Omid, Reloaded: Scalable and Highly-Available Transaction Processing

Ohad Shacham
Yahoo Research

Francisco Perez-Sorrosal
Yahoo

Edward Bortnikov
Yahoo Research

Eshcar Hillel
Yahoo Research

Idit Keidar
Technion and Yahoo Research

Ivan Kelly
Midokura

Matthieu Morel
Skyscanner

Sameer Paranjpye
Arimo

Abstract

We present Omid – a transaction processing service that powers web-scale production systems at Yahoo. Omid provides ACID transaction semantics on top of traditional key-value storage; its implementation over Apache HBase is open sourced as part of Apache Incubator. Omid can serve hundreds of thousands of transactions per second on standard mid-range hardware, while incurring minimal impact on the speed of data access in the underlying key-value store. Additionally, as expected from always-on production services, Omid is highly available.

Omid's database-neutral philosophy and real-world deployment considerations have led to unique design choices, including managing (and in particular, scaling) compute and metadata storage of transaction management independently, and a novel high availability algorithm that runs synchronization-free most of the time.

1 Introduction

In recent years, there is an increased focus on supporting large-scale distributed transaction processing; examples include [6, 7, 11, 17, 18, 20, 28]. Transaction systems have many industrial applications, and the need for them is on the rise in the big data world. One prominent use case is Internet-scale data processing pipelines, for example, real-time indexing for web search [31]. Such systems process information in a streamed fashion, and use shared storage in order to facilitate communication between processing stages. Quite often, the different stages process data items in parallel, and their execution is subject to data races. Overcoming such race conditions at the application level is notoriously complex; the system design is greatly simplified by using the abstraction of transactions with well-defined *atomicity*, *consistency*, *isolation*, and *durability* (ACID) semantics [27].

We present Omid, an ACID transaction processing system for key-value stores. Omid has replaced an initial prototype bearing the same name, to which we refer here as Omid1 [25], as Yahoo's transaction processing engine; it has been entirely re-designed for scale and reliability, thereby bearing little resemblance with the origin (as discussed in Section 3 below). Omid's open source version recently became an Apache Incubator project[1].

Internally, Omid powers Sieve[2], Yahoo's web-scale content management platform for search and personalization products. Sieve employs thousands of tasks to digest billions of events per day from a variety of feeds and push them into a real-time index in a matter of seconds. In this use case, tasks need to execute as ACID transactions at a high throughput [31].

The system design has been driven by several important business and deployment considerations. First, guided by the principle of separation of concerns, Omid was designed to leverage battle-tested key-value store technology and support transactions over data stored therein, similar to other industrial efforts [6, 31, 17]. While Omid's design is compatible with multiple NoSQL key-value stores, the current implementation works with Apache HBase [1].

A second consideration was simplicity, in order to make the service easy to deploy, support, maintain, and monitor in production. This has led to a design based on a centralized *transaction manager* (*TM*)[3]. While its clients and data storage nodes are widely-distributed and fault-prone, Omid's centralized TM provides a single source of truth regarding the transaction history, and facilitates conflict resolution among updating transactions (read-only transactions never cause aborts).

Within these constraints, it then became necessary to find novel ways to make the service scalable for throughput-oriented workloads, and to ensure its con-

[1] http://omid.incubator.apache.org

[2] http://yahoohadoop.tumblr.com/post/129089878751

[3] The TM is referred to as Transaction Status Oracle (TSO) in the open source code and documentation.

tinued availability following failures of clients, storage nodes, and the TM. Omid's main contribution is in providing these features:

Scalability Omid runs hundreds of thousands of transactions per second over multi-petabyte shared storage. As in other industrial systems [31, 25, 6], scalability is improved by providing *snapshot isolation* (SI) rather than serializability [27] and separating data management from control. Additionally, Omid employs a unique combination of design choices in the control plane: (i) synchronization-free transaction processing by a single TM, (ii) scale-up of the TM's in-memory conflict detection (deciding which transactions may commit) on multi-core hardware, and (iii) scale-out (and scale-up) of metadata update (HBase).

High availability The data tier is available by virtue of HBase's reliability, and the TM is implemented as a primary-backup process pair with shared access to critical metadata. Our solution is unique in tolerating a potential overlap period when two processes act as primaries, and at the same time avoiding costly synchronization (consensus), as long as a single primary is active. Note that, being generic, the data tier is not aware of the choice of primary and hence serves operations of both TMs in case of such overlap.

We discuss Omid's design considerations in Section 2 and related transaction processing systems in Section 3. We detail the system guarantees in Section 4. Section 5 describes Omid's transaction protocol, and Section 6 discusses high-availability. An empirical evaluation is given in Section 7. We conclude, in Section 8, by discussing lessons learned from Omid's production deployment and our interaction with the open source community, as well as future developments these lessons point to.

2 Design Principles and Architecture

Omid was incepted with the goal of adding transactional access on top of HBase, though it can work with any strongly consistent key-value store that provides multi-versioning with version manipulation and atomic *putIfAbsent* insertions as we now describe.

The underlying data store offers *persistence* (using a write-ahead-log), *scalability* (via sharding), and *high availability* (via replication) of the data plane, relieving Omid to manage only the transaction control plane. Omid further relies on the underlying data store for fault-tolerant and persistent storage of transaction-related metadata. This metadata includes a dedicated table that holds a single record per committing transaction, and in addition, per-row metadata for items accessed transactionally. The Omid architecture is illustrated in Figure 1.

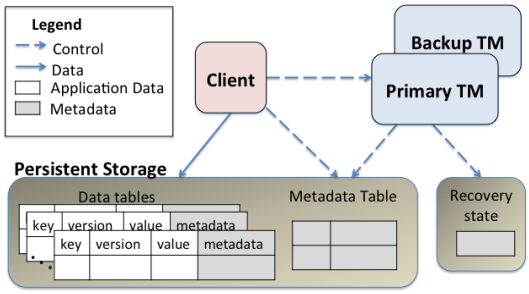


Figure 1: **Omid architecture**. Clients manipulate data that resides in data tables in the underlying multi-versioned persistent data store (for example, HBase) and use the TM in the control path for conflict detection. Only the primary TM is active, and the backup is in hot standby mode. The TM maintains persistent metadata in the data store as well as separately managed recovery state (for example, using Zookeeper).

Omid leverages *multi-versioning* in the underlying key-value store in order to allow transactions to read consistent snapshots of changing data as needed for snapshot isolation. The store's API allows users to manipulate versions explicitly. It supports atomic *put(key, val, ver)* and *putIfAbsent(key, val, ver)* operations for updating or inserting a new item with a specific version, and an atomic *get(key, ver)* operation for retrieving the item's value with highest version not exceeding *ver*. Specifically, when the item associated with an existing key is overwritten, the new version (holding the key, its new value, and a new version number) is created, while the previous version persists. An old version might be required as long as there is some active transaction that had begun before the transaction that overwrote this version has committed. Though this may take a while, overwritten versions eventually become obsolete. A cleaning process, (in HBase, implemented as a coprocessor [2]), frees up the disk space taken up by obsolete versions.

