Omid: Lock-free Transactional Support for Distributed Data Stores

Daniel Gómez Ferro* — Splice Machine, Barcelona, Spain — dgomezferro@splicemachine.com
Flavio Junqueira* — Microsoft Research, Cambridge, UK — fpj@apache.org
Ivan Kelly — Yahoo! Research, Barcelona, Spain — ivank@yahoo-inc.com
Benjamin Reed* — Facebook, Menlo Park, CA, USA — br33d@fb.com
Maysam Yabandeh*† — Twitter, San Francisco, CA, USA — myabandeh@twitter.com

***Abstract*—In this paper, we introduce Omid, a tool for lock-free transactional support in large data stores such as HBase. Omid uses a centralized scheme and implements *snapshot isolation*, a property that guarantees that all read operations of a transaction are performed on a consistent snapshot of the data. In a lock-based approach, the unreleased, distributed locks that are held by a failed or slow client block others. By using a centralized scheme for Omid, we are able to implement a lock-free commit algorithm, which does not suffer from this problem. Moreover, Omid *lightly* replicates a read-only copy of the transaction metadata into the clients where they can locally service a large part of queries on metadata. Thanks to this technique, Omid does not require modifying either the source code of the data store or the tables' schema, and the overhead on data servers is also negligible. The experimental results show that our implementation on a simple dual-core machine can service up to a thousand of client machines. While the added latency is limited to only 10 ms, Omid scales up to 124K write transactions per second. Since this capacity is multiple times larger than the maximum reported traffic in similar systems, we do not expect the centralized scheme of Omid to be a bottleneck even for current large data stores.**

I. Introduction

A transaction comprises a unit of work against a database, which must either entirely complete (i.e., *commit*) or have no effect (i.e., *abort*). In other words, partial executions of the transaction are not defined. For example, in WaltSocial [28], a social networking service, when two users A and B become friends, a transaction adds user A to B's friend-list *and* user B to A's friend-list. The support of transactions is an essential part of a database management system (DBMS). We term the systems that lack this feature *data stores* (rather than DBMS). Examples are HBase [1], Bigtable [13], PNUTS [15], and Cassandra [2], that have sacrificed the support for transactions in favor of efficiency. The users however are burdened with ensuring correct execution of a transaction despite failures as well as concurrent accesses to data. For example, Google reports that the absence of cross-row transactions in Bigtable [13] led to many complaints and that they did Percolator [27] in part to address this problem [17].

The data in large data stores is distributed over hundreds or thousands of servers and is updated by hundreds of *clients*, where node crashes are not rare events. In such environments, supporting transactions is critical to enable the system to cope with partial changes of faulty clients. The recent attempts to bring transactional support to large data stores have come with the cost of extra resources [30], expensive cross-partition transactions [27], or cumbersome restrictions [10], making its efficient support in data stores very challenging. As a result, many popular data stores lack this important feature.

Proprietary systems [26], [27] often implement Snapshot Isolation (SI) [11] on top of multi-version database, since it allows for high concurrency between transactions. Two concurrent transactions under SI conflict if they write into the same data item. The conflict must be detected by the system, and at least one of the transactions must abort. Detecting conflicts requires access to *transaction metadata*, such as commit time of transactions. The metadata is also required to determine the versions of data that should be read by a transaction, i.e., the *read snapshot* of the transaction. In other words, the metadata is necessary to make sense of the actual data that is stored in the multi-version data store. The metadata is logically separate from the actual data, and part of the system responsibility is to provide access to the metadata when the corresponding data is read by transactions. Due to the large volume of transactions in large data stores, the transaction metadata is usually partitioned across multiple nodes. To maintain the partitions, previous approaches use distributed locks on nodes that store the transaction metadata and run a distributed agreement algorithm such as two-phase commit [22] (2PC) among them.

The immediate disadvantage of this approach is the need for many additional nodes that maintain the transaction metadata [30].[1] To avoid this cost, Google Percolator [27] stores the metadata along with the actual data and hence uses the same data servers to also maintain the metadata. This design choice, however, resulted in a non-negligible overhead on data servers, which was partly addressed by heavy batching of messages to data servers, contributing to the multi-second delay on transaction processing. Although this delay is acceptable in the particular use case of Percolator, to cover more general use cases such as OLTP traffic, short commit latency is desirable.

Another downside of the above approach is that the distributed locks that are held by the incomplete transactions of a failed client prevent others from making progress. For example, Percolator [27] reports delays of several minutes caused by unreleased locks. The alternative, lock-free approach [24] could be implemented by using a centralized *transaction status*

*The research was done while the author was with Yahoo! Research.
†Corresponding Author

[1] For example, the experiments in [30] use as many transactional nodes (LTM) as HBase servers, which doubles the number of required nodes.

978-1-4799-2555-1/14/$31.00 © 2014 IEEE

oracle (SO) that monitors the commits of all transactions [20]. The SO maintains the transaction metadata that is necessary to detect conflicts as well as defining the read snapshot of a transaction. Clients read/write directly from/to the data servers, but they still need access to transaction metadata of the SO to determine which version of the data is valid for the transaction. Although the centralized scheme allows the appealing property of avoiding distributed locks, the *limited capacity* and *processing power* of a single node questions the scale of the traffic that it could handle:

1) In a large distributed data store, the transaction metadata grows beyond the capacity of the SO. Similarly to the approach used in previous works [6], [12], [20], we can truncate the metadata and keep only the most recent changes. The partial metadata could, however, violate consistency. For example, forgetting the status of an aborted transaction txn_a leaves a future reading transaction uncertain about the validity of the data written by txn_a.
2) The serialization/deserialization cost limits the rate of messages that a single server could send/receive. This consumes a large part of the processing power of the SO, and makes the SO a bottleneck when servicing a large volume of transactions.

In this paper, we design and implement a lock-free, centralized transactional scheme that is not a bottleneck for the large volume of transactions in the current large data stores. To limit the memory footprint of the transaction metadata, our tool, Omid, truncates the metadata that the SO maintains in memory; memory truncation is a technique that has been used successfully in previous work [20], [6]. As we explain in § III, however, the particularity of metadata in Omid requires a different, more sophisticated variation of this idea.

Moreover, Omid *lightly* replicates a read-only copy of the transaction metadata into the clients where they can locally service a large part of queries needed by SI. At the core of the light replication technique lies the observation that under SI the reads of a transaction do not depend on changes to the transaction metadata after its start timestamp is assigned. The SO, therefore, piggybacks the recent changes to the metadata on the response to the timestamp request. The client node aggregates such piggybacks to build a view of the metadata state up until the transaction start. Using this view, the client could locally decide which version of data read from the data store is in the read snapshot of the transaction, without needing to contact the SO. As the experiments in § V show, the overhead of the replication technique on the SO is negligible. Since the client's replica of transaction metadata is read-only, a client failure affects neither the SO nor the other clients.

Main Contributions:

1) We present Omid, an efficient, lock-free implementation of SI in large data stores. Being lock-free, client crashes do not affect the progress of other transactions.
2) Omid enables transactions for applications running on top of data stores with no perceptible impact on performance.
3) Being client-based, Omid does not require changing the data store[2] and could be installed on top of any multi-version data store with a basic API typical to a key-value store. Omid could also support transactions that span tables distributed across heterogeneous data stores.
4) The above features are made feasible in Omid due to (i) the novel technique that enables efficient, read-only replication of transaction metadata on the client side, and (ii) the well-engineering of memory truncation techniques that safely limits memory footprint of the transaction metadata.

The experimental results show that Omid (in isolation from data store) can service up to ∼71K write transactions per second (TPS), where each transaction modifies 8 rows on average, and up to 124K write TPS for applications that require small transactions to update a graph, e.g., a social graph. (These numbers exclude read-only transactions, which are much lighter and do not add the conflict detection overhead to the SO.) This scale is multiple times larger than the maximum achieved traffic in similar data stores [27]. In other words, the traffic delivered by many existing large data stores is not enough to saturate the SO. To extend the application of Omid to a cloud provider that offers services to many independent customers, partitioning can be safely employed since transactions of a cloud user are not supposed to access the other users' data, i.e., each user (or each set of users) can be assigned to a separate instance of the SO.

