UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

HoneyPot: A Caching Protocol for Causally Consistent Data Stores

Dissertation presented by
Ionel Munteanu

for obtaining the master's degree in
Software Engineering

Supervisor(s)
Prof. Peter Van Roy

Readers(s)
Prof. Kim Mens, Zhongmiao Li

Academic year 2016-2017

Acknowledgments

I would like to express my gratitude and appreciation to Professor Peter Van Roy for his guidance, for the understanding and for the confidence. I am grateful to Zhongmiao Li for his time and very useful remarks. His help and patience are greatly appreciated. I would like to thank my reader, Professor Kim Mens, for his time. Finally, I would like to thank Andreea without whom all of this could not be possible.

In memory of my dearest friend,

Ionuț

Abstract

Each day the content generated is exceedingly bigger and the quality of service much more demanding. To cope with this shortcoming, stronger consistency guarantees are bartered for higher availability and lower latency.

In this thesis we describe HoneyPot, a protocol that improves the overall performance of the target system at the expense of slight data staleness. Honeypot is a lightweight caching-layer protocol with transactional causal+ consistency semantics. This transactional model offers the strongest consistency guarantees under availability and convergence and insures causal ordering of transaction execution. As a result, HoneyPot improves throughput and reduces latency by adapting data locality regardless to the access pattern and workload. HoneyPot carries out the incoming transactions on locally replicated keys, on behalf of their owner, reducing network usage and preventing the formation of hot-spots in the cluster.

The results against the target system show an improvement of throughput and latency proportional to the cache-hit ratio and a small overhead in the worst case scenario.

1 INTRODUCTION

Today's large-scale online services strive to serve millions of clients simultaneously. Providing low latency is of crucial importance to online services: Amazon has reported that a 100ms latency penalty implies a 1% sales loss [15]. As such, a prevalent trend for recent distributed data store, which is the backbone of modern online services [16] [17] [18] [19] [20], is to opt weak consistency criteria for high availability and low latency. Among these proposed solutions, causal consistency appears to a particularly appealing consistency guarantee: while providing low latency and high availability, it is known to be the strongest guarantee that can be achieved without sacrificing availability, precluding a large spectrum of anomalies [21].

While classical proposals to ensure causal consistency typically assume each replica to fully replicate all data, recent causally-consistent data stores are designed to be highly-scalable to accommodate to today's large-scale applications: a data store typically consists of several replicas deployed in geographical scale and within each replica, data is partitioned among multiple nodes. Nevertheless, data partitioning raises several concerns: if nodes have low data locality, it has to frequently access data from other nodes, which result in high latency; if workload presents skewed access pattern (cite some paper from E-Store), a common scenario for real-world workloads, certain nodes of a data store may be highly loaded while others are sitting idle, providing non-optimal throughput.

Caching is a well-known mechanism to improve data locality and alleviate load imbalance and we seek to exploit this technique to mitigate the above problems of partitioned causally-consistent data stores. While classical caching techniques can suffer from excessive cache invalidation with high write rate, applying caching to causal consistency avoids this pitfall. In a nutshell, since causal consistency allows concurrent updates to be applied to the same data item without synchronization and reads are not required to reflect the most recent updates of a data item, cache invalidation is not needed. Nevertheless, this brings several challenges. First, the drawback of allowing concurrent updates without synchronization is that, only one of concurrent updates to the same data item can be preserved, causing clients to lose their updates. Still, a more fundamental problem is that caching allows each node to have different versions of data items, which makes it challenging to enforce causal consistency.

In this thesis we propose HoneyPot, a novel caching mechanism for partitioned causally-consistent data stores. HoneyPot provides a seamless caching layer, which can cache both updates and reads to data items, that aims at increasing data locality (thus reducing latency) and mitigates load imbalance due to hotspots (thus improving throughput). To overcome the lost update anomaly aggravated by caching, we exploit Conflict-free Replicated Data Types, which are conflict-free by design and could safely merge concurrent updates. In order to enforce causal consistency, HoneyPot prunes cached data in a causally-consistent fashion, and employs a lightweight mechanism to detect potential causality violation while reading data items.

We performed extensive evaluation of HoneyPot against a partitioned causally-consistent data store. The results show that HoneyPot improves throughput up to 3 times, reduces latency by half.

1.1 CONTRIBUTIONS

In this thesis we strive to address the problem of throughput and latency optimizations for distributed key-value stores. To this end, we created a light-weight protocol that provides better performance under strongest guarantees, given the context.

HoneyPot is a transactional causal+ consistent caching tier that boosts the overall efficiency of the underlying target system with a smaller demand on network resources and acts as an internal load balancing.

We test the target system against its HoneyPot improved version and find that the results show an overall positive impact.

1.2 STRUCTURE

In the second section, we describe some of the technologies that are similar or have a source of inspiration for HoneyPot. We provide a short overview after which we illustrate with some implementation examples.

The third section is dedicated to HoneyPot, the subject of the current thesis. We commence with a high-level glance over the architecture in the Overview subsection. We then take a brief look at Antidote DB, the target platform after which we formulate the problem and argue that the problem we are solving is not trivial. In the HoneyPot Protocol subsections, we provide some intuition on the protocol and its underlying logic while under Correctness sketch the proof.

Under the Performance chapter we test the protocol under various conditions and measure the performance, depicting the performance gains, low resource consumption and internal load balancing.

We shortly conclude in the fourth chapter and provide some ideas for further development.

2 BACKGROUND

To understand how Honeypot works we first need to overview the underlying key concepts. In this chapter, we will briefly introduce them and some of their implementations relevant to understanding how our protocol works, as well as some technologies in which the formers have a similar purpose.

2.1 TRANSACTIONAL CAUSAL+ CONSISTENCY

Causal consistency is an eventually-consistent model providing the best trade-off between strong guarantees and availability [21] [29]. Causal consistency, or causal order, is defined as a dependency between operations which occurs in three different circumstances. The first case, called *execution thread*, consecutive writes coming from the same client are causally ordered, with each write causally depending on the