¡Viva la *Distribución*!

Building Fully-Distributed Transactions as a Service

#119

5 pages

Abstract

Despite their potential benefits, fully-distributed transactions are notoriously difficult to get right; this is the reason why recent proposals come with aggressive hardware optimizations, optimize using a centralized component or support only relaxed guarantees. This paper describes a novel transaction protocol that is fully distributed and mostly optimistic with the hope that it scales well across different levels. It is built on top of Xenon, VMware's decentralized management framework, taking full advantage of a number of key features the framework provides. We have already implemented the core protocol, which we describe in detail, and sketch a number of enhancements we are planning to add. We believe it can adapt well, and we plan to evaluate soon.

1. Introduction

Today's computing infrastructure increasingly relies on distribution: data storage and processing employ ever more computing power scattered across different clusters, data centers or even regions. In order to provide useful abstractions for developers, data platforms increasingly require the ability to manipulate multiple data objects atomically.

As a small, motivating example, suppose we have the following blocks of operations, invoked by two different clients (Tab. 1). On the left block, two counter objects, $loc1$ and $loc2$, are updated to reflect a transfer of a hundred items (*e.g.*, virtual machines) from location 1 to location 2. On the right one, the number of items on

line	"left" block	"right" block
1	`i` $\leftarrow$ `READ`$_1$`(loc1)`	`l1` $\leftarrow$ `READ`$_2$`(loc1)`
2	`WRITE`$_1$`(loc1, i-100)`	`l2` $\leftarrow$ `READ`$_2$`(loc2)`
3	`j` $\leftarrow$ `READ`$_1$`(loc2)`	`WRITE`$_2$`(loc3, l1+l2)`
4	`WRITE`$_1$`(loc2, j+100)`	

Table 1: Two blocks of operations: (left) move 100 items from loc1 to loc2; (right) get the sum of both locations.

both locations is tallied. The two counter objects may be stored on different nodes, *e.g.*, at the location they model. The main challenges are that (i) these blocks may be invoked concurrently by distinct clients and (ii) either one might fail uncompleted. For example, an interleaving of the two blocks could leave $loc3$ short of 100; and a half-completed left-hand block could result in 100 items gone "missing".

A well understood solution is to wrap the blocks within *transactions*. Transactions offer an elegant abstraction addressing the problems mentioned above, by grouping related operations into individual, indivisible blocks that either succeed or fail altogether, guaranteeing atomicity and isolation among the blocks.

Our design targets large, scalable distributed data platforms, and our goal is to support transactions via a completely decentralized mechanism. Full decentralization allows the system to scale well at many different levels, without a single point of failure or contention.

The classical way of doing this is via strict locking: with locking, isolation and atomicity between a pair of transactions is ensured because there is a point in time where one transaction holds all locks on their common items before the other. An additional guideline of our design is to avoid locks, in order to allow (possibly lengthy) read-transactions to proceed concurrently with update-transactions.

To avoid locks, we rely on multiple object versions. Indeed, in our setting, the underlying data system supports versioning. Instead of locking, when a transaction accesses an object it clones the object and leaves an annotation on it. An overlapping transaction that accesses the same object observes the annotation, but it may continue *optimistically*. At commit time, a transaction checks all overlapping transactions with common objects, and determines whether it is possible to commit. As discussed below, this framework allows a variety of commit/abort policies to be implemented.

The commit/abort decision is arranged between (only associated) transaction coordinators without support from any centralized authority. More specifically, each transaction is assigned a coordinator that is responsible for driving the transaction, possibly communicating with other coordinators to resolve any potential conflicts. Arbitration between coordinators requires a *single* round-trip of messages.

We have already implemented the core protocol (Sec. 2) in the context of VMware Xenon, a framework for building massively distributed, stateful applications as sets of collaborating nano-services (Sec. 3). We show how the protocol's main building blocks map directly onto Xenon's core abstractions (Sec. 4), in par with Xenon's design goals to scale across different levels (*i.e.*, cluster, data center, region, globe – each one posing different kinds of challenges). We also discuss a series of enhancements and optimizations (Sec. 5). Still at an early stage of development, we plan to soon evaluate extensively under different classes of workloads.

