure that only the version read by the client is deleted.

Like in the case of WAS, both the current and previous versions of the object are stored together in the data store. The $x$-goog-meta- prefix is used to store record headers and an offset to the previous version that is stored in a serialized Java object format.

Rate limiting is performed by the service at two levels. The token approval is restricted resulting in HTTP failure code 403 (Forbidden) while service level rate limiting results in HTTP error code 503 (Resource temporarily unavailable).

### 3.3 Tora: a transaction-aware NoSQL data store

We have developed a high-performance key-value data store that will have native support for native 2-phased updates and deletes without compromising the scalability and availability characteristics of typical distributed NoSQL data stores. We are of the opinion that these extensions to the traditional API with the operations; PREPARE, COMMIT, ABORT, and RECOVER in addition the the standard GET, PUT, and DELETE methods will enable the client-coordinated transaction commitment protocol to work more efficiently by reducing the number of API calls to the data store while continuing to support traditional non-transactional access using only the basic set.

Tora is a key-value store that implements this API to store blobs addressable using a key. It is written in C++ using the Boost ASIO library (used to achieve asynchronous I/O operations) to implement the socket and HTTP interface and
uses the transactional capabilities of WiredTiger, a high-performance embedded key-value store, to persist data in a transactional manner.

3.4 Datastore abstraction to Tora

The extended transactional support to HTTP provided by Tora makes it much easier to build the Datastore abstraction to it. The standard GET, PUT, DELETE HTTP methods are used for read(), write() and delete() methods for the Datastore class implementation. The prepare() method uses the extended HTTP PREPARE method provided by Tora. Similarly, the commit(), abort() and recover() methods in the Datastore class use the COMMIT, ABORT and RECOVER extended-HTTP (called REST+T) methods respectively.

4 Related Work

In recent years there have been numerous implementations of distributed key-value stores, each exhibiting different mix of performance, scalability, availability characteristics and alternate architectures. These include Amazon Dynamo [10], Google BigTable [5], Yahoo! PNUTS [6], HBase, and Cassandra [14], Amazon SimpleDB, Amazon S3. These systems use commodity hardware and exhibit scalability and high-availability but provided lower consistency guarantees, often limited to only eventual consistency [20]. Typically only single item transactions are guaranteed and query capability is limited. Data is accessed using the primary key and data scan support is limited to only a few systems like PNUTS and SimpleDB.

More recently, there have been developments like Spinnaker [19], Windows Azure Storage [4], Google Cloud Storage provide single item consistency guarantees. The system design focuses on scalability, performance and single key consistency. Spinnaker uses a Paxos-based protocol to ensure consistency while COPS [16] and Granola [8] use replication protocol optimizations to achieve greater performance while supporting native multi-item transactions.

Despite these advances, the bulk of the systems leave multi-item transactional data access to the client application. This is prone to programmer error and the results are often completely incorrect.

In order to address these issues, some systems have implemented a relational database engine to provide the query capabilities and transaction support with the raw data stored in a distributed key-value store [3]. This is suitable for applications that require an RDBMS for persistence with the advantage that it provides a complete SQL interface with full transaction support. The performance and transaction throughput of the system is limited only by the underlying queue implementation.

Most applications built using key value stores work well because of the relative simplicity of the programming interface to the data store. Many of these applications use write-once-read-many (WORM) data access to the key value store and function well under the eventual consistency setting. However, there
are applications that are built to run on the same data that require better consistency across multiple keys.

The first approach to address this issue is to implement transactional capabilities within the data store itself. The data store manages the storage as well as transaction management. The Spanner [7] is a distributed data store that supports transactions.

The COPS [16] and Granole [8] implement the distributed key-value store with a custom API to enable applications to transactional access the data store. Similarly, HyperDex Warp [11] is a high-performance distributed key-value store that provides a client library that supports linearizable transactions. The client library simplifies the access to the data items on behalf of the application using an API provided by the data store which maintains multiple versions of each data item. These systems support transactions across multiple keys with a distributed, homogeneous key-value store. The focus of these systems is to build a better, more capable distributed data store and optimize the transaction coordination across it.