Roadmap The remainder of this paper is organized as follows. § II explains SI and offers an abstract design of a SO. The design and implementation of Omid are presented in § III and § IV. After evaluating our prototype in § V, we review the related work in § VI. We finish the paper with some concluding remarks in § VII.

II. Background

A transaction is an atomic unit of execution and may contain multiple read and write operations to a given database. A reliable transactional system provides ACID properties: atomicity, consistency, isolation, and durability. *Isolation* defines the system behavior in the presence of concurrent transactions. Proprietary data storage systems [26], [27] often implement Snapshot Isolation (SI) [11], since it allows for high concurrency between transactions. Here, we discuss examples of both distributed and centralized implementations of SI.

Snapshot Isolation SI is an optimistic concurrency control [24] that further assumes a multiversion database [8], [9], which enables concurrent transactions to have different views of the database state. SI guarantees that all reads of a transaction are performed on a snapshot of the database that corresponds to a valid database state with no concurrent transaction. To implement SI, the database maintains multiple versions of the data in *data servers*, and transactions, executed by *clients*, observe different versions of the data depending on their start time. Implementations of SI have the advantage that writes of a transaction do not block the reads of others. Two concurrent transactions still conflict if they write into the same data item, say a database row.[3] The conflict must be detected by the SI implementation, and at least one of the transactions must abort.

[2] As we explain in § IV, the garbage collection of old versions, however, has to be adapted to take into account the values that might being read by in-progress transactions.

[3] Here, we use the row-level granularity to detect the write-write conflicts. It is possible to consider finer degrees of granularity, but investigating it further is out of the scope of this work.

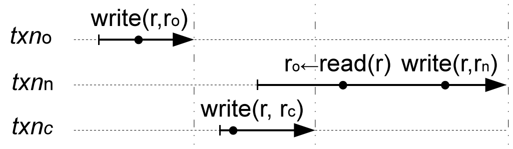


Fig. 1: An example run under SI guarantee. $write(r, v)$ writes value v into data item r, and $read(r)$ returns the value in data item r.

To implement SI, each transaction receives two timestamps: one before reading and one before committing the modified data. In both lock-based and lock-free approaches, timestamps are assigned by a centralized server, the *timestamp oracle*, and hence provide a commit order between transactions. Transaction txn_i with assigned start timestamp $T_s(txn_i)$ and commit timestamp $T_c(txn_i)$ reads its own writes, if it has made any, or otherwise the latest version of data with commit timestamp $\delta < T_s(txn_i)$. In other words, the transaction observes all of its own changes as well as the modifications of transactions that have committed before txn_i starts. In the example of Fig. 1, txn_n reads the modifications by the committed transaction txn_o, but not the ones made by the concurrent transaction txn_c.

If txn_i does not have any write-write conflict with another concurrent transaction, it commits its modifications with a commit timestamp. Two transactions txn_i and txn_j conflict if both of the following hold:

1) *Spatial overlap*: both write into row r;
2) *Temporal overlap*:
   $T_s(txn_i) < T_c(txn_j)$ and $T_s(txn_j) < T_c(txn_i)$.

Spatial overlap could be formally expressed as $\exists r \in rows$: $\{txn_i, txn_j\} \subset writes(r)$, where $writes(r)$ is the set of transactions that have written into row r. In the example of Fig. 1, both transactions txn_n and txn_c write into the same row r and therefore conflict (spatial overlap). Since they also have temporal overlap, the SI implementation must abort at least one of them.

From spatial and temporal overlap conditions, we see that an implementation of SI has to maintain the following transaction metadata: **(i)** T_s: the list of start timestamps of transactions, **(ii)** T_c: the list of commit timestamps of transactions, and **(iii)** ***writes***: the list of transactions that have modified each row. To illustrate the spectrum of different possible implementations of SI, we summarize in the following two main approaches for implementing SI in distributed data stores.

Distributed Implementations In a naïve, distributed implementation of SI, the three transactional lists could be distributed on the clients, where each client maintains its partial copy of transaction metadata based on the transactions that it runs. Since the lists are distributed over all clients, each client must run a distributed agreement algorithm such as two-phase commit [22] (2PC) to check for write-write conflicts with all other clients. The obvious problem of this approach is that the agreement algorithm does not scale with the number of clients. Moreover, the clients are stateful and fault-tolerant mechanisms must be provided for the large state of every client. In particular, the whole system cannot progress unless all the clients are responding.

To address the mentioned scalability problems, the transaction metadata could be partitioned across some nodes. The distributed agreement algorithm could then be run only among the partitions that are affected by the transaction. Percolator [27] is an elegant implementation of this approach, where the transaction metadata is partitioned and placed in the same servers that maintain data (tablet servers in Percolator terminology). The uncommitted data is written directly into the main database. The T_s list is trivially maintained by using the start timestamp as the versions of values. The database maintains the *writes* list naturally by storing multiple version of data. Percolator [27] adds two extra columns to each column family: *lock* and *write*. The write column maintains the T_c list, which is now partitioned across all the data servers. The client runs a 2PC algorithm to update this column on all modified data items. The lock columns provide fine grained locks that the 2PC algorithm uses.

Although using locks simplifies the write-write conflict detection, the locks held by a failed or slow transaction prevent the others from making progress until the locks are released, perhaps by a recovery procedure. Moreover, maintaining the transaction metadata (in lock and write columns) puts additional load on data servers, which was addressed by heavy batching of messages sent to data servers, contributing to the multi-second delay on transaction processing [27].

Transaction Status Oracle In the centralized implementation of SI, a single server, *i.e.*, the SO, receives the commit requests accompanied by the set of the identifiers (id) of modified rows, W [20].[4] Since the SO has observed the modified rows by the previous commits, it could maintain (i) T_c and (ii) *writes* lists, and therefore has enough information to check if there is temporal overlap for each modified row [20]. Here, we present one possible abstract design for the SO, in which timestamps are obtained from a timestamp oracle integrated into the SO and the uncommitted data of transactions are stored on the same data tables. Similar to Percolator [27], the timestamp oracle generates unique timestamps, which could be used as transaction ids, i.e., $T_s(txn_i) = txn_i$. This eliminates the need to maintain T_s in the SO.

Algorithm 1 describes the procedure that is run sequentially to process commit requests. In the algorithm, W is the list of all the modified rows by transaction txn_i, T_c and *lastCommit* are the in-memory state of the SO containing the commit timestamp of transactions and the last commit timestamp of the modified rows, respectively. Note that *lastCommit* $\subseteq$ *writes* and the SO, therefore, maintains only a subset of *writes* list.

Algorithm 1 Commit request (txn_i, W) : {cmt, abrt}

```
for each row $r \in W$ do
    if $lastCommit(r) > T_s(txn_i)$ then
        return abort;
    end if
end for
                                        ▷ Commit $txn_i$
$T_c(txn_i) \leftarrow$ TimestampOracle.next();
for each row $r \in W$ do
    $lastCommit(r) \leftarrow T_c(txn_i)$;
end for
return commit;
```

To check for write-write conflicts, Algorithm 1 checks temporal overlap for all the already committed transactions. In

[4] In the case of multiple updates to the same row, the id of the modified row is included once in W.

other words, in the case of a write-write conflict, the algorithm commits the transaction for which the commit request is received first. The temporal overlap property must be checked on every row r modified by txn_i (if there is any) against all the committed transactions that have modified the row. Line 2 performs this check, but only for the latest committed transaction txn_l that has modified row r. It can be shown by induction that this check guarantees that the temporal overlap property is respected by all the committed transactions that have modified row r.[5] This property greatly simplifies Algorithm 1 since it has to maintain only the latest commit timestamp for each row (Line 8). Also, notice that Line 2 verifies only the first part of temporal overlap property. This is sufficient because the SO itself obtains the commit timestamps in contrast to the general case in which clients obtain commit timestamps [27]. Line 6 maintains the mapping between the transaction start and commit timestamps. This data will be used later to process queries about the transaction status.