2. Distributed Transactions

Our scheme is completely distributed, in the sense that each transaction is being driven by a different coordinator. Fig. 1 (left) shows at a very high level how the two transactions described in Sec. 1 (Tab.1) proceed in our scheme, and in particular, how responsibilities are split between the two coordinators. Unbeknownst to each other, clients first begin a transaction (operation 0), where each client is assigned a transaction coordinator. Then, they invoke a number of operations (whose internal ordering is not important – operations 1, 2, 3 and 4, corresponding to the lines of code in Tab. 1) to different locations. Each location sends information to coordinators asynchronously. Finally, clients try to commit (operations 5 and 4, for green/left and yellow/right respectively). The coordina-

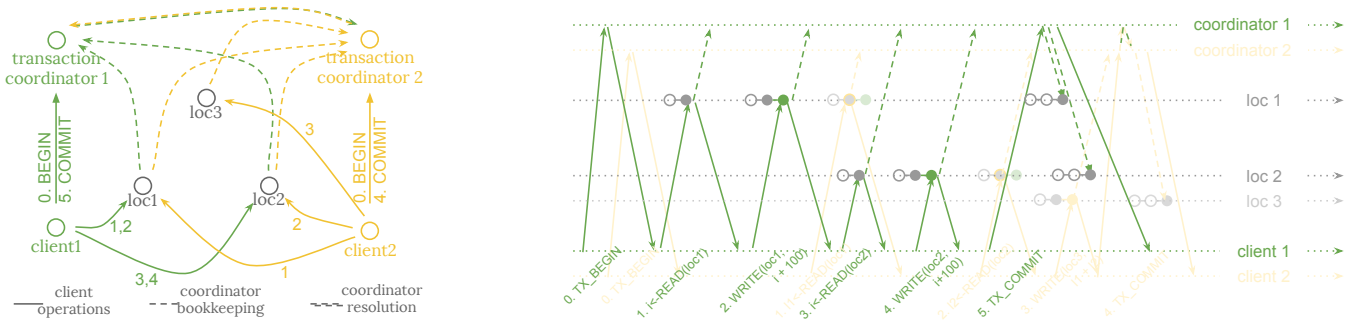

Figure 1: Distributed transaction processing (example from Sec. 1 – numbers correspond to lines from Tab.1). The left gives a high-level idea of the boundaries between the responsibilities of different coordinators; The right shows an end-to-end timeline of these transactions (focusing on the left batch of Tab.1).

tors ensure precedence ordering among all items by *communicating* with each other.

Our transaction solution relies on underlying system support of key capabilities. We assume a multiversioned, labeled, queryable object store: multiple versions of objects are kept in the system; objects can have arbitrary labels attached; and objects can be queried efficiently by any attribute.

We first provide intuition on our transaction conflict resolution mechanism. Consider a transaction that accesses an object, and before it commits, another transaction accesses the same object. If both transactions read it, there are no ordering constraints. If both transactions write it, the first one to commit must precede the other. If one transaction writes and the other one reads, then the reading transaction must precede the write; this is because, even if the read landed after the write, it still read the original version of the object – since the write has not committed yet. Essentially, the distributed transaction concurrency control mechanism verifies that one transaction can precede the other with respect to all common object. In order to be able to detect overlap, a transaction annotates each object it accesses, with the coordinator collecting all the existing pending annotations on the object.

2.1 Transaction life-cycle

Upon startup, a transaction coordinator task is started and is given a unique `tx_id`. Coordinator tasks execute as normal client code, but a coordinator's decision must persist durably in the presence of faults – at least until all concurrent transactions commit. They can reside on any node and communicate through any programming paradigm (*e.g.*, message passing, asynchronous).

During the execution of a transaction, every write operation produces a new version of the object, with the version-number attribute automatically incremented by the system. The new version of the object is annotated with `tx_id`. We call a value annotated with a transaction ID *pending*.