Another way is to use middleware to provide caching and transactional data access. Google Megastore [1] is a transactional key-value store built on top of BigTable. Records are collocated in a tree structure called entity groups where each leaf record is associated with the root record using a foreign key. The foreign key is used to cluster related records and partition data across storage nodes and transactions across records spanning clusters is done using 2PC.

In G-Store [9], related data items are grouped into key groups that are cached locally on a single node. Transactions are only allowed within key groups and keys are allowed to migrate from one key group to another using a group migration protocol. Greater transaction throughput is achieved because data items are cached on the local node.

Deuteronomy [15] unbundle the data storage and transaction manager into two separate entities. It defines a protocol, which is an optimization of their earlier work [17], to perform transactional data access using the transaction component (TC) and the data component (DC). This system allows multiple hybrid data stores to be used.

The CloudTPS [22] design uses data store access through a transaction manager split across multiple nodes into local transaction managers (LTM). LTM failures are handled using transaction logs replication across LTMs.

The middleware approach works well when the application is hosted in the cloud and there is a known and controlled set of data stores used by the application. They perform well in this situations and provides one programming interface to the application simplifying the data store access. However, these systems require to be setup and maintained separately.

This is not suitable in our use case where individual application instances need the hands-off, low maintenance features of key value stores and each may have different access privileges to individual data stores.

Another way of implementing multi-key transaction support for distributed key-value stores is to incorporate the transaction coordinator into the client. We know of two implementations that use this approach. Percolator [18] implements
multi-key transactions with snapshot isolation semantics [2]. It depends on a central fault-tolerant timestamp service called a timestamp oracle (TO) to generate timestamps to help coordinate transactions and a locking protocol to implement isolation. The locking protocol relies on a read-evaluate-write operation on records to check for a lock field associated with each record. It does not take advantage of test-and-set operations available in most key value stores making this technique unsuitable for client applications spread across relatively high-latency WANs. No deadlock detection or avoidance is implemented further limiting its use over these types of networks.

ReTSO [13] relies on a transaction status oracle (TSO) that monitors the commit of all transactions to implement a lock-free commit algorithm resulting in high transaction throughput. It utilizes a high-reliability distributed write-ahead log (WAL) system called BookKeeper to implement the TSO providing snapshot isolation semantics. Timestamps are generated by a central timestamp oracle. The need to have a TSO and a TO for transaction commitment is a bottleneck over a long-haul network. This prevents this approach to be effective in a WAN layout.

Our approach is similar in many ways to Percolator and ReTSO. We use the transaction start time to obtain the transaction read set. We also use the transaction commit timestamp to tag all the records that belong to the transaction write set. Unlike these systems ours does not depend on any centralized timestamp oracle or logging infrastructure. We utilize the underlying key-value store and its features to provide transaction across multiple records. There is no need to install or maintain additional infrastructure. Our approach enables transactions to span across hybrid data stores that can be deployed in different regions and does not rely upon a central timestamp manager. In the current version, we rely on the local clock to keep time but it is compatible with approaches like TrueTime [7]. We use a simple ordered locking protocol to ensure deadlock detection and recovery without the need of a central lock manager.

5 Future Work

We will explore the HTTP TX protocol further and evaluate it for applications beyond distributed NoSQL key-value data stores including applications in the sphere of the Internet of Things and autonomous smart devices. We intend to publish the details of the HTTP TX protocol, our reference implementation of it and our evaluations of the system.

We have used a simplified version of the TrueTime API [7] implemented at Google in its Spanner system. We have explored the use of a technique similar to the one used by the Simple Network Time Protocol (SNTP) using timestamps over our HTTP extension. We expect to publish the details in future publications.
6 Conclusions

This paper describes a reliable and efficient multi-item transactional data store access protocol. We have implemented that enables ubiquitous client applications to access large-scale distributed key-value data store in a transactional manner. Our evaluations across three different types of data stores spanning multiple data centres and subsequent analysis indicates that this approach is suited for high transaction rates by applications deployed across vastly diverse network latencies.