The transaction control plane is implemented by a centralized transaction manager. The TM has three roles: (i) *version (timestamp) allocation* for each transaction; (ii) *conflict detection* in order to determine which transactions may commit; and (iii) *persistent logging of the commits*. The TM provides high availability via a primary-backup approach— if the primary TM becomes unresponsive, then the backup becomes the new primary and takes over. This design offers durability and high availability; it further facilitates scalability of storage and compute resources separately – metadata storage ac-

cess scales out on the underlying distributed data store, whereas conflict management is done entirely in RAM, and scales up on a shared-memory multi-core server.

Conflict detection is lightweight and can be performed at a high throughput – at least an order of magnitude faster than the transaction processing rates the network can sustain. For example, with eight threads, it can handle five million transactions per second. To reduce the communication cost and RAM footprint of the TM, we use approximate conflict detection, which is based on hashes rather than full keys, and also represents some information in aggregate form. Although this may lead to false aborts, the abort rates witnessed both in production and in microbenchmarks are negligible.

Our high availability solution tolerates "false" failovers, where a new primary replaces one that is simply slow, (for example, due to a garbage collection stall), leading to a period with two active primaries. Synchronization between the two is based on shared persistent metadata storage, and induces overhead only in rare cases when more than one TM acts as primary. Omid uses time-based leases in order to minimize potential overlap among primaries. The implementation employs Apache Zookeeper [4] for lease management and synchronization between primary and backup.

3 Related Work

Distributed transaction processing has been the focus of much interest in recent years. Most academic-oriented papers [7, 8, 11, 18, 20, 32, 36] build full-stack solutions, which include transaction processing as well as a data tier. Some new protocols exploit advanced hardware trends like RDMA and HTM [19, 20, 33]. Generally speaking, these solutions do not attempt to maintain separation of concerns between different layers of the software stack, neither in terms of backward compatibility nor in terms of development efforts. They mostly provide strong consistency properties such as serializability.

On the other hand, production systems such as Google's Spanner [17], Megastore [9] and Percolator [31], Yahoo's Omid1 [25], Cask's Tephra [6], and more [22, 30, 5], are inclined towards separating the responsibilities of each layer. These systems, like the current work, reuse an existing persistent highly-available data-tier; for example, Megastore is layered on top of Bigtable [16], Warp [22] uses HyperDex [21], and CockroachDB [5] uses RocksDB.

Omid most closely resembles Tephra [6] and Omid1 [25], which also run on top of a distributed key-value store and leverage a centralized TM (sometimes called *oracle*) for timestamp allocation and conflict resolution. However, Omid1 and Tephra store all the information about committed and aborted transactions in the TM's RAM, and proactively duplicate it to every client that begins a transaction (in order to allow the client to determine locally which non-committed data should be excluded from its reads). This approach is not scalable, as the information sent to clients can consist of many megabytes. Omid avoids such bandwidth overhead by storing pertinent information in a metadata table that clients can access as needed. Our performance measurements in Section 7 below show that Omid significantly out-performs Omid1, whose design is very close to Tephra's. For high availability, Tephra and Omid1 use a write-ahead log, which entails long recovery times for replaying the log; Omid, instead, reuses the inherent availability of the underlying key-value store, and hence recovers quickly from failures.

Percolator also uses a centralized "oracle" for timestamp allocation but resolves conflicts via two-phase commit, whereby clients lock database records rendering them inaccessible to other transactions; the Percolator paper does not discuss high availability. Other systems like Spanner and CockroachDB allot globally increasing timestamps using a (somewhat) synchronized clock service. Spanner also uses two-phase commit whereas CockroachDB uses distributed conflict resolution where read-only transactions can cause concurrent update transactions to abort. In contrast, Omid never locks (or prevents access to) a database record, and never aborts due to conflicts with read-only transactions.

The use cases production systems serve allow them to provide SI [31, 25, 6, 5], at least for read-only transactions [17]. It is nevertheless straightforward to extend Omid to provide serializability, similarly to a serializable extension of Omid1 [35] and Spanner [17]; it is merely a matter of extending the conflict analysis to cover read-sets [24, 14], which may degrade performance.

A number of other recent efforts avoid the complexity of two-phase commit [26] by serializing transactions using a global serialization service such as highly-available log [11, 23, 13] or totally-ordered multicast [15]. Omid is unique in utilizing a single transaction manager to resolve conflicts in a scalable way.

4 Service Semantics and Interface

Omid provides transactional access to a large collection of persistent data items identified by unique keys. The service is persistent and highly available, whereas its clients are ephemeral, i.e., they are alive only when performing operations and may fail at any time. We now discuss the semantics and API Omid offers.

Semantics. A *transaction* is a sequence of put and get operations on different objects that ensures the so-called

ACID properties: *atomicity* (all-or-nothing execution), *consistency* (preserving each object's semantics), *isolation* (in that concurrent transactions do not see each other's partial updates), and *durability* (whereby updates survive crashes).

Different isolation levels can be considered for the third property. Omid opts for snapshot isolation [12], which is provided by popular database technologies such as Oracle, PostgreSQL, and SQL Server. Intuitively, SI ensures that the information a transaction retrieves from the database does not mix old and new values. For example, if a task updates the values of two stocks, then no other transaction may observe the old value of one of these stocks and the new value of the other. More precisely, SI enforces a total order on committed transactions according to their commit times so that

1. each transaction's get operations see a consistent snapshot of the database reflecting put operations by exactly those transactions that committed prior to the transaction's start time; and
2. a transaction commits only if none of the items it updates has been modified since that snapshot.

Note that under SI, concurrent transactions conflict only if they *update* the same item, whereas with serializability, a transaction that updates an item conflicts with transactions that *get* that item. Thus, for read-dominated workloads, SI is amenable to implementations (using multi-versioned concurrency control) that allow more concurrency than serializable ones, and hence scale better.

API. Omid's client API offers abstractions both for control (*begin, commit*, and *abort*) and for data access (*get* and *put*). Programmers delineate transactions via the begin and commit APIs: the sequence of get and put operations a client invokes between begin and commit pertains to one transaction. Following a commit call, the transaction may successfully *commit*, whereby all of its operations take effect; in case of conflicts, (i.e., when two concurrent transactions attempt to update the same item), the transaction may *abort*, in which case none of its changes take effect. An abort may also be initiated by the programmer, e.g., on encountering an error. Applications typically retry a transaction upon (either type of) abort. Omid's API is offered by a client library, which accesses the data directly in the data store, and interacts with the TM only to begin, commit, or abort transactions.

5 Transaction Processing

We now explain how Omid's TM manages transactions so as to guarantee SI semantics. For clarity of the exposition, we defer discussion of the TM's reliability to the next section; for now, let us assume that this component never fails. We describe Omid's data model in Section 5.1, then proceed to describe the client operation in Section 5.2 and the TM's operation in Section 5.3.

5.1 Data and metadata

Omid employs optimistic concurrency control with commit-time conflict resolution. Intuitively, with SI, a transaction's reads all appear to occur at one logical point in time (namely, when it begins), while its writes appear to execute at a later point— when it commits. Omid therefore associates two timestamps with each transaction: a *read timestamp* ts_r when it begins, and a *commit timestamp* ts_c upon commit. Both timestamps are provided by the TM using a logical clock it maintains. In addition, each transaction has a unique transaction id *txid*, for which we use the read timestamp; in order to ensure its uniqueness, the TM increments the clock whenever a transaction begins.