Read operations lookup objects by primary attribute, combined with optional selection criteria on other attributes (*e.g.*, versions). For instance, it is possible to look for an object with "key K, not marked with a pending `tx_id` attribute, and with version less than or equal to 123". The state returned is the latest value annotated with `tx_id`; if such a state does not exist, the state returned is the latest non-annotated version of the state object.

The transaction coordinator maintains a set RS of all the read objects and a set WS of all writes. Each object O in these sets includes all the pending transactions on O when it was accessed within the transaction. This set, denoted $O.CW[]$, represents the set of *potentially conflicting* transactions. A member `tx_id'` of $O.CW[]$ is itself a transaction ID, and induces a partial ordering with `tx_id`: it should either follow or precede it.

During the commit resolution of `tx_id`, the coordinator goes through every object O in WS and RS. For every pending transaction `tx_id'` in O, it interacts with the coordinator of `tx_id'` to find out if it has committed already. Subsequently, the coordinator has all the required information to determine the precedence relationship between `tx_id` and `tx_id'`, and decide commit/abort for `tx_id`. If `tx_id'` wrote O, `tx_id` read O, and `tx_id'` is still pending then `tx_id` must precede `tx_id'` (`tx_id`$\rightarrow$ `tx_id'`). If `tx_id'` read O, `tx_id` wrote O, and `tx_id'` is still pending then `tx_id` must follow `tx_id'` (`tx_id'` $\rightarrow$ `tx_id`). If `tx_id'` read O, `tx_id` wrote O, and `tx_id'` has already committed, `tx_id`is required to abort; intuitively, this makes sense: it just means that `tx_id` read a stale value at some point.

Committing or aborting removes bookkeeping information from pending, read and write sets. A commit also persists state. Aborting does not need to remove annotated state, since it is annotated as transaction-local.

The essence of the protocol is captured in pseudocode below:

```
Set<Node> readSet, writeSet
Map<Node, Coordinator> coordsOfPendingOps
Coordinator.OnResolve():
  forall s in readSet, writeSet:
    forall c in coordsOfPendingOps[s]:
      if c has not committed yet:
        ensure this precedes c
      else:
        if s in readSet: this.ABORT
        else: ensure c precedes this
Node.OnRead():
  readSet.add(node);
  coordsOfPendingOps.add(node, tx_id)
  return (latest tagged tx_id OR committed)
Node.OnWrite(newState):
  create newState and tag it with tx_id
  writeSet.add(node);
  coordsOfPendingOps.add(node, tx_id)
```

3. Overview of VMware Xenon

Xenon is a framework and associated programming model for building distributed applications at scale. It was the result of our need—and, coincidentally, the need of many groups inside and outside VMware—to make it easy to build stateful distributed applications at scale. Instead of rebuilding all the infrastructure required for an application, Xenon exposes a programming model that allows users to express applications as sets of collaborating *nano-services*.

Nano-services can be thought as independent little agents with their own (i) invokable identity (a URI supporting RESTful HTTP operations [5]), (ii) corresponding policy (handlers for the aforementioned operations, allowing validation, propagation, collection *etc.*), and (iii) associated context (the latest version of the state).

Each such service can be configured at a particular point in the space defined by the CAP theorem [7], by toggling per-service capabilities (*e.g.*, `REPLICATION`, `ENFORCE_QUORUM`). For instance, in a single distributed application, friendships can be eagerly consistent, endorsements eventually consistent, and per-node stats not even replicated. For the discussion of transactions, we can think of individual Xenon nano-services as a location (and its associated state object).

The underlying framework takes care of issues such as scalability, fault-tolerance, consensus, state querying and introspection, instrumentation – all of these while allowing millions of operations per second on thousands of nodes. Most of Xenon's machinery is built to provide the needed guarantees (*e.g.*, consensus, consistent hashing, replication, gossip, grouping), while part of it focuses on satellite tooling (*e.g.*, complex queries, concurrency calculus, stats, persistence). All of these capabilities are implemented as services themselves (many exposed under the `/core` namespace), communicating using a programming model identical to the one user-generated services do. Unsurprisingly, transaction coordination is implemented in a similar way.