We describe an extension of the YCSB cloud services benchmark with transaction support, we call YCSB+T. It is suitable for evaluation of transaction access to distributed key-value stores by enabling operations on the data store to be grouped into transactions.

Further, we have given a description of an implementation of our client library in Java we call Cherry Garcia with data store abstraction plugins for Windows Azure Storage (WAS), Google Cloud Storage (GCS) and our own high-performance data store implementation called Tora and have written applications to evaluate it.

The system described here relies on the local system clock to obtain transaction timestamps. However, it can be made to work with the Spanner TrueTime API [7] with little or no modifications. We are currently exploring ways to determine degree of the uncertainty in the local clock on the client using the information available from the data exchanged with the NoSQL data stores participating in a transaction so that it can be used with a system like the TrueTime API.

We are continuing to evaluate our library using YCSB+T on EC2 with data stored in WAS and GCS. Initial results are promising and we will publish the results once we have a more information in future publications. The evaluations conducted so far have focused on YCSB+T instances running in a single cluster. Next, we will evaluate our system in a more heterogeneous setup that spans multiple regions and differing network latencies, bandwidth, and reliability characteristics.

We have also implemented a distributed key-value store called Tora with the extended REST API that will enable the client coordinated transaction protocol to work more efficiently and yet continue to support traditional application use cases. We will describe this system in more detail and share our experiences with it in future publications.

References


A  Algorithms

1: function START
2:     T.identifier ← UUID()
3:     T.start ← now()
4:     T.state ← STARTED
5: end function

1: function READ(datastore, key)
2:     if ∃ T.cache(datastore, key) then
3:         return T.cache(datastore, key)
4:     end if
5:     record ← datastore.read(key)  # read record from data store
6:     if record.state ≠ COMMITTED then
7:         if recover(record.TxID) then  # recover uncommitted records
8:             datastore.commit(key, record)
9:         else
10:             datastore.abort(key, record)
11:         end if
12:     go to 5  # now read the recovered record
13: end if
14:     if record.Tvalid ≥ T.start then
15:         record ← datastore.prev(key)  # read the previous version
16:     if record.Tvalid ≥ T.start then
17:         return NULL
18: end if
19: end if
20:     T.cache.put(datastore, key, record)
21: return record
22: end function
1: function WRITE(datastore, key, record)
2:   record.prev ← T.cache.get(datastore, key)
3:   record.TxID ← T.TxID
4:   T.cache.put(datastore, key, record)
5: end function

1: function COMMIT
2:   $T.commit\_time \leftarrow now()$  \(\triangleright\) phase 1: prepare
3:   $T.lease\_time \leftarrow now() + commit\_timeout$
4:   for (datastore, key, record) ∈ ordered(cache) do
5:     record.$Tvalid\_time \leftarrow T.commit\_time$
6:     record.$Tlease\_time \leftarrow T.lease\_time$
7:     if ERROR = datastore.prepare(key, record) then
8:       abort()
9:     end if
10: end for
11: $T.state \leftarrow COMMITTED$  \(\triangleright\) phase 2: commit
12: coord.datastore.write(T.TxID, T.record)
13: for all (datastore, key, record) ∈ cache.keys() do \(\triangleright\) execute in parallel
14:   datastore.commit(key, record)
15: end for
16: return SUCCESS
17: end function

1: function ABORT
2:   $T.state \leftarrow ABORTED$
3:   coord.datastore.write(T.TxID, T.record)
4:   for all (datastore, key, record) in cache.keys() do \(\triangleright\) execute in parallel
5:     if record.state = PREPARED then
6:       datastore.abort(key, record)
7:     end if
8: end for
9: return SUCCESS
10: end function

1: function RECOVER(TxID)
2:   $tx\_record \leftarrow coord.datastore.read(TxID)$
3: if $\not\exists tx\_record \vee tx\_record.state \neq ABORT$ then
4:   return true
5: end if
6: return false
7: end function