The data store is multi-versioned. A write operation by a transaction starting at some time t needs to be associated with a version number that exceeds all those written by transactions that committed before time t. However, the version order among concurrent transactions that attempt to update the same key is immaterial, since at least one of these transactions is doomed to abort. To ensure the former, we use the writing transaction's *txid*, which exceeds that of all previously committed transactions, as the version number. Multi-versioning is then used in conjunction with the read timestamp to ensure that a transaction's get operations observe a consistent snapshot of the data reflecting all updates by transactions that committed before the reading transaction began.

Since transaction commit needs to be an atomic step, Omid tracks the list of committed transactions in a persistent *Commit Table (CT)*, as shown in Table 1, which in the current implementation is also stored in HBase. Each entry in the CT maps a committed transaction's *txid* to its respective ts_c. When deciding to commit a transaction, the TM writes the $(txid, ts_c)$ pair to the CT, which makes the transaction durable, and is considered the commit point of the transaction. Gets refer to the CT using the *txid* in the data record in order to find out if a read value has been committed or not. In case it has, they use the commit timestamp to decide whether the value appears in their snapshot.

While checking the CT for every read ensures correctness, it imposes communication costs and contention on the CT. To avoid this overhead, Omid augments each record in the data store with a *commit field (cf)*, indicating whether the data is committed, and if it is, its commit timestamp. Initially the commit field is *nil*, indicating that the write is *tentative*, i.e., potentially uncommitted.

Data Table

key	value	version (*txid*)	commit (cf)
k_1	a	**5**	7
k_1	b	**8**	nil

Commit Table

txid	commit *ts*
5	7

Table 1: **Omid data and metadata**. Data is multi-versioned, with the *txid* as the version number. The *commit field* indicates whether the data is committed, and if so, its commit timestamp. The commit table (CT) maps incomplete committed transaction ids to their respective commit timestamps. Transaction 5 has already committed and updated cf for k_1, but has not yet removed itself from CT; transaction 8 is still pending.

Following a commit, the transaction updates the commit fields of its written data items with its ts_c, and then removes itself from the CT. Only then, the transaction is considered complete. A background cleanup process helps old (crashed or otherwise slow) committed transactions complete.

Table 1 shows an example of a key k_1 with two versions, the second of which is tentative. A transaction that encounters a tentative write during a read still refers to the CT in order to find out whether the value has been committed. In case it has, it helps complete the transaction that wrote it by copying its ts_c to the commit field. The latter is an optimization that might reduce accesses to the commit table by ensuing transactions.

5.2 Client-side operation

Transactions proceed optimistically and are validated at commit time. In the course of a transaction, a client's get operations read a snapshot reflecting the data store state at their read timestamp, while put operations write tentative values with *txid*. Since SI needs to detect only write-write conflicts, only the transaction's *write-set* is tracked. The operations, described in pseudocode in Algorithm 1, execute as follows:

Begin. The client obtains from the TM a read timestamp ts_r, which also becomes its transaction id (*txid*). The TM ensures that this timestamp exceeds all the commit timestamps of committed transactions and precedes all commit timestamps that will be assigned to committing transactions in the future.

Put(key,val). The client adds the tentative record to the data store (line 6) and tracks the key in its local *write-set*. To reduce memory and communication overheads, we track 64-bit hashes rather than full keys.

Get(key). A get reads from the data store (via ds.get()) records pertaining to *key* with versions smaller than ts_r, latest to earliest (line 9), in search of the value written

Algorithm 1 Omid's client-side code.

```
1:  local variables txid, write-set
2:  procedure BEGIN
3:      txid ← TM.BEGIN()
4:      write-set ← ∅
5:  procedure PUT(key, value)
6:      ds.put(key, value, txid, nil)
7:      add 64-bit hash of key to write-set
8:  procedure GET(key)
9:      for rec ← ds.get(key, versions down from ts_r) do
10:         if rec.commit≠nil then                    ▷ not tentative
11:             if rec.commit < ts_r then
12:                 return rec.value
13:         else                                      ▷ tentative
14:             value ← GETTENTATIVEVALUE(rec, key)
15:             if value ≠nil then
16:                 return value
17:     return nil
18: procedure GETTENTATIVEVALUE(rec,key)
19:     lookup rec.version in CT
20:     if present then                               ▷ committed
21:         update rec.commit                         ▷ helping
22:         if rec.commit < ts_r then return rec.value
23:     else                    ▷ re-read version not found in CT
24:         rec ← ds.get(key, rec.version)
25:         if rec.commit≠nil ∧ rec.commit < ts_r then
26:             return rec.value
27:     return nil
28: procedure COMMIT
29:     ts_c ← TM.COMMIT(txid, write-set)
30:     for all key in write-set do
31:         rec ← ds.get(key, txid)
32:         if ts_c = ⊥ then                          ▷ abort
33:             remove rec
34:         else
35:             rec.cf ← ts_c
36:     remove record with txid from CT
```

for this key by the latest transaction whose ts_c does not exceed ts_r (i.e., the latest version written by a transaction that committed before the current transaction began).

If the read value is committed with a commit timestamp lower than ts_r, it is returned (line 12). Upon encountering a tentative record (with cf=*nil*), the algorithm calls GETTENTATIVEVALUE (line 18) in order to search its ts_c in the CT. If this *txid* was not yet written, then it can safely be ignored, since it did not commit. However, a subtle race could happen if the transaction has updated the commit timestamp in the data store and then removed itself from the CT between the time the record was read and the time when the CT was checked. In order to discover this race, a record is re-read after its version is not found in the CT (line 23). In all cases, the first value encountered in the backward traversal with a commit timestamp lower than ts_r is returned.

Commit. The client requests *commit(txid, write-set)* from the TM. The TM assigns it a commit timestamp ts_c and checks for conflicts. If there are none, it commits the transaction by writing $(txid, ts_c)$ to the CT and returns a response. Following a successful commit, the client writes ts_c to the commit fields of all the data items it wrote to (indicating that they are no longer tentative), and finally deletes its record from the CT. Whether the commit is successful or not a background process helps transactions to complete or cleans their uncommitted records from the data store, thereby overcoming client failures.

5.3 TM operation

The TM uses an internal (thread-safe) clock to assign read and commit timestamps. Pseudocode for the TM's begin and commit functions is given in Algorithm 2; both operations increment the clock and return its new value. Thus, read timestamps are unique and can serve as transaction ids. Begin returns once all transactions with smaller commit timestamps are finalized, (i.e., written to the CT or aborted).

Commit involves compute and I/O aspects for conflict detection and CT update, resp. The TM uses a pipelined SEDA architecture [34] that scales each of these stages separately using multiple threads. Note that the I/O stage also benefits from such parallelism since the CT can be sharded across multiple storage nodes and yield higher throughput when accessed in parallel.

In order to increase throughput, writes to the commit table are batched. Both begin and commit operations need to wait for batched writes to complete before they can return – begin waits for all smaller-timestamped transactions to be persisted, while commit waits for the committing transaction. Thus, batching introduces a tradeoff between I/O efficiency, (i.e., throughput), and begin/commit latency.

The CONFLICTDETECT function checks for conflicts using a hash table in main memory. (The TM's compute aspect is scaled by running multiple instances of this function for different transactions, accessing the same table in separate threads.) For the sake of conflict detection, every entry in the write-set is considered a *key*, (though in practice it is a 64-bit hash of the appropriate key). Each *bucket* in the hash table holds an array of pairs, each consisting of a key hashed to this bucket and the ts_c of the transaction that last wrote to this key.