Although full exposition is beyond the scope of this paper, it is worth mentioning a couple of transaction-related features. First, Xenon's persistence and indexing capabilities are powered by a highly-performant, multi-version, append-only store (Lucene [8]). This makes certain operations on arbitrary state extremely powerful, while maintaining (and indexing) all versions of a particular state type – even identical instances! Clients can submit highly-expressive queries using a query service; such queries are broadcast and executed across different nodes within a group, and can be poked, consulted, and introspected at any point during their execution as any other state in Xenon. Transactions take full advantage of these capabilities.

Second, Xenon embraces both asynchrony and message passing, mechanisms that blur the distinction between local and remote execution and communication. Operations return instantly but kick off actions that themselves invoke callbacks when complete. The framework provides tooling to allow users to compose, sequence and parallelize sets of operations as well as isolate content through cloning. Asynchrony proved a double-edged sword when implementing transactions. On the positive side, transaction context flows naturally between different services. However, we had to take extensive care to make sure no progress was made until some invariant was satisfied (say, receipt of messages or resolution of potential conflicts).

A final note on Xenon's state and abstractions: Xenon has mainly two different types of services. *Stateful services* have been already described above and form Xenon's *modus operandi*. Another abstraction is *stateless services*, whose incoming operations often have some kind of side-effect. Such services usually invoke an action, propagate results to a number of other services, create and delete entities or are simply compositors, collecting and formatting data through other services. A particular example of stateless service are *service factories*, endpoints that accept POST HTTP requests and create service *instances* – that is, actual service occurrences with their own URI, policy and state, and the capabilities they should be offering. Service (*i.e.*, instance) URIs, if not provided during the initial POST to the factory, are chosen arbitrarily, offering the uniqueness guarantees of a universally unique identifier (UUID, v.4 [10]):

`http://x1.vmwa.re/service1/1d57adc3-58cc-4372-a567-0e...`

Everything until and including `service1` would be the factory, whereas the rest is the unique instance ID. Clients can now interact with this service using HTTP verbs (*i.e.*, PUT and PATCH to write, GET to read).

4. Transactions for Xenon

Transactions in Xenon are implemented as a core service. There is a transaction coordinator factory (or, transactions) responsible for launching transaction coordinator instances (or, coordinators). Semantically, instances are equivalent to *live* transaction IDs: they can be introspected and queried at any time, giving information about their operations, dependencies and whole sequence of captured events.

More concretely, a typical interaction is as follows. The transaction factory service is listening on `/core/transactions/`. A client can signal the beginning of a transaction by sending a POST request, optionally including any configuration parameters that affect the coordinator's behavior (see Sec. 5). As a result (i) a new coordinator is created, keeping track of the transaction context, and (ii) the URI for this coordinator is returned to the client (*e.g.*, `/core/transactions/tx_id`). Subsequently, the client can attach this ID in any operation that is part of this transaction (in HTTP terms this is simply a header, whereas in Java it is just augmenting the `operation` object). Note that more than one client can share such an ID, and invoke operations as parts of the same transaction; they can also interleave operations that are not part of the same transaction. Finally, the client chooses to complete the transaction by issuing a resolution request (*i.e.*, commit or abort) to the coordinator. Resolution requests are idempotent: the moment they hit the coordinator their effect is captured in a single, immutable state transition (Xenon internals provide such guarantees – details in the next subsection).

Technically, the coordinator is placed at an arbitrary node [9] (which may well be the same as the original node), but has its state replicated and eagerly consistent across many different nodes. The protocol does not in any way depend on a centralized factory service (after all, factories have no state and execute identically on every node); it could very well be that the client itself generates the UUID, and when sending an action to the first service, it is that very service that takes care of creating the coordinator on the fly (by routing deterministically based on the given UUID).