A transaction txn_r under SI reads from a snapshot of the database that includes the written data by all transactions that have committed before transaction txn_r starts. In other words, a row written by transaction txn_f is visible to txn_r if $T_c(txn_f) < T_s(txn_r)$. The data of transaction txn_f written to the database is tagged with the transaction start timestamp, $T_s(txn_f)$. Transaction txn_r can inquire of the SO, whether $T_c(txn_f) < T_s(txn_r)$, since it has the metadata of all committed transactions. Algorithm 2 shows the SO procedure to process such queries. (We will show in § III how the clients can locally run Algorithm 2, to avoid the communication cost with the SO.) If transaction txn_f is not committed yet, the algorithm returns false. Otherwise, it returns true if txn_f is committed before transaction txn_r starts (Line 2), which means that the read value is valid for txn_r.

Algorithm 2 inSnapshot(txn_f, txn_r) : {true, false}

```
1: if $T_c(txn_f) \neq$ null then
2:     return $T_c(txn_f) < T_s(txn_r)$;
3: end if
4: return false;
```

Status Oracle in Action Here we explain an implementation of SI using a SO on top of HBase, a clone of Bigtable [14] that is widely used in production applications. It splits groups of consecutive rows of a table into multiple regions, and each region is maintained by a single data server (RegionServer in HBase terminology). A transaction client has to read/write cell data from/to multiple regions in different data servers when executing a transaction. To read and write versions of cells, clients submit get/put requests to data servers. The versions of cells in a table row are determined by timestamps.

Fig. 2 shows the steps of a successful commit. Since the timestamp oracle is integrated into the SO, the client obtains the start timestamp from the SO. The following list details the steps of transactions:

Single-row write. A write operation by transaction txn_r is performed by simply writing the new data tagged with the transaction start timestamp, $T_s(txn_r)$.

Single-row read. Each read in transaction txn_r must observe the last committed data before $T_s(txn_r)$. To do so, starting with the latest version (assuming that the versions are sorted by timestamp in ascending order), it looks for the first value written by a transaction txn_f, where $T_c(txn_f) < T_s(txn_r)$. To verify, the transaction inquires *inSnapshot* of the SO. A value is skipped if the corresponding transaction is aborted or was not committed when txn_r started. Depending on the implementation, this process could be run by the client or the data server.

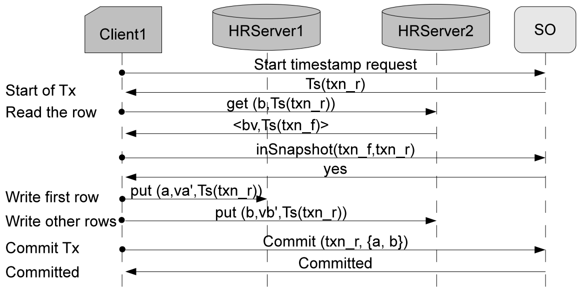


Fig. 2: Sequence diagram of a successful commit. The transaction reads from key ”b” and writes values for keys ”a” and ”b” into two data servers.

Transaction commit. After a client has written its values to the rows, it tries to commit them by submitting to the SO a commit request, which consists of the transaction id txn_w (=$T_s(txn_w)$) as well as the list of all the modified rows, W. Note that W is an empty set for read-only transactions.

(Optional) single-row cleanup. If a transaction aborts, its written values are ignored by the other transactions and no further action is required. To efficiently use the storage space, the client could also clean up the modified rows of its aborted transactions by deleting its written versions.[6] The failure of the client to do so, although does not affect correctness, leaves writes of aborted transactions in data servers. This data will be eventually removed when data servers garbage collect old versions. Alternatively, the SO could actively delegate a daemon to regularly scan the data store and to remove the data written by aborted transactions.

Serializability SI prevents write-write conflicts. To provide the stronger guarantee of serializability, the SO can be modified to prevent both read-write and write-write conflicts. This would lead to a higher abort rate and essentially provides a tradeoff between concurrency and consistency. In this paper, we focus on SI and hence use the SO to prevent only write-write conflicts.

III. Omid Design

In this section, we analyze the bottlenecks in a centralized implementation of the SO and explain how Omid deals with each of them.

System Model Omid is designed to provide transactional support for applications that operate inside a data center. We hence do not design against network partitioning, and in such a case the clients separated by the partitioned network cannot make progress until the SO as well as the data store are reachable again. The transactions are run by *stateless* clients, meaning that a client need not to persist its state and its crash does not compromise the safety. Our design assumes a multi-version

[5] The proof is omitted due to space limit.

[6] We make use of HBase Delete.deleteColumn() API to remove a particular version in a cell.

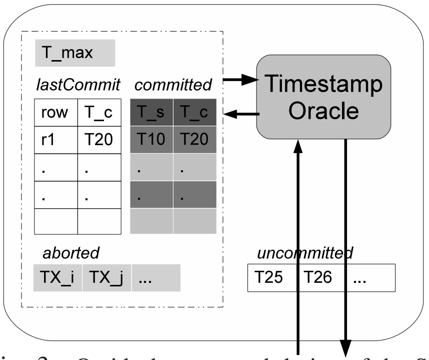


Fig. 3: Omid: the proposed design of the SO.

data store from/to which clients directly read/write via an API typical to a key-value store. In particular, the clients in Omid make use of the following API: (i) put(k,v,t): writes value v to key k with the assigned version t, (ii) get(k,[$t1..t2$],n): reads the list of values assigned to key k and tagged with a version in the range of [$t1..t2$] (the optional number n limits the number of returned values), (iii) delete(k,v): deletes the version v from key k. We assume that the data store *durably* persists the data. We explain in § III that how we provide durability for metadata in the SO.

Design Overview In a large distributed data store, the transaction metadata grows beyond the storage capacity of the SO. Moreover, to further improve the rate of transactions that the SO can service, it is desirable to avoid accessing the hard disk and hence fit the transaction metadata all in main memory. There are two main challenges that limit the scalability of the SO: (i) the limited amount of memory, and (ii) the number of messages that must be processed per transaction. We adjust Algorithms 1 and 2 to cope with the partial data in the main memory of the SO. To alleviate the overhead of processing queries at the SO, the transaction metadata could be replicated on data servers, similar to previous approaches [27]. Omid, on the contrary, benefits from the properties of SI to lightly replicate the metadata on the clients. We show that this approach induces a negligible overhead to data servers as well as clients.

Memory Limited Capacity To detect conflicts, Omid checks if the last commit timestamp of each row $r \in W$ is less than $T_s(\textit{txn}_i)$. If the result is positive, then it commits, and otherwise aborts. To be able to perform this comparison, Omid requires the commit timestamp of all the rows in the database, which obviously will not fit in memory for large databases. To address this issue, similarly to the related work [12], [6], we truncate the *l*astCommit list. As illustrated in Fig. 3, Omid keeps only the state of the last NR committed rows that fit into the main memory, but it also maintains $T_{\max}$, the maximum timestamp of all the removed entries from memory. Algorithm 3 shows the Omid procedure to process commit requests.

If the last commit timestamp of a row r, *lastCommit*(r), is removed from memory, the SO is not able to perform the check at Line 5. However, since $T_{\max}$ is by definition larger than all the removed timestamps from memory including *lastCommit*(r), we have:

$$T_{\max} < T_s(\textit{txn}_i) \Rightarrow \textit{lastCommit}(r) < T_s(\textit{txn}_i) \quad (1)$$

which means that there is no temporal overlap between the conflicting transactions. Otherwise, Line 2 conservatively aborts the transaction, which means that some transactions could unnecessarily abort. Note that this is not a problem if $\Pr(T_{\max} < T_s(\textit{txn}_i)) \simeq 1$, which is the case if the memory could hold the write set of transactions that are committing during the lifetime of *txn*$_i$. The following shows this is the case for a typical setup. Assuming 8 bytes for unique ids, we estimate the required space to keep the information of a row is 32 bytes, including row id, start timestamp, and commit timestamp. For each 1 GB of memory, we can therefore fit data of 32M rows in memory. If each transaction modifies 8 rows on average, then the rows for the last 4M transactions are in memory. Assuming a workload of 80K TPS, the row data for the last 50 seconds are in memory, which is far more than the average commit time, *i.e.*, tens of milliseconds.