CONFLICTDETECT needs to (i) validate that none of the keys in the write-set have versions larger than *txid* in the table, and (ii) if validation is successful, update the table entries pertaining to the write-set to the transaction's newly assigned ts_c. However, this needs to be done atomically, so two transactions committing in parallel won't miss each other's updates. Since holding a lock on the entire table for the duration of the conflict detection procedure would severely limit concurrency, we instead limit the granularity of atomicity to a single bucket: for each key in the write-set, we lock the corresponding bucket (line 52), check for conflicts in that bucket (line 54), and if none are found, optimistically add the key with the new ts_c to the bucket (lines 56–61). The latter might prove redundant in case the transaction ends up aborting due to a conflict it discovers later. However, since our abort rates are low, such spurious additions rarely induce additional aborts.

Algorithm 2 TM functions.

```
37: procedure BEGIN
38:     txid = Clock.FetchAndIncrement()
39:     wait until there are no pending commit operations
40:           with ts_c < txid
41:     return txid

42: procedure COMMIT(txid, write-set)
43:     ts_c ← Clock.FetchAndIncrement()
44:     if ConflictDetect(txid, write-set) = COMMIT then
45:         UpdateCT(txid, ts_c)              ▷ proceed to I/O stage
46:         return ts_c
47:     else
48:         return ABORT

49: procedure CONFLICTDETECT(txid,write-set)
50:     for all key ∈ write-set do
51:         b ← key's bucket
52:         lock b
53:         small ← entry with smallest ts in b
54:         if ∃ (key, t) ∈ b s.t. t > txid then          ▷ conflict
55:             unlock b; return ABORT
                ▷ no conflict on key found – update hash table
56:         if ∃ (key, t) ∈ b s.t. t < txid then
57:             overwrite (key, t) with (key, ts_c)
58:         else if ∃ empty slot s ∈ b then
59:             write (key, ts_c) to s
60:         else if small.t ≤ txid then
61:             overwrite small with (key, ts_c)
62:         else                                  ▷ possible conflict
63:             unlock b; return ABORT
64:         unlock b
65:     return COMMIT
```

A second challenge is to limit the table size and garbage-collect information pertaining to old commits. Since a transaction need only check for conflicts with transactions whose ts_c exceeds its *txid*, it is safe to remove all entries that have smaller commit times than the *txid* of the oldest active transaction. Unfortunately, this observation does not give rise to a feasible garbage collection rule: though transactions usually last few tens of milliseconds, there is no upper bound on a transaction's life span, and no way to know whether a given outstand-

ing transaction will ever attempt to commit or has failed. Instead, we use the much simpler policy of restricting the number of entries in a bucket. Each bucket holds a fixed array of the most recent (key, ts_c) pairs. In order to account for potential conflicts with older transactions, a transaction also aborts in case the minimal ts_c in the bucket exceeds its *txid* (line 62). In other words, a transaction expects to find, in every bucket it checks, at least one commit timestamp older than its start time or one empty slot, and if it does not, it aborts.

The size of the hash table is chosen so as to reduce the probability for spurious aborts, which is the probability of all keys in a given bucket being replaced during a transaction's life span. If the throughput is T transactional updates per second, a bucket in a table with e entries will overflow after e/T seconds on average. For example, if 10 million keys are updated per second, a bucket in a one-million-entry table will overflow only after 100ms on average, which is much longer than most transactions. We further discuss the impact of the table size in Section 7.

Garbage collection. A dedicated background procedure (*co-processor*) cleans up old versions. To this end, the TM maintains a *low water mark*, which is used in two ways: (1) the co-processor scans data store entries, and keeps, for each key, the biggest version that is smaller than the low water mark along with all later versions. Lower versions are removed. (2) When a transaction attempts to commit, if its *txid* is smaller than the low water mark, it aborts because the co-processor may have removed versions that ought to have been included in its snapshot. The TM attempts to increase the low water mark when the probability of such aborts is small.

6 High Availability

Very-high-end Omid-powered applications are expected to work around the clock, with a mean-time-to-recover of just a few seconds. Omid therefore needs to provide *high availability (HA)*. Given that the underlying data store is already highly available and that client failures are tolerated by Omid's basic transaction processing protocol, Omid's HA solution only needs to address TM failures. This is achieved via the primary-backup paradigm: during normal operation, a single *primary* TM handles client requests, while a *backup* TM runs in hot standby mode. Upon detecting the primary's failure, the backup performs a *failover* and becomes the new primary.

The backup TM may falsely suspect that the primary has failed. The resulting potential simultaneous operation of more than one TM creates challenges, which we discuss in Section 6.1. We address these in Section 6.2 by adding synchronization to the transaction commit step. While such synchronization ensures correctness, it also introduces substantial overhead. We then optimize the solution in Section 6.3 to forgo synchronization during normal (failure-free) operation.

Our approach thus resembles many popular protocols, such as Multi-Paxos [29] and its variants, which expedite normal mode operation as long as an agreed leader remains operational and unsuspected. However, by relying on shared persistent state in the underlying highly available data store, we obtain a simpler solution, eliminating the need to synchronize with a quorum in normal mode or to realign state during recovery.

6.1 Failover and concurrent TMs

The backup TM constantly monitors the primary's liveness. Failure detection is timeout-based, namely, if the primary TM does not re-assert its existence within a configured period, it is deemed failed, and the backup starts acting as primary. Note that the primary and backup run independently on different machines, and the time it takes the primary to inform the backup that it is alive can be unpredictable due to network failures and processing delays, (e.g., garbage-collection stalls or long I/O operations). But in order to provide fast recovery, it is undesirable to set the timeout conservatively so as to ensure that a live primary is never detected as faulty.

We therefore have to account for the case that the backup performs failover and takes over the service while the primary is operational. Though such simultaneous operation of the two TMs is a necessary evil if one wants to ensure high availability, our design strives to reduce such overlap to a minimum. To this end, the primary TM actively checks if a backup has replaced it, and if so, "commits suicide", i.e., halts. However, it is still possible to have a (short) window between the failover and the primary's discovery of the existence of a new primary when two primary TMs are active.

When a TM fails, all the transactions that began with it and did not commit (i.e., were not logged in the CT) are deemed aborted. However, this clear separation is challenged by the potential simultaneous existence of two TMs. For example, if the TM fails while a write it issued to the CT is still pending, the new TM may begin handling new transactions before the pending write takes effect. Thus, an old transaction may end up committing after the new TM has begun handling new ones. Unless handled carefully, this can cause a new transaction to see partial updates of old ones, as illustrated in Figure 2. To avoid this, we must ensure that once a new transaction obtains a read timestamp, the status of all transactions with smaller commit timestamps does not change.

A straightforward way to address the above challenge

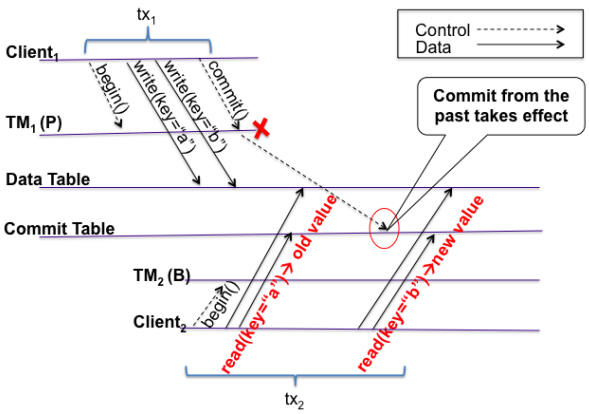


Figure 2: **The challenge with two concurrent TMs.** An old transaction, tx_1, commits while a new one tx_2 is processed, causing tx_2 to see an inconsistent snapshot.