Operation Processing

Service implementors need not make any special arrangements regarding transactions. Transaction processing is done as part of the request processing pipeline for all services, without affecting the common path (in theory, the processing overhead is a couple of CPU cycles calculating whether or not to avoid the transaction processing branch, but even then, CPU branch prediction should accelerate this).

If the request is a state update (*i.e.*, PUT, PATCH), the new state is first amended to include the transaction ID (annotations described in Sec. 2), and a new version is forwarded to the persistent store creating a new version branch. If the request is a state fetch (*i.e.*, GET), the state returned is the latest version tagged with the same transaction ID or the latest un-annotated version. Technically, returning the appropriate state is either served via the cache, for the un-annotated version, or results in a query that is forwarded the core query service. In both cases, if the operation is successful, the service notifies the coordinator responsible for this transaction by sending a message (through a single, atomically-processed PATCH to its URI) with (i) information about the latest operation, (ii) pending operations (both reads and writes) from other transactions that reached the service prior to the current operation. In the unusual case that an operation was unsuccessful, the coordinator is notified accordingly, in order to fail the transaction. In any case, it keeps receiving multiple requests for state update from services taking part in a transaction, until it receives a request to resolve (*i.e.*, commit or abort).

Resolution Phase

A coordinator is responsible for (i) deciding whether it is safe to commit, and if it is, (ii) which other transactions it should follow or precede – guiding services to serialize state updates accordingly. Its read and write sets include service URIs that received read (*i.e.*, GET) or write (*i.e.*, PUT, PATCH, DELETE) operations; dependencies are captured via URIs to coordinators of transactions that were pending at the time when a particular operation reached a service. These coordinators will be contacted through a PATCH request when resolving whether a transaction can commit or not.

Exchange of information with potentially conflicting coordinators is done via parallel patches. Since responses are asynchronous, our implementation makes an interesting use of Xenon's mini concurrency calculus to capture dependencies before moving forward. After gathering the results, it uses a PATCH-to-self to change its state, reflecting and indexing state across all replicas. It then notifies all services to: (i) remove this transaction from their pending structures, (ii) in the case of a commit, remove the annotations from the state and update the latest state, effectively serializing one of the state branches over the existing timeline. In the cases where a transaction needs to follow another transaction, such metadata is sent across the services, so that services serialize operations accordingly.

Concurrency During Resolution

The resolution phase described above results in exchanging resolution request messages with remote coordinators, while handling resolution requests on behalf of other coordinators (potentially overlapping). Since request processing in Xenon is atomic and linearizable, this poses an interesting challenge: while a coordinator is processing a request (such as the commit request from a client), it cannot process requests from other coordinators. In particular, given a request to commit, a coordinator enters the resolution handler which itself will not complete (*i.e.*, send a resolution response to the client) before a decision of whether to commit or abort has been made. While this is happening, it might be receiving requests from other coordinators regarding conflicts, which cannot be addressed until it has finished processing the resolution request. This can practically cause a deadlock, since transactions might have circular dependencies.

To solve this, we use the following asynchronous trick. The coordinator exposes a simple a `/resolve` suffix, which is implemented as a simple, *stateless* utility service. When a client is about to commit (abort), instead of notifying the coordinator`/core/transactions/txid`, it sends a commit (abort) request to the `/core/transactions/txid/resolve` utility service associated with this coordinator. This, in turn, sends a resolution request to the stateful transaction coordinator to kick off the resolution protocol. This request returns immediately, so the coordinator can continue handling requests, but the utility service registers for notifications from the coordinator. At the same time, the utility service does not get back to client before receiving a notification. Upon completion, the coordinator's state transition (to `committed` or `aborted` depending on the result of the resolution) fires up a notification prompting the utility service to respond to the client.

Further Considerations

We now discuss some subtleties related to other parts of the protocol implementation on Xenon. Implemented naively, these would either result in side-channels that bypass transaction context or exhibit poor performance.