Algorithm 3 Commit request ($\textit{txn}_i$, W) : {cmt, abrt}

```
 1: if $T_{\max} > T_s(\textit{txn}_i)$ then
 2:     return abort;
 3: end if
 4: for each row $r \in W$ do
 5:     if $\textit{lastCommit}(r) > T_s(\textit{txn}_i)$ then
 6:         return abort;
 7:     end if
 8: end for
                                        ▷ Commit txn_i
 9: $T_c(\textit{txn}_i) \leftarrow$ TimestampOracle.next();
10: for each row $r \in W$ do
11:     $\textit{lastCommit}(r) \leftarrow T_c(\textit{txn}_i)$;
12: end for
13: return commit
```

Algorithm 2 also requires the commit timestamps to decide if a version written by a transaction is in the read snapshot of the running transaction. Omid, therefore, needs to modify Algorithm 2 to take into account the missed commit timestamps. We benefit from the observation that the commit timestamp of old, non-overlapping transactions could be forgotten as long as we could distinguish between transaction commit statuses, i.e., *committed*, *aborted*, and *in progress*. Assuming that most transactions commit, we use *committed* as the default status: a transaction is committed unless it is aborted or in progress. Omid consequently maintains the list of aborted and in-progress transactions in two *aborted* and *uncommitted* lists, accordingly. After an abort, the SO adds the transaction id to the *aborted* list. A transaction id is recorded in the *uncommitted* list after its start timestamp is assigned, and is removed from the list after it commits or aborts.

Notice that the problem of truncated T_c list is addressed by maintaining two new lists: *aborted* and *uncommitted*. These two lists, nevertheless, could grow indefinitely and fill up the memory space: e.g., transactions running by faulty clients could remain in the *uncommitted* list. Therefore, we need further mechanisms that truncate these two lists, without however compromising safety. Below, we provide two techniques for truncating each of the lists.

Once $T_{\max}$ advances due to eviction of data, we check for any transaction *txn*$_i$ in the *uncommitted* list for which $T_{\max} > T_s(\textit{txn}_i)$, and add it to the *aborted* list. In other words, we abort transactions that do not commit in a timely manner, *i.e.*, before $T_{\max}$ advances their start timestamp. This keeps the size of the *uncommitted* list limited to the number of recent in-progress transactions.

To truncate the *aborted* list, after the written versions of an aborted transaction are physically removed from the data store, we remove the aborted transaction from the *aborted* list. This could be done passively, after the old versions are garbage collected by the data store. Using the optional cleanup phase, explained in § II, the clients could also actively delete the written versions of the aborted transactions and notify the SO by a *cleaned-up* message. SO then removes the transaction id from the *aborted* list. To ensure that the truncation of the *aborted* list does not interfere with the current in-progress transactions, the SO defers the removal from the *aborted* list until the already started transactions terminate.

Algorithm 4 inSnapshot(txn_f, txn_r) : {true, false}

```
1: if $T_c(txn_f) \neq$ null then
2:     return $T_c(txn_f) < T_s(txn_r)$;
3: end if
4: if $T_{\max} < T_s(txn_f)$ or $aborted(txn_f)$ then
5:     return false;
6: end if
7: return true;
```

Algorithm 4 shows the procedure to verify if a transaction txn_f has been committed before a given timestamp. If $T_s(txn_f)$ is larger than $T_{\max}$, then the commit time could not be evicted and its absence in memory indicates that txn_f is not committed (Line 4). Plainly, if txn_f is in the aborted list, it is not committed either. If neither of the above conditions applies, txn_f is an old, committed transaction and we thus return true.

RPC Overhead One major load on the SO is the RPC (Remote Procedure Call) cost related to receiving and sending messages. The RPC cost comprises processing TCP/IP headers, as well as application-specific headers. To alleviate the RPC cost, we should reduce the number of sent/received messages per transaction. For each transaction, the SO has to send/receive the following main messages: Timestamp Request (TsReq), Timestamp Response (TsRes), inSnapshot Query, inSnapshot Response, Commit Request, Commit Response, and Abort Cleaned-up. Among these, *inSnapshot* are the most frequent messages, as they are at least as many as the size of the transaction read set. An approach that offloads processing onto some other nodes potentially reduces the RPC load on the SO.

Following the scheme of Percolator [27], the transaction metadata of the SO could be replicated on the data servers via writing the metadata into the modified rows. Fig. 4 depicts the architecture of such an approach. The problem with this approach is the non-negligible overhead induced to the data servers because of the second write of commit timestamps. For example, with Percolator this overhead resulted in a multi-second latency on transaction processing. Alternatively, Omid replicates the transaction metadata on the clients. Fig. 5 depicts the architecture of Omid. Having the SO state replicated at clients, each client could locally run Algorithm 2 to process *inSnapshot* queries without causing any RPC cost on the SO.

To lightly replicate the metadata on clients, Omid benefits from the following key observations:

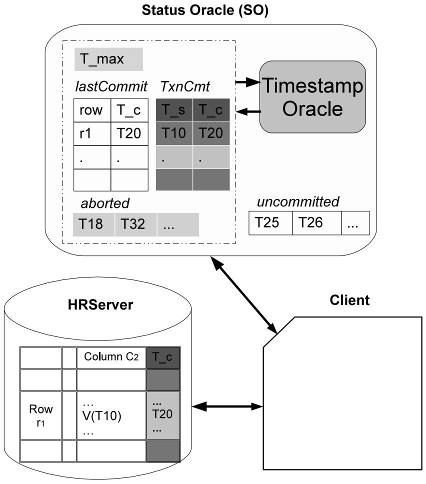


Fig. 4: Replication of txn metadata on data servers.

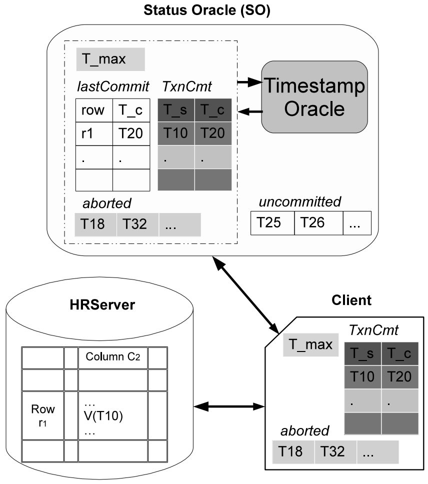


Fig. 5: Omid: replicating txn metadata on clients.

1) Processing the *inSnapshot* query requires the transaction metadata of the SO up until the transaction starting point. Therefore, the commits performed after the transaction start timestamp assignment can be safely ignored.
2) Since the clients write directly into HBase and the actual data does not go through the SO, it sends/receives mostly small messages. The SO is, therefore, a CPU-bound service and the network interface (NIC) bandwidth of the SO is greatly under-utilized.
3) Since the TsRes is very short, it easily fits into a single packet with enough extra space to include extra data. Piggybacking some data on the message, therefore, comes with almost no cost. With a proper compression algorithm, the metadata of hundreds of transactions could fit into a packet.
4) Suppose tps is the total throughput of the system and N is the number of clients. If a client runs transactions with the average rate of tps/N, between two consecutive TsReq messages coming from the same client the SO on average has processed N commit requests. Piggybacking the new transaction metadata on TsRes messages induces little overhead, given that N is no more than a thousand (§ V evaluates the scalability under different rates of running transactions at clients.)