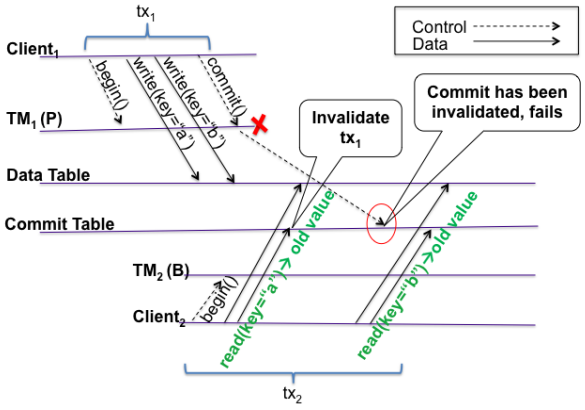


Figure 3: **Addressing the challenge of two concurrent TMs.** The old transaction is invalidated by the new one and therefore cannot commit.

is via mutual exclusion, i.e., making sure that at most one TM commits operations at a time. However, this solution would entail synchronization upon each commit, not only at failover times, which would adversely affect performance. We therefore forgo this option.

6.2 Basic HA algorithm

Upon failover from TM_1 (the old primary) to TM_2 (the new one), we strive to ensure the following properties:

P1 all timestamps assigned by TM_2 exceed all those assigned by TM_1;

P2 after a transaction tx_2 with read timestamp $ts2_r$ begins, no transaction tx_1 that will end up with a commit timestamp $ts1_c < ts2_r$ can update any additional data items (though it may still commit); and

P3 when a transaction reads a tentative update, it can determine whether this update will be committed with a timestamp smaller than its read timestamp or not.

Properties P1–P3 are sufficient for SI: P1 implies that commit timestamps continue to be totally ordered by commit time, P2 ensures that a transaction encounters every update that must be included in its snapshot, and P3 stipulates that the transaction can determine whether to return any read value.

To ensure the first two properties, the TMs publish the read timestamps they allot as part of initiating a transaction in a persistent shared object, *maxTS*. Before committing, the TM checks maxTS. If it finds a timestamp greater than its last committed one, it deduces that a new TM is active, aborts the transaction attempting to commit, and halts.

In Figure 2 we saw a scenario where the third property, P3, is violated— when tx_2 reads key a it cannot tell that tx_1, which wrote it, will end up committing with a smaller $ts1_c$ than $ts2_r$. This leads to an inconsistent snapshot at tx_2, as it sees the value of key b written by tx_1.

To enforce P3, tx_2 cannot wait for TM_1, because the latter might have failed. Instead, we have tx_2 proactively abort tx_1, as illustrated in Figure 3. More generally, when a read encounters a tentative update whose *txid* is not present in the CT, it forces the transaction that wrote it to abort. We call this *invalidation*, and extend the CT's schema to include an *invalid* field to this end. Invalidation is performed via an atomic put-if-absent (supported by HBase's *checkAndMutate* API) to the CT, which adds a record marking that tx_1 has "invalid" status. The use of an atomic put-if-absent achieves *consensus* regarding the state of the transaction.

Commits, in turn, read the CT after adding the commit record in order to check whether an invalidation record also exists, and if so, halt without returning a commit response to the client. In addition, every read of a tentative update checks its invalid field in the CT, and ignores the commit record if the transaction has been invalidated.

While this solution satisfies the three required properties, it also induces a large number of synchronization steps: (i) writing allotted read timestamps to maxTS to ensure P1; (ii) checking maxTS at commit time to ensure P2; and (iii) checking the CT for invalidation at the end of every commit to ensure P3. The next section presents an optimization that reduces the cost of synchronization.

6.3 Synchronization-free normal operation

In order to eliminate the synchronization overhead most of the time, Omid's HA solution uses two mechanisms. First, to reduce the overheads (i) and (ii) associated

with timestamp synchronization, it allocates timestamp ranges in large chunks, called *epochs*. That is, instead of incrementing maxTS by one timestamp at a time, the TM increments it by a certain *range*, and is then free to allot timestamps in this range without further synchronization. Second, to reduce cost (iii) of checking for invalidations, it uses locally-checkable *leases*, which are essentially locks that live for a limited time. As with locks, at most one TM may hold the lease at a given time (this requires the TMs' clocks to advance roughly at the same rate). Omid manages epochs and leases as shared objects in Zookeeper, and accesses them infrequently.

Algorithm 3 summarizes the changes to support HA. On the TM side, CHECKRENEW is called at the start of every commit and begin. It first renews the lease every δ time, for some parameter δ (lines 68–70). This parameter defines the tradeoff between synchronization frequency and recovery time: the system can remain unavailable for up to δ time following a TM failure. Since clocks may be loosely synchronized, Omid defines a *guard period* of $\delta' < \delta$, so that the lease must be renewed at least δ' time before it expires. The production default for δ' is $\delta/4$. The primary TM fails itself (halts) if it cannot renew the lease prior to that time. From the clients' perspective, this is equivalent to a TM crash (line 70). Second, CHECKRENEW allocates a new epoch if needed (lines 71–74).

The backup (not shown in pseudocode) regularly checks the shared lease, and if it finds that it has expired, it immediately sets its clock to exceed maxTS, allocates a new epoch for itself (by increasing maxTS), and begins serving requests, *without any special recovery procedure*. Since the epoch claimed by a new TM always exceeds the one owned by the old one, Property P1 holds.

Property P2 is enforced by having the TM (locally) check that its lease is valid before committing a transaction (lines 68–70). Since at most one TM can hold the lease at a given time, and since the commit is initiated after all writes to items that are part of the transaction complete, Property P2 holds.

Nevertheless, the lease does not ensure Property P3, since the lease may expire while the commit record is in flight, as in the scenario of Figures 2 and 3. To this end, we use the invalidation mechanism described above. However, we limit its scope as follows: (1) A commit needs to check whether the transaction has been invalidated only if the TM's lease has expired. This is done in the TMCHECKINVALIDATE function. (2) A read needs to invalidate a transaction only if it pertains to an earlier epoch of a different TM. We extend client's GETTENTATIVEVALUE function to perform such invalidation in Algorithm 3 lines 83, 87–96. Note that a transaction reading a tentative update still checks its validity status regardless of the epoch, in order to avoid "helping" invalidated transactions complete their tentative updates.