Queries and their expressiveness are one of Xenon's unique features: everything that has ever happened in Xenon is indexed and retrievable through queries. Issuing a query to the query service (also part of core), which takes care of translating and broadcasting to other nodes, should also be aware of transactional semantics. That is, if a query is issued within the context of a transaction, it should not match any state that is part of another pending transaction; conversely, if a query is issued outside the context of any transaction, it should not match any state pending commit. To do so, whenever a query service instance receives a query within a transaction, it appends a disjunctive clause that either matches this transaction ID or has the transaction ID empty. If it is not within a transaction it simply conjuncts a clause where the transaction ID field is empty. Note that it is still possible for a client to request only states pending commit (by including the appropriate clause).

Services that have side-effects (*e.g.*, create or delete a service, transition a task, or interact with an external resource), are in some sense trickier, since they don't fall into the read/write category: while they do not affect the complexity of the resolution protocol, they do need to execute only after it is unanimously agreed that the transaction has committed. Therefore action is withheld and usually taken upon receipt of the final commit message to the service. We are still discussing subtleties related to cases where the side-effect should be visible instantly within the transaction context (*e.g.*, tasks whose new state is queried in a follow-up operation within the same transaction).

Xenon provides a caching layer for service state that allows requests to avoid hitting the underlying store for the latest state version, with this latest version being propagated across different caches around the system. In the case of transactions though, annotated versions could potentially grow from a single value per service to a exponential explosion of cached state. We implement a bounded mapping from transactions to latest known state that maintains only n latest "branched" versions of the state, with an eviction policy that approaches Least Recently-Used (LRU). GET requests not within the mapping's working set result in a query, that eventually replaces a spot in the map. Upon commit, the new state fills the spot of the latest version for that state.

A final note regarding operations not part of a transaction. Xenon is all about managing trade-offs, so we wanted to allow non-transactional operations to have the option of "opting out" of the overhead of transactions—*i.e.*, execute without incurring any overhead associated with transactions, but possibly with limited guarantees. Elaborating on the example application of Sec. 3, operations reading both logs and endorsements could bypass transactions, since, showing a somewhat stale view of endorsements the very first seconds of loading a page does not normally affect a user's experience. Wrapping these operations within single-operation transactions (simply by attaching an arbitrary `txid` when sending the request) is isomorphic to reserving a spot on a globally serial timeline. One could easily implement a new service capability (*e.g.*, by, say, toggling `ENFORCE_SERIALIZABILITY` to true) at a per-service level that would upgrade all operations to serializable – with a possible cost in performance and abort ratio.

5. Protocol Enhancements

So far we described the core protocol, which we have already implemented and tested in Xenon. There is a number of directions we are planning to follow. One has to do with an extensive evaluation of the protocol.[1] Another direction has to do with adding various capabilities to the protocol, such as the ability for the services to

[1] A combination of factors did not allow such an analysis sooner: lack of a commonly accepted framework of benchmarks (recent efforts plan to address this [12]), precise characterization of Xenon production workloads and the cost of backporting other transactional implementations onto Xenon.

invoke referential integrity checks prior to commit, allow system-wide snapshot isolation, and selectively enable relaxed guarantees.

Integrity Checks

Services should be able to do final integrity checks with each other right before committing everything, to ensure that the state of the world is exactly as intended. Such checks might be complicated and include cycles of dependencies of arbitrary length. As a simple example, consider a transaction that takes care of deleting a pool of virtual machines, while at the same time releasing all the resources of the pool: a virtual machine cannot be deleted when it is part of an existing pool, resources cannot be released because they are being used by VMs, and pools cannot be collected before having all VMs deleted. Such validations cannot be made prior to the request for commit, nor after the notifying services to commit changes. They are also not the responsibility of the client, as a client might not even be fully aware of interdependencies of the operations it is invoking.

The idea here is to convert the final commit stage to a two phase commit where services run the checks they need before agreeing to commit. In particular, during the first phase (i) the coordinator notifies the services that it is about to commit, (ii) services issue requests or queries to validate state according to the invariants they need satisfied, and (iii) they reply to the coordinator. During the second phase, the coordinator decides to commit or abort based on the results, and notifies the client accordingly. The main challenge has to do with expressing the invariants for each service: it seems that different operations would require different invariants to be validated, and it is not entirely clear how we could make it easy for service designers to express them.