Consequently, Omid could replicate the transaction meta-

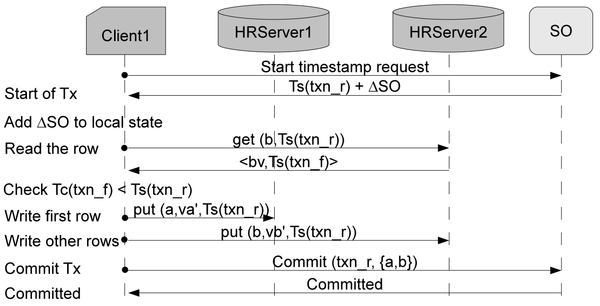


Fig. 6: Sequence diagram of a successful commit in Omid. The transaction reads key ”b” and writes values for keys ”a” and ”b” into two data servers.

data of the SO to transaction clients for almost no cost. The TsRes T_k^i to Client C^i will be accompanied with $\Delta SO_k^i = SO_k^i - SO_{k-1}^i$, where SO_k^i is the state of transaction metadata at the SO when the TsRes T_k^i is processed. Ideally, this metadata after compression fits into the main packet of TsRes message to avoid the cost of framing additional packets. For the clients that run with a slower rate of $\alpha.tps/N$ ($0 < \alpha < 1$), the size of the transaction metadata increases to $\alpha^{-1}.N$, which could have a negative impact on the scalability of the SO with N, the number of clients. § V evaluates the scalability with clients running with different speeds and shows that in practice the SO performance is not much affected by the distribution of the load coming from clients.

More precisely, the *transaction metadata* (i.e., SO) consists of (i) the T_c list (i.e., mapping between start and commit timestamps), (ii) *aborted* list, and (iii) T_{max}. Note that *lastCommit* list is not replicated to clients. (*lastCommit* constitutes the most of the memory space in SO since it has an entry per each modified row by transactions whereas T_c that has an entry per transaction.) Consequently, the amount of required memory at the client to keep the replicated data is small.

Fig. 6 shows the steps of a successful commit in Omid. If the read value from the data server is not committed, the client should read the next version from the data server. To avoid additional reads, the client could read last nv ($nv \geq 1$) versions in each read request sent to the data server. Since the data in the data server is also sorted by version, reading more consecutive versions does not add much retrieving overhead to the data server. Moreover, by choosing a small nv (e.g., three) the read versions will all fit into a packet and does not have tangible impact on the response message framing cost.

IV. Implementation

Here, we present some important implementation details.

SO Data Reliability After a failure in the SO, the in-memory data will be lost. Another instance of the SO should recover the essential data to continue servicing the transactions. Since *lastCommit* is used solely for detecting conflicts between concurrent transactions, it does not have to be restored after a failure, and the SO could simply abort all the uncommitted transactions that started before the failure. the SO, however, must be able to recreate T_{max}, the T_c list, and the *aborted* list. The *uncommitted* list is recreated from the T_c and *aborted* lists: whatever that is not committed nor aborted, is in the *uncommitted* list.

One widely used solution to reliability is *journaling*, which requires persisting the changes into a write-ahead log (WAL). The WAL is also ideally replicated across multiple remote storage devices to prevent data loss after a storage failure. We use Bookkeeper [3] for this purpose. Since Omid requires frequent writes into the WAL, multiple writes could be batched with no perceptible increase in processing time. The write of the batch to BookKeeper is triggered either by batch size, after 1 KB of data is accumulated, or by time, after 5 ms since the last trigger.

Recovery To recover from a failure, the SO has to recreate the memory state by reading data from the WAL. In addition to the last assigned timestamp and $T_{\max}$, the main state that has to be recovered consists of (i) T_c list down until $T_{\max}$, and (ii) *aborted* list. Because (i) the assigned timestamps are in monotonically increasing order and (ii) obtaining a commit timestamp and writing it to the WAL is performed atomically in our implementation, the commit timestamps in the WAL are also in ascending order. We, therefore, optimize the recovery of T_c list by reading the WAL from the end until we read a commit timestamp $T_c < T_{\max}$.

The *aborted* list has to be recovered for the transactions that have not cleaned up after aborting. The delayed cleanup occurs rarely, only in the case of faulty clients. To shorten recovery time, we perform a light snapshotting only for aborted transactions of faulty clients. The SO periodically checkpoints the *aborted* list and if an aborted item is in the list during two consecutive checkpoints, then it is included in the checkpoint. The recovery procedure reads the aborted items until it reads two checkpoints from the WAL. Taking checkpoints is triggered after an advance of $T_{\max}$ by the size of T_c list, which roughly corresponds to the amount of transaction metadata that has to be recovered from the WAL. Our recovery techniques allow a fast recovery without, however, incurring the problems typical of a traditional snapshotting: non-negligible overhead and complexity. Overall, the amount that needs to be recovered from the WAL is much smaller than the SO memory footprint. This is because *lastCommit*, which is not persisted in the WAL, is an order of magnitude larger than the T_c list. To further reduce the fail-over time, a hot backup could continuously read from the WAL.

Client Replica of the SO To answer the *inSnapshot* query locally, the client needs to maintain a mapping between the start timestamps and commit timestamps as well as the list of aborted transactions. The client, therefore, maintains a hash map between the transaction start timestamp and commit timestamp. The hash map is updated based on the new commit information piggybacked on each TsRes message, ΔSO_k^i. The hash map garbage collects its data based on the updated $T_{\max}$ that it receives alongside ΔSO_k^i. In addition to the recent aborted transactions, the piggybacked data also includes the list of recently cleaned-up aborts to maintain the aborted list at the client side.

Client Startup Since the transaction metadata is replicated on clients, the client can answer *inSnapshot* queries locally, without needing to contact the SO. After a new client C^i establishes its connection to the SO, it receives a startup timestamp T_{init}^i that indicates the part of transaction metadata that is not replicated to the client. If a read value from the data store is tagged with timestamp T_s, where $T_{\max} < T_s < T_{init}^i$, the client has to inquire the SO to find the answer to $inSnapshot$ query. This is the only place that $inSnapshot$ causes an RPC cost to the SO. Note that as the system progresses, $T_{\max}$ advances T^i_{init}, which indicates that the recent commit result in the SO is already replicated to the client C^i and the client no longer needs to contact the SO for $inSnapshot$ queries.

Silent Clients If a client C^i is silent for a very long time (average 30 s in our experiments), the SO pushes the ΔSO^i to the client connection anyway. This prevents the ΔSO^i from growing too large. If the client connection does not acknowledge the receipt of the metadata, the SO breaks the connection after a certain threshold (5 s in our experiments).

Computing ΔSO^i_k When computing ΔSO^i_k, it is desirable to avoid impairing the SO performance while processing commits. After commit of txn_w, the SO *appends* the $zip(T_s(txn_w), T_c(txn_w))$ into *commitinfo*, a byte array shared between all the connections, where zip is a compression function that operates on the start and commit timestamps. Per each open connection, the SO maintains a pointer to the last byte of *commitinfo* that is sent to the client with the latest TsRes message. The SO piggybacks the newly added data of *commitinfo* into the next TsRes message and updates the corresponding pointer to the last byte sent over the connection, accordingly. The benefit of this approach is that the piggybacked data is computed only once in *commitinfo*, and the send operation causes only the cost of a raw memory copy on the SO. With more connections, nevertheless, the size of piggybacked data increases and the cost of memory copy operation starts to be nontrivial. The experimental results in § V show that Omid scales up to 1000 clients with an acceptable performance.

HBase Garbage Collection Since HBase maintains multiple versions per value, it requires a mechanism to garbage collect the old versions. We changed the implementation of the garbage collection mechanism to take into account the values that are read by in-progress transactions. The SO maintains a T_{min} variable, which is the minimum start timestamp of uncommitted transactions. Upon garbage collection, the region server contacts the SO to retrieve the T_{min} variable as well as the aborted list. Using these two, the garbage collection mechanism ensures to keep at least a value that is not aborted and has a start timestamp less than T_{min}.