Algorithm 3 Omid's HA algorithm.

```
66: procedure CHECKRENEW
67:         ▷ called by the TM at start of BEGIN and COMMIT
68:     if lease < now + δ′ then
69:         renew lease for δ time                    ▷ atomic operation
70:         if failed then halt
71:     if Clock = epoch then
72:         epoch ← Clock + range
73:         extend maxTS from Clock to epoch
74:         if failed then halt

75: procedure TMCHECKINVALIDATE(txid)
76:                 ▷ called by the TM before COMMIT returns
77:     if lease < now + δ′ then
78:         if txid invalid in CT then halt

79: procedure GETTENTATIVEVALUE(REC)
80:                 ▷ replaces same function from Algorithm 1
81:     lookup rec.version in CT
82:     if present then
83:         if invalidated then return nil
84:         update rec.commit                                ▷ helping
85:         if rec.commit < ts_r then return rec.value
86:     else          ▷ new code – check if need to invalidate
87:         if rec.version ∈ old epoch by an old TM then
88:             invalidate t in CT                  ▷ try to invalidate
89:             if failed then
90:                 lookup rec.version in CT
91:                 if invalidated then return nil
92:                 update rec.commit                        ▷ helping
93:                 if rec.commit < ts_r then
94:                     return rec.value
95:             else                                     ▷ invalidated
96:                 return nil
97:         else                ▷ original code – no invalidation
98:             rec ← ds.get(key, rec.version)
99:             if rec.commit≠nil ∧ rec.commit < ts_r then
100:                return rec.value
101:    return nil
```

Finally ,we note that on TM failover, some clients may still be communicating with the old TM. While the old TM may end up committing some of their requests, a problem arises if the client times out on the old TM before getting the commit response, since the client might unnecessarily retry a committed transaction. To avoid this problem, a client that times out on its TM checks the CT for the status of its transaction before connecting to a new TM. If the status is still undetermined, the client tries to invalidate the CT entry, thus either forcing the transaction to abort or learning that it was committed (in case the invalidation fails).

7 Evaluation

Omid's implementation complements Apache HBase with transaction processing. It exploits HBase to store both application data and the CT metadata. HBase, the TMs, and Zookeeper are all deployed on separate dedicated machines.

In large-scale deployments, HBase tables are sharded (partitioned) into multiple *regions*. Each region is managed by a *region server*; one server may serve multiple regions. HBase is deployed on top of Hadoop Distributed Filesystem (HDFS), which provides the basic abstraction of scalable and reliable storage. HDFS is replicated 3-fold in all the settings described below.

Section 7.1 presents performance statistics obtained in Omid's production deployment, focusing on the end-to-end application-level overhead introduced by transaction processing. Section 7.2 further zooms in on the TM scalability under very high loads.

7.1 End-to-end performance in production

We present statistics of Omid's use in a production deployment of Sieve – Yahoo's content management system. Sieve digests streams of documents from multiple sources, processes them, and indexes the results for use in search and personalization applications. Each document traverses a pipeline of tasks, either independently or as part of a mini-batch. A task is an ACID processing unit, framed as a transaction. It typically reads one or more data items generated by preceding tasks, performs some computation, and writes one or more artifacts back.

Sieve scales across task pipelines that serve multiple products, performing tens of thousands of tasks per second on multi-petabyte storage. All are powered by a single Omid service, with the CT sharded across 10 regions managed by 5 region servers. Sieve is throughput-oriented, and favors scalability over transaction latency.

Figure 4 presents statistics gathered for five selected Sieve tasks. For each task, we present its average latency broken down to components – HBase access (two bottom components in each bar), compute time, and the TM's begin and commit (top two components). In this deployment, Omid updates the commit fields synchronously upon commit, that is, commit returns only after the commit fields of the transaction's write-set have been updated. Note that since a begin request waits for all transactions with smaller *txids* to commit, its processing latency is similar to that of a commit operation, minus the time commit takes to update the commit fields.

We see that for tasks that perform significant processing and I/O, like document inversion and streaming to index, Omid's latency overhead (for processing begin and commit) is negligible – 2–6% of the total transaction duration. In very short tasks such as duplicate detection and out-link processing, Omid accounts for up to roughly one third of the transaction latency.

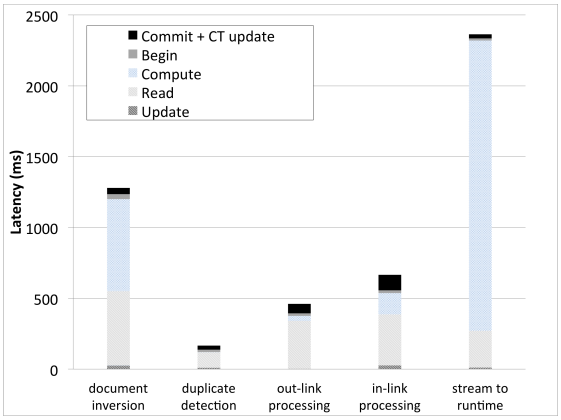


Figure 4: **Transaction latency breakdown in production deployment of Omid in Sieve.** The top two components represent transaction management overhead.

The transaction abort rates observed in Sieve are negligible (around 0.002%). They stem from either transient HBase outages or write-write conflicts, e.g., concurrent in-link updates of extremely popular web pages.

7.2 TM microbenchmarks

We now focus on TM performance. To this end, our microbenchmarks invoke only the TM's begin and commit APIs, and do not access actual data. We run both the TM and HBase (holding the CT) on industry-standard 8-core Intel Xeon E5620 servers with 24GB RAM and 1TB magnetic drive. The interconnects are 1Gbps Ethernet.

We generate workloads in which transaction write-set sizes are distributed Zipf, i.e., follow a power-law ($Pr[X \geq x] = x^{-\alpha}$) with exponent values of $\alpha = 1.2$, $\alpha = 1.6$, and $\alpha = 2$ (the smaller the heavier-tailed), cut-off at 256 keys. Each transaction's latency, (i.e., the time we wait after invoking its begin and before invoking its commit), is set to 5ms per write. Note that read-sets are not sent to the TM and hence their size is immaterial.

We note that key selection affects *real* conflicts: if the written keys are drawn from a heavy-tailed distribution, then two concurrent transactions are likely to update the same key, necessitating one of them to abort. Since this is an artifact of the workload, which is unaffected by our system design, we attempt to minimize this phenomenon in our experiments. We therefore uniformly sample 64-bit integers for the key hashes. Recall that our experience in production shows that real conflicts are indeed rare.

We begin by evaluating scalability, which is our principal design goal. The TM throughput is constrained by two distinct resources – the storage access required

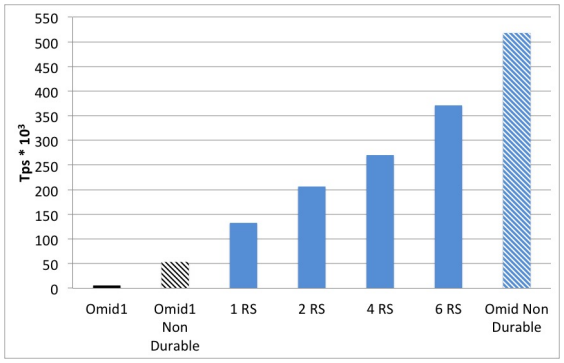


Figure 5: **Scalability of Omid's CT updates with the number of HBase region servers, and comparison with Omid1.** Non-durable versions do not persist transactions and thus provide upper bounds on throughput under perfect storage scaling.

for persisting commits in the CT, and the compute resources used for conflict detection. These resources scale independently: the former, evaluated in Section 7.2.1, scales out across multiple HBase region servers, whereas the latter scales up on multi-core hardware, and is studied in Section 7.2.2. Section 7.2.3 then evaluates the throughput-latency tradeoff that Omid exhibits when using a single region server. Finally, in Section 7.2.4, we exercise Omid's high-availability mechanism.