Snapshot-Isolation Enhancements

In read-mostly workloads, we should be able to increase the performance and throughput for long-lived read-only transactions and offer snapshot isolation—both out of the critical path, without any blocking or waiting, or hindering the performance of other operations in the system.

The current idea is to import insights from mutli-version concurrency control (MVCC). By using a logically-centralized sequencer that hands off carefully-selected sequence numbers, all transactions could be *a priori* aware of their position in the serial transaction timeline (if and only if the request to commit succeeds). This way, read-only transactions could be assigned a sequence number that would effectively read the latest state produced from any earlier transaction (*i.e.*, one with the largest, smaller than current, sequence number). Sequence numbers would not need to be perfect—jitter or skew would not affect the safety properties of the protocol.

The main challenge here is that reads and writes *must* go through the sequencer. Nailing the properties of such a sequencer (*i.e.*, enforce good numbers) is also under discussion. Incidentally, UUIDs generated from the coordinators take local time into account, which means that they could be used as a loosely coupled sequence generation. Our vision is that deployments within the same physical host or tight networks will have extremely good performance characteristics by turning on this option.

Transaction Options

An interesting research direction further in the future would be to allow the interplay between transactional protocols offering different guarantees. It is fairly trivial to have the initial `POST` to the transaction service specify the kinds of requirements the client is expecting – for instance, opt out of two-phase commit for integrity checks in order to shave off some overhead. But for more complex interactions the interplay is not as straightforward. For instance, allowing some transactions and their respective coordinators to offer *read-only* consistency guarantees [1] while the rest play with the full ACID rules would be interesting. And since some classes of services do not require the full ACID properties (and overhead), such an option would be in line with Xenon's configurable service model.

6. Conclusion

This paper describes a fully distributed transactional protocol built on Xenon, VMware's decentralized management framework, taking full advantage of several key features the framework provides. Our proposal shares many ideas with existing work: it takes full advantage of the system's underlying support for multi-version storage (*e.g.*, Percolator [11], TSO [3]) and efficient, expressive querying on arbitrary attributes (*e.g.*, BigTable [2], HBase [6]). However, it is different on a number of ways: it provides serializability (*e.g.*, *vs.* Percolator [11]), does not rely on augmented hardware capabilities (*e.g.*, *vs.* Spanner [4]), and does not rely on centralized coordination or conflict resolution (*e.g.*, *vs.* TSO [3]). While early in the full implementation, we are confident it has the potential to adapt well to different workloads (across many different levels of scale) and plan to evaluate soon.

References

[1] Peter Bailis, Alan Fekete, Joseph M Hellerstein, Ali Ghodsi, and Ion Stoica. "Scalable atomic visibility with ramp transactions". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 27–38.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), p. 4.

[3] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. "PNUTS: Yahoo!'s hosted data serving platform". In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1277–1288.

[4] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[5] Roy T Fielding and Richard N Taylor. "Principled design of the modern Web architecture". In: *ACM Transactions on Internet Technology (TOIT)* 2.2 (2002), pp. 115–150.

[6] Lars George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[7] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *ACM SIGACT News* 33.2 (2002), pp. 51–59.

[8] Erik Hatcher, Otis Gospodnetic, and Michael McCandless. *Lucene in action*. 2004.

[9] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM. 1997, pp. 654–663.

[10] Paul J Leach, Michael Mealling, and Rich Salz. *A Universally Unique Identifier (UUID) URN Namespace*. 2005.

[11] Daniel Peng and Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications." In: *OSDI*. Vol. 10. 2010, pp. 1–15.

[12] Naveen Kr Sharma, Brandon Holt, Irene Zhang, Dan R. K. Ports, and Marcos K Aguilera. "Transtorm: a Benchmark Suite for Transactional Key-Value Storage Systems". In: *SOSP 2015 Poster Session*. 2015.