V. Evaluation

The experiments aim to answer the following questions: (i) Is Omid a bottleneck for transactional traffic in large distributed data stores? (ii) What is the overhead of replicating transaction metadata on clients? (iii) What is the overhead of Omid on HBase? Since clients access the SO and HBase separately, scalability of the SO is independent of that of HBase. In other words, the load that transactions running under Omid put on HBase is independent of the load they put on the SO. We thus evaluate the scalability of SO in isolation from HBase. To evaluate the bare bones of the design, we did not batch messages sent to the SO and data server. For an actual deployment, of course, batching could offer a trade-off between throughput and latency. We used 49 machines with 2.13 GHz Dual-Core Intel(R) Xeon(R) processor, 2 MB cache, and 4 GB memory: 1 for the ZooKeeper coordination service [23], 2 for BookKeeper, 1 for SO, 25 for data servers (on which we install both HBase and HDFS), and the rest for hosting the clients (up to 1024). Each client process runs one transaction at a time.

Workloads An application on top of HBase generates a workload composed of both read and write operations. The ratio of reads and writes varies between the applications. To show that the overhead of Omid is negligible for any given workload (independent of the ratio of read and writes), we verify that the overhead of Omid is negligible for both read and write workloads separately. To generate workloads, we use the Yahoo! Cloud Serving Benchmark, YCSB [16], which is a framework for benchmarking large key-value stores. The vanilla implementation does not support transactions and hence operates on single rows. We modified YCSB to add support for transactions, which touch multiple rows. We defined three types of transactions: (i) *ReadTxn*: where all operations only read, (ii) *WriteTxn*: where all operations only write, and (iii) *ComplexTxn*: consists of 50% read and 50% write operations. Each transaction operates on n rows, where n is a uniform random number with average 8. Based on these types of transactions, we define four workloads: (i) *Read-only*: all *ReadTxn*, (ii) *Write-only*: all *WriteTxn*, (iii) *Complex-only*: all *ComplexTxn*, and (iv) *Mixed*: 50% *ReadTxn* and 50% *ComplexTxn*.

Micro-benchmarks In all the experiments in this section, the clients communicate with a remote SO. In some, the clients also interact with HBase. To enable the readers to interpret the results reported here, we first break down the latency of different operations involved in a transaction: (i) start timestamp request, (ii) read from HBase, (iii) write to HBase, and (iv) commit request. Each read and write into HBase takes 38.8 ms and 1.13 ms on average, respectively. The writes are in general less expensive since they usually include only writing into memory and appending into a WAL[7]. Random reads, on the other hand, might incur the cost of loading an entire block from HDFS, and thus have higher delays. Note that for the experiments that evaluate the SO in isolation from HBase, the read and write latencies are simulated by a delay at the client side.

The commit latency is measured from the moment that the commit request is sent to the SO until the moment its response is received. The average commit latency is 5.1 ms, which is mostly contributed by the delay of sending the recent modifications to BookKeeper. The average latency of start timestamp request is 0.17 ms. Although the assigned start timestamps must also be persisted, the SO reserves thousands of timestamps per each write into the WAL, and hence on average servicing timestamps do not incur a persistence cost.

Replication Cost A major overhead of Omid is the replication of the transaction metadata onto the client nodes, to which we refer as CrSO (Client-Replicated Status Oracle). To assess scalability with the number of client nodes, we exponentially increase the number of clients from 1 to 2^{10} and plot the average latency vs. the average throughput in Fig. 7. The clients simulate transactions by sleeping for 50 ms before sending the commit request. The read-only transactions do not cause to

[7] Note that the WAL the HBase uses trades reliability for higher performance.

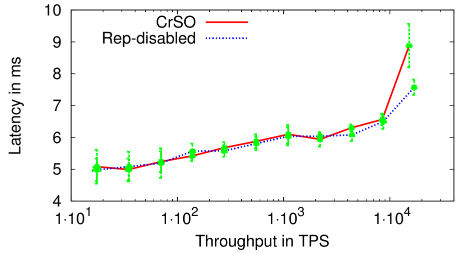


Fig. 7: Replication overhead.

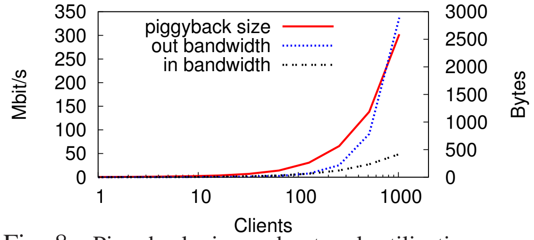


Fig. 8: Piggyback size and network utilization.

the SO the cost of checking for conflicts as well as the cost of persisting data into the WAL. Moreover, modifying more rows per transaction implies a higher cost of checking for conflicts at the SO. To evaluate the Omid performance under high load, we use a *write-only* workload where rows are randomly selected out of 20M rows. When exponentially increasing the number of client nodes, the latency only linearly increases up to 8.9 ms. We also conduct an experiment with the replication component disabled, which is labeled *Rep-disabled* in Fig. 7. Note that the resulting system (with replication disabled) is *incomplete* since the clients do not have access to metadata to determine which read version is in the read snapshot of the running transaction. Nevertheless, this experiment is insightful as it assesses the efficiency of the replication component; i.e., how much overhead does the replication component puts on the SO? The negligible difference indicates the efficiency of the metadata replication thanks to the light replication design presented in § III. We will explain the 1 ms jump in the latency with 1024 clients, later when we analyze the piggyback size.

Replication on clients also consumes the NIC bandwidth of the SO. This overhead is depicted in Fig. 8. The inbound bandwidth usage slowly increases with the number of clients, as the SO receives more requests from clients. Even with 1024 clients, the consumed bandwidth is 50 Mbps, which was expected since the SO is a CPU-bound service. The usage of the outbound bandwidth, however, is higher since it is also used to replicate the metadata on clients. Nevertheless, the NIC is still underutilized even with 1024 clients (337 Mbps). The average memory footprint per client varies from 11.3 MB to 16.8 MB, indicating that the replication of transaction metadata on clients does not require much memory space at the client.

The replication of the transaction metadata is done through piggybacking recent metadata on the TsRes message. Fig. 8 depicts the piggybacked payload size as we add clients. As we expected from the analysis in § III, the size is proportional to the number of clients. The size increases up to 1186 bytes and 2591 bytes with 512 and 1024 clients, respectively. This means that with a thousand clients and more, the replication incurs the cost of sending extra packets for each transaction. This cost contributes to the 1 ms higher latency for replication onto 1024 clients, depicted in Fig. 7.

Txn Time	tps	lat	piggyback (stdv)	flush/s (stdv)
50 ms	15.0	8.9	2591 (39)	0.0 (00)
exp	15.1	8.9	2585 (43)	0.0 (00)
10% slow	15.4	8.4	2548 (45)	0.0 (00)
10% glacial	15.4	8.4	2511 (862)	2.4 (25)

TABLE I: Slow Clients.

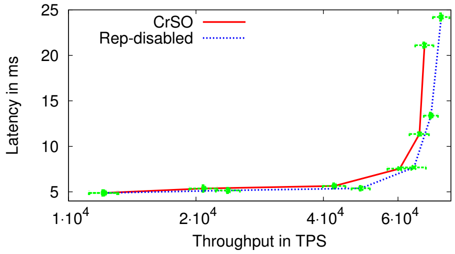


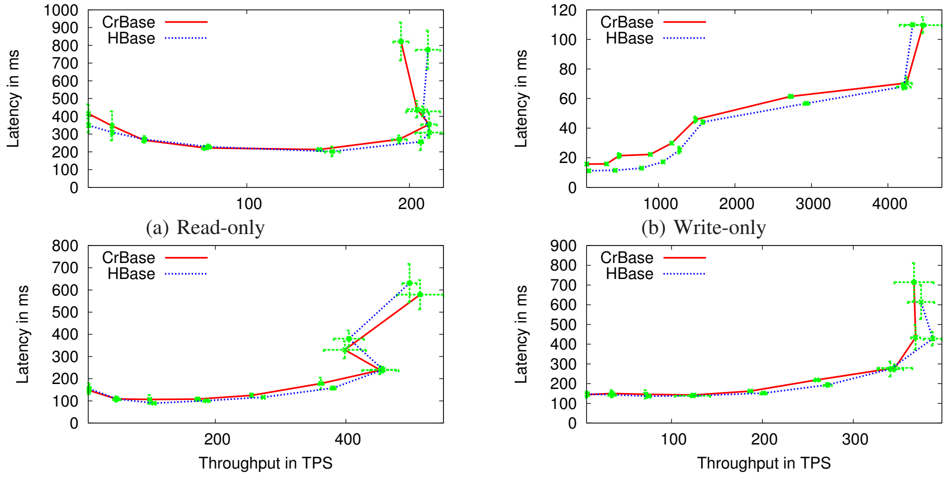
Fig. 9: Scalability.