7.2.1 Commit table scalability

Since the commit records are fixed-length (two 64-bit integers), the CT performance does not depend on transaction sizes, and so we experiment only with $\alpha = 1.6$. Recall that in order to optimize throughput, the TM batches writes to the CT and issues multiple batches in parallel. Experimentally, we found that the optimal number of concurrent CT writer threads is 4, and the batch size that yields the best throughput is 2K transactions per writer.

Figure 5 depicts Omid's commit rate as function of the number of HBase region servers, which scales to almost 400K tps. It further compares Omid's throughput to that of Omid1 [25], which, similarly to Omid, runs atop HBase, and uses a centralized TM. It is worth noting that even in the single-server configuration, Omid outperforms Omid1 by more than 25x. This happens because upon each begin request, Omid1 sends to the client a large amount of information (equivalent to the combination of Omid's CT and the in-memory conflict detection table). This saturates the CPU and network resources.

The "non-durable" bars – leftmost and second from the right – represent experiments where commits are not persisted to stable storage. In Omid this means forgoing the write to the CT, whereas in Omid1 it means disabling the write to BookKeeper in which the system stores its commit log. These results provide upper bounds on the throughput that can be obtained with perfect storage scaling in both systems. Omid peaks at 518K transactions per second, whereas Omid1 peaks at 50K.

7.2.2 Conflict detection scalability

In the experiment reported above, the conflict detection algorithm is evaluated as part of the system. There, the commit table I/O is the bottleneck, and the conflict detection process can keep up with the pace of four I/O threads even when running sequentially, i.e., in a single thread.

We next focus on scale-up of this component running by itself using 1 to 8 threads, in order to study its potential scalability in even larger configurations. The experiment employs a conflict table of 128M 64-bit integers (1G total size). The bucket size is 32 integers, i.e., the table is 4M buckets big.

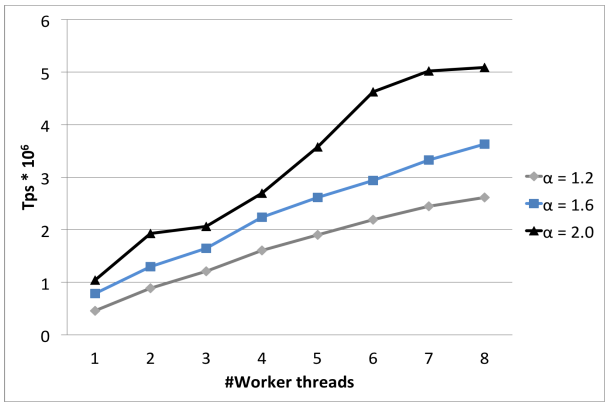
Figure 6(a) illustrates the processing rate. As expected processing shorter transactions (a bigger α) is faster. The rate scales to 2.6M transactions per second for $\alpha = 1.2$, and to 5M for $\alpha = 2$. Note that exercising such high throughput in a complete system would require an order of magnitude faster network to sustain the request/response packet rate. Clearly the TM's compute aspect is far from being a bottleneck.

Finally, we analyze the false abort rate. (The uniform sampling of key hashes and relatively short transaction latencies render real collisions unlikely, hence all aborts are deemed false). The overall abort rate is negligibly small. In Figure 6(b) we zoom-in on transactions clustered into three buckets: shorter than 8 writes, 8 to 63 writes, and 64+ writes. The worst abort rate is below 0.01%. It occurs, as expected, for long transactions in the most heavy-tailed distribution. Further reduction of the false abort rate would require increasing the table size or using multiple hashes (similarly to Bloom filters).

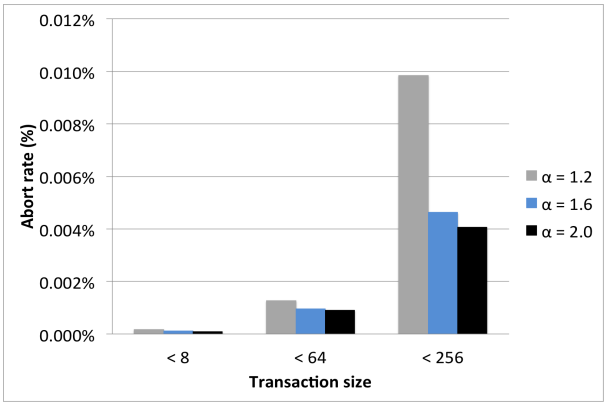
7.2.3 Latency-throughput tradeoff

We now examine the impact of load on TM access latency with a single region server managing the CT. We use here $\alpha = 1.6$. For every given system load, the batch size is tuned for optimal latency: under light load, no batching is employed, (i.e., commits are written one at a time), whereas under high load, we use batches of 10K.

Figure 7 reports the average client-side latency of commit operations, broken down to three components: (1) network round-trip delay and conflict detection, which are negligible, and do not vary with the load or batch size; (2) HBase CT write latency, which increases with the batch size; and (3) queueing delay at the TM, which increases with the load. Begin latency is similar, and is therefore omitted. We increase the load up to 70K



(a) Conflict detection scalability



(b) Conflict detection false abort rate

Figure 6: **Conflict detection scalability and false abort rate.** Transaction write-set sizes are distributed power-law ($Pr[X \geq x] = x^{-\alpha}$) with exponent values of $\alpha = 1.2$, $\alpha = 1.6$, and $\alpha = 2$ (the smaller the heavier-tailed); the key hashes are 64-bit integers, uniformly sampled to avoid real conflicts whp; transaction latency is 5ms per write.

transactions per second, after which the latency becomes excessive; to exceed this throughput, one may use multiple region servers as in the experiment of Section 7.2.1.

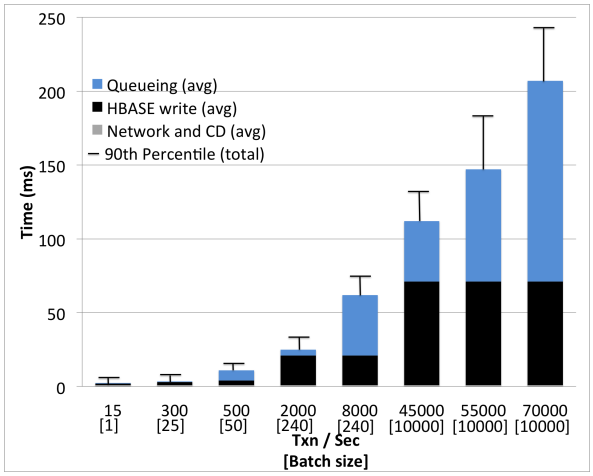


Figure 7: **Omid throughput vs. latency.** Client-perceived commit latency (average broken down and 90% of total); single region server; power-law transaction sizes with $\alpha = 1.6$; batch sizes optimized for minimum latency (in square brackets below each bar).

7.2.4 High availability

Finally, we exercise the high-availability mechanism. As long as the primary TM does not fail, HA induces negligible overhead. We now examine the system's recovery following a primary TM failure. The failure detection timeout is $\delta = 1$ sec. Figure 8 depicts the system throughput over time, where the primary TM is forcefully shut down after 40 sec, is then allowed to recover, and the new primary (original backup) is shut down after 120 sec. The primary is shut down two more times at 180 and 220 sec; the failover completes within 4 sec.