Slow Clients The SO piggybacks on the TsRes message the recent transaction metadata since the last timestamp request from the client. The larger the piggyback, the higher is the overhead on the SO. The overhead would be at its lowest level if clients run transactions with a similar rate. When a client operates with a lower rate, the piggyback is larger, which implies a higher overhead on the SO. As Fig. 7 showed, the replication technique of Omid is stressed when lots of clients connect to the SO. Here, we report on experiments with 1024 clients with different execution time for transactions. Each transaction has a write set of average size 8.

The first row of Table I depicts the results when the transaction execution time of all clients is set to 50 ms. (This time excludes the latency of obtaining the start timestamp and commit.) For the 2nd row the execution time is an exponential random variable with average of 50 ms, modeling a Poisson process on requests arriving at the SO. In the next two rows, 10% of clients are faulty (10^2 and 10^3 times slower, respectively).

The *tps* and *lat* columns show the average throughput and latency of non-faulty clients, respectively. No perceptible change into these two parameters indicates that the overhead on the SO is not much affected by the distribution of requests coming from clients. The average size of piggybacks also does not change much across the setups. The variance, nevertheless, increases for the setup with extremely slow clients. The reason is that the metadata sent to the slow clients are larger than the average size. *flush* in the last column shows the number of times that the SO pushes the transaction metadata down to the connection since it has not received a timestamp request for a long time. This only occurs in the setup with 10% extremely slow clients (average run time of 50 s) with the average rate of 2.4 flushes per second. The variance is very high since this mechanism is triggered periodically (every 30 s on average), when the SO garbage collects the old data in *commitinfo*.

Omid Scalability To assess scalability of Omid with the number of transactions, we repeat the same experiment with the difference that each client allows for 100 outstanding transactions with the execution time of zero, which means that the clients keep the pipe on the SO full. As Fig. 9 depicts, by increasing the load on the SO, the throughput increases up to 71K TPS with average latency of 13.4 ms. After this point, increasing the load increases the latency (mostly due to the buffering delay at the SO) with only marginal throughput



(a) Read-only

(b) Write-only

(c) Complex-only

(d) Mixed

Fig. 11: Overhead on HBase.

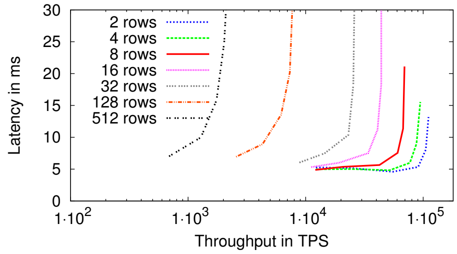


Fig. 10: Scalability with transaction size.

improvement (76K TPS). Similarly to Fig. 7, the difference with the case when the replication is disabled indicates the low overhead of our light replication technique. Recall that our replication technique incurs the cost of a raw memory copy from *commitinfo* to the outgoing TsRes packets.

Scalability with Transaction Size The larger the write set of the transaction, the higher is the overhead of checking for write-write conflicts on the SO. To assess scalability of Omid with the number of transactions, we repeat the previous experiment varying the average number of modified rows per transactions from 2 to 4, 8, 16, and 32. As Fig. 10 depicts, by increasing the load on the SO, the throughput with transactions of size 2 increases up to 124K TPS with average latency of 12.3 ms. Increasing the average transaction size lowers the capacity of the SO and makes it saturate with lower throughput. The SO saturates with 110K, 94K, 67K, 41K TPS with transactions of average size of 4, 8, 16, and 32, respectively.

Omid is designed for OLTP workloads, which typically consist of small transactions. From a research point of view, however, it is interesting to evaluate the SO with large transactions. To this aim, we repeat the experiments with transactions of average write size 128 and 512. The SO saturates at 11K and 3.2K, respectively. Note that although TPS drops with longer transactions, the overall number of write operations per second is still very high. For example, for transactions of write size 512, the data store has to service 1.6M writes/s.

Recovery Delay Fig. 12 shows the drop in the system throughput during recovery from a failure in the SO. After approximately 100 minutes, the SO servicing 1024 clients fails, another instance of the SO recovers the state from the WAL, the clients reset their read-only replica of transaction metadata, and connect to the new SO. The recovery takes around 13 seconds, during which the throughput drops to 0. To further reduce the fail-over time, a hot backup could continuously read the state from the WAL. We observed a similar latency independent of the time SO is running before the failure. This is because the recovery delay is independent of the size of the WAL, thanks to our recovery techniques explained in § IV.

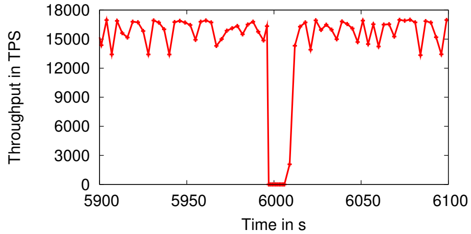


Fig. 12: Recovery from the SO failure.

Overhead on HBase Here, we evaluate CrBase, our prototype that integrates HBase with Omid, to measure the overhead of transactional support on HBase. HBase is initially loaded with a table of size 100 GB comprising 100M rows. This table size ensures that the data does not fit into the memory of data servers.

Fig. 11 depicts the performance when increasing the number of clients from 1 to 5, 10, 20, 40, 80, 160, 320, and 640. We repeat each experiment five times and report the average and variance. As expected, the performance of both systems under *read-only* workload is fairly similar (Fig. 11a), with the difference that CrBase has a slightly higher latency due to contacting the SO for start timestamps and commit requests. Fig. 11b depicts the results of the same experiment with *write-only* workload. CrBase exhibits the same performance as HBase since both perform the same number of writes per transaction. Since the overhead of Omid on HBase is negligible under both read and write workloads, we expect the same pattern under any workload, independent of ratio of reads to writes. This is validated in Fig. 11c and 11d for the *complex-only* and *mixed* workloads, respectively. HBase shows some anomalies under *read-only* workload with few clients and under *complex-only* workload with 320 clients. Note that the main goal of these experiments was to determine the transaction overhead on HBase rather than perform a thorough analysis of HBase. As the curves show, the anomalies observed appear with and without our transaction management, indicating that they originate from HBase.

VI. Related Work

CloudTPS [30] runs a two-phase commit (2PC) algorithm [22] between distributed Local Transaction Managers (LTM) to detect conflicts at the commit time. In the first phase, the coordinator LTM puts locks on other LTMs, and in the second phase, it commits the changes and removes the locks. Each LTM has to cache the modified data until the commit time. This approach has some obvious scalability problems: (i) the size of data that each LTM has to cache to prevent all the conflicts is large [30], (ii) the agreement algorithm does not scale with the number of LTMs, and (iii) the availability must be provided for each of the LTMs, which is very expensive considering the large size of the LTM state. CloudTPS, therefore, could only be used for Web applications when little data is modified by transactions [30]. Moreover, it requires the specification of the primary keys of all data items that are to be modified by the transaction at the time the transaction is submitted.

Percolator [27] takes a lock-based, distributed approach to implement SI on top of Bigtable. To commit transactions, the clients run O2PL [21], which is a variation of 2PC that defers the declaration of the write set (and locking it) until the commit time. Although using locks simplifies the write-write conflict detection, the locks held by a failed or slow transaction prevent others from making progress until the full recovery from the failure (Percolator reports up to several minutes delay caused by unreleased locks.). Moreover, maintaining the lock column as well as responding the queries about a transaction status coming from reading transactions puts extra load on data servers. To mitigate the impact of this extra load, Percolator [27] relies on batching of messages sent to data servers, contributing to the multi-second delay on transaction processing.

hbase-trx [4] is an abandoned attempt to extend HBase with transactional support. Similar to Percolator, hbase-trx runs a 2PC algorithm to detect write-write conflicts. In contrast to Percolator, hbase-trx generates a transaction id locally (by generating a random integer) rather than acquiring one from a global oracle. During the commit preparation phase, hbase-trx detects write-write conflicts and caches the write operations in a server-side state object. On commit, the data server (i.e., RegionServer) applies the write operations to its regions. Each data server considers the commit preparation and applies the commit in isolation. There is no global knowledge of the commit status of transactions.