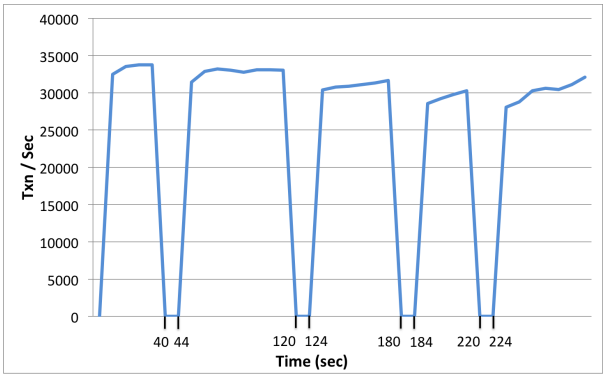


Figure 8: **Omid throughput with four failovers**; recovery takes around 4 seconds.

8 Lessons Learned and Future Work

Omid was originally designed as a foundational building block for Sieve – Yahoo's next-generation content management platform. The need for transactions emerges in scenarios similar to Percolator [31]. Analogously to other data pipelines, Sieve is more throughput-sensitive than latency-sensitive. This has led to a design that trades off latency for throughput via batching. The original design of Omid1 [25] did not employ a CT, but instead had the TM send clients information about all pending transactions. This design was abandoned due to limited scalability in the number of clients, and was replaced by

Omid, which uses the CT to track transaction states. The CT may be sharded for I/O scalability, but its update rate is bounded by the resources of the single (albeit multi-threaded) TM; this is mitigated by batching.

Since becoming an Apache Incubator project, Omid is witnessing increased interest, in a variety of use cases. Together with Tephra, it is being considered for use by Apache Phoenix – an emerging OLTP SQL engine over HBase storage [3]. In that context, latency has increased importance. We are therefore developing a low-latency version of Omid that has clients update the CT instead of the TM, which eliminates the need for batching and allows throughput scaling without sacrificing latency. Similar approaches have been used in Percolator [31], Corfu [10], and CockroachDB [5]. We note, however, that such decentralization induces extra synchronization oerhead at commit time and may increase aborts (in particular, reads may induce aborts); the original design may be preferable for throughput-oriented systems.

Another development is using application semantics to reduce conflict detection. Specifically, some applications can identify scenarios where conflicts need not be checked because the use case ensures that they won't happen. Consider, e.g., a massive table load, where records are inserted sequentially, hence no conflicts can arise. Another example is a secondary index update, which is guaranteed to induce no conflict given that the primary table update by the same transaction has been successful. To reduce overhead in such cases, we plan to extend the write API to indicate which written keys need to be tracked for conflict detection.

On the scalability side, faster technologies may be considered to maintain Omid's commit metadata. In particular, since Omid's commit table is usually written sequentially and infrequently read, it might be more efficient to use log-structured storage that is better optimized for the above scenario. Modern hardware (e.g., SSD storage, RDMA networks) could bring further speedups.

Acknowledgments

We acknowledge the many contributions to Omid, as concept and code, since its early days. Our thanks go to Aran Bergman, Daniel Gomez Ferro, Yonatan Gottesman, Flavio Junqueira, Igor Katkov, Francis Christopher Liu, Ralph Rabbat, Benjamin (Ben) Reed, Kostas Tsioutsiouliklis, Maysam Yabandeh, and the Sieve team at Yahoo. We also thank Dahlia Malkhi and the FAST reviewers for insightful comments.

References

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

[2] Apache HBase Coprocessor Introduction. https://blogs.apache.org/hbase/entry/coprocessor_introduction.

[3] Apache Phoenix. https://phoenix.apache.org.

[4] Apache Zookeeper. http://zookeeper.apache.org.

[5] CockroachDB. https://github.com/cockroachdb/cockroach/blob/master/docs/design.md.

[6] Tephra: Transactions for Apache HBase. https://tephra.io.

[7] AGUILERA, M. K., LENERS, J. B., AND WALFISH, M. Yesquel: Scalable sql storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 245–262.

[8] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev. 41*, 6 (Oct. 2007), 159–174.

[9] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[10] BALAKRISHNAN, M., MALKHI, D., DAVIS, J. D., PRABHAKARAN, V., WEI, M., AND WOBBER, T. CORFU: A distributed shared log. *ACM Trans. Comput. Syst. 31*, 4 (2013), 10.

[11] BALAKRISHNAN, M., MALKHI, D., WOBBER, T., WU, M., PRABHAKARAN, V., WEI, M., DAVIS, J. D., RAO, S., ZOU, T., AND ZUCK, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13, pp. 325–340.

[12] BERENSON, H., BERNSTEIN, P. A., GRAY, J., MELTON, J., O'NEIL, E. J., AND O'NEIL, P. E. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.* (1995), pp. 1–10.

[13] BERNSTEIN, P. A., REID, C. W., AND DAS, S. Hyder - a transactional record manager for shared flash. In *CIDR* (January 2011), pp. 9–20.

[14] CAHILL, M. J., ROHM, U., AND FEKETE, A. Serializable isolation for snapshot databases. In *SIGMOD* (2008), pp. 729–738.

[15] CAMARGOS, L. Sprint: a middleware for high-performance transaction processing. In *In EuroSys 07: Proceedings of the ACM SIGOPS/EuroSys Eu Conference on Computer Systems 2007* (2007), ACM, pp. 385–398.

[16] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[17] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), pp. 261–264.

[18] COWLING, J., AND LISKOV, B. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (2012), USENIX ATC'12, pp. 21–21.

[19] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), pp. 401–414.

[20] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 54–70.

[21] ESCRIVA, R., WONG, B., AND SIRER, E. G. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012), SIGCOMM '12, pp. 25–36.

[22] ESCRIVA, R., WONG, B., AND SIRER, E. G. Warp: Lightweight multi-key transactions for key-value stores. *CoRR abs/1509.07815* (2015).

[23] EYAL, I., BIRMAN, K., KEIDAR, I., AND VAN RENESSE, R. Ordering transactions with prediction in distributed object stores. In *LADIS* (2013).

[24] FEKETE, A., LIAROKAPIS, D., O'NEIL, E., NEIL, P. O., AND SHASHA, D. Making snapshot isolation serializable. *ACM TODS 30* (2005), 492–528.

[25] FERRO, D. G., JUNQUEIRA, F., KELLY, I., REED, B., AND YABANDEH, M. Omid: Lock-free transactional support for distributed data stores. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014* (2014), pp. 676–687.

[26] GRAY, J., HELLAND, P., O'NEIL, P. E., AND SHASHA, D. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.* (1996), pp. 173–182.

[27] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[28] KRASKA, T., PANG, G., FRANKLIN, M. J., MADDEN, S., AND FEKETE, A. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, pp. 113–126.

[29] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[30] PATTERSON, S., ELMORE, A. J., NAWAB, F., AGRAWAL, D., AND ABBADI, A. E. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. In *PVLDB* (2012), vol. 5, pp. 1459–1470.

[31] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)* (2010).

[32] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD '12, pp. 1–12.

[33] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 87–104.

[34] WELSH, M. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California, Berkeley, 2002.

[35] YABANDEH, M., AND GOMEZ-FERRO, D. A critique of snapshot isolation. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, pp. 155–168.

[36] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), E. L. Miller and S. Hand, Eds., ACM, pp. 263–278.