In the case of a client failure after a commit preparation, the transaction will eventually be applied optimistically after a timeout, regardless of the correct status of the transaction. This could lead to inconsistency in the database. To resolve this issue, hbase-trx would require a global transaction status oracle similar to that presented in this paper. hbase-trx does not use the timestamp attribute of HBase fields; transaction ids are randomly generated integers. Consequently, hbase-trx is unable to offer snapshot isolation, as there is no fixed order in which transactions are written to the database.

To achieve scalability, MegaStore [10], ElasTras [18], and G-Store [19] rely on partitioning the data store, and provide ACID semantics within partitions. The partitions could be created statically, such as with MegaStore and ElasTras, or dynamically, such as in G-Store. However, ElasTras and G-Store have no notion of consistency across partitions and MegaStore [10] provides only limited consistency guarantees across them. ElasTras [18] partitions the data among some transaction managers (OTM) and each OTM is responsible for providing consistency for its assigned partition. There is no notion of global serializability. In G-Store [19], the partitions are created dynamically by a *Key Group* algorithm, which essentially labels the individual rows of the database with the group id.

MegaStore [10] uses a WAL to synchronize the writes within a partition. Each participant writes to the main database only after it successfully writes into the WAL. Paxos is run between the participants to resolve the contention between multiple writes into the WAL. Although transactions across multiple partitions are supported with a 2PC implementation, the applications are discouraged from using that feature because of performance issues.

Similar to Percolator, Deuteronomy [25] uses a lock-based approach to provide ACID. In contrast to Percolator where the locks are stored in the same data tables, Deuteronomy uses a centralized lock manager (TC). Furthermore, TC is the portal to the database and all the operations must go through it. This leads to a low throughput offered by TC [25]. On the contrary, our approach is lock-free and can scale up to 71K TPS. ecStore [29] also provides SI. To detect write-write conflicts it runs a 2PC algorithm among all participants, which has scalability problem for general workloads.

Similar to Percolator, Zhang and Sterck [31] use the HBase data servers to store transaction metadata. However, the metadata is stored on some separate tables. Even the timestamp oracle is a table that stores the latest timestamp. The benefit is that the system can run on bare-bone HBase. The disadvantage, however, is the low performance due to the large volume of accesses to the data servers to maintain the metadata. Our approach provides SI with a negligible overhead to data servers.

In our tool, Omid, each client maintains a read-only replica of the transaction metadata, such as the aborted transactions and the commit timestamps, where the clients regularly receive messages from the SO to update their replica. This partly resembles the many exiting techniques in which the client caches the data to reduce the load on the server. A direct comparison to such works is difficult since in Omid the flow of data and transaction metadata is separated, and the clients maintain a read-only copy of only the metadata in contrast to the *actual* data. Moreover, the new received metadata does not invalidate the current state of the replica (e.g., a previously received commit timestamp would not change) whereas a cached object could be invalid due to the recent updates in the server. There is a large body of work for client cache invalidation (See [21] for a taxonomy on this topic.). In some related work, after an object modification, the server sends object invalidation messages to the clients that have cached the object. Adya et. al. further suggest delaying the transmission of such messages to have a chance of piggybacking them on other types of messages that probably flow between the server and the client [7], [6]. Omid also employs piggybacking but for the recent transaction metadata. A main difference is that Omid does not send such messages after each change in the metadata. Instead, it benefits from the semantics of SI that the client does not require the recent changes until it asks for a start timestamp. The SO hence piggybacks the recent metadata on the TsRes message. Furthermore, Omid guarantees that the client replica of the metadata is sufficient to service the reads of the transaction. This is in contrast to many of the cache invalidation approaches that the client cache could be stale and further mechanisms are required to later verify the reads performed by a transaction [7], [6].

Some systems are designed specifically to efficiently support transactions that span multiple data centers. Google Spanner [17] is a recent example that makes use of distributed locks accompanied by a novel technique named TrueTime that allows using local clocks for ordering the transactions without needing an expensive clock synchronization algorithm. TrueTime nevertheless operates based on an assumed bound on the clock drift rates. Although Spanner makes use of *GPS* and *atomic clocks* to reduce the assumed bound, a clock drift rate beyond it could violate the data consistency. As explained in § III the scope of Omid is the many existing applications that operate within a data center. To sequence the transactions, Omid uses a centralized timestamp oracle built in the SO. Although this simple design limits the rate of write transactions under Omid, it offers the advantage that it operates on off-the-shelf hardware with no assumption about the clock drift rate.

VII. Concluding Remarks

We have presented a client-replicated implementation of lock-free transactions in large distributed data stores. The approach does not require modifying the data store and hence can be used on top of any existing multi-version data stores. We showed by design as well as empricially that the overhead of Omid for both write and read-only transactions is negligible, and therefore is expected to be negligible for any given workload. Being scalable to over 100K write TPS, we do not expect Omid to be a bottleneck for many existing large data stores. These promising results provide evidence that lock-free transactional support could be brought to large data stores without hurting the performance or the scalability of the system. The open source release of our prototype on top of HBase is publicly available [5].

Acknowledgement

We thank Rui Oliveira for the very useful comments on an earlier version of Omid, which resulted in developing the technique for replicating the SO transaction metadata to the clients. This work has been partially supported by the Cumulo Nimbo project (ICT-257993), funded by the European Community.

References

[1] http://hbase.apache.org.

[2] http://cassandra.apache.org.

[3] http://zookeeper.apache.org/bookkeeper.

[4] https://github.com/hbase-trx.

[5] https://github.com/yahoo/omid.

[6] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *SIGMOD*, 1995.

[7] A. Adya and B. Liskov. Lazy consistency using loosely synchronized clocks. In *PODC*, 1997.

[8] D. Agrawal, A. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1):45–59, 1987.

[9] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. In *SIGMOD*, 1989.

[10] J. Baker, C. Bondç, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011.

[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 1995.

[12] M. Bornea, O. Hodson, S. Elnikety, and A. Fekete. One-copy serializability with snapshot isolation under the hood. In *ICDE*, 2011.

[13] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *TOCS*, 2008.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *TOCS*, 2008.

[15] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *VLDB*, 2008.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally-distributed database. In *SDI*, 2012.

[18] S. Das, D. Agrawal, and A. El Abbadi. Elastras: an elastic transactional data store in the cloud. In *HotCloud'09*, 2009.

[19] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC'10*, 2010.

[20] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *SRDS 2005*, pages 73–84, 2005.

[21] M. Franklin, M. Carey, and M. Livny. Transactional client-server cache consistency: alternatives and performance. *TODS*, 1997.

[22] J. Gray. Notes on data base operating systems. *Operating Systems*, 1978.

[23] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[24] H. Kung and J. Robinson. On optimistic methods for concurrency control. *TODS*, 1981.

[25] J. J. Levandoski, D. Lome, M. F. Mokbel, and K. K. Zhao. Deuteronomy: Transaction Support for Cloud Data. In *CIDR*, 2011.

[26] Y. Lin, K. Bettina, R. Jiménez-Peris, M. Patiño Martínez, and J. E. Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *TODS*, 2009.

[27] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[28] Y. Sovran, R. Power, M. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[29] H. Vo, C. Chen, and B. Ooi. Towards elastic transactional cloud storage with range query support. *PVLDB*, 2010.

[30] Z. Wei, G. Pierre, and C.-H. Chi. CloudTPS: Scalable transactions for Web applications in the cloud. *IEEE Trans. on Services Comp.*, 2011.

[31] C. Zhang and H. De Sterck. Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase. In *Grid*, 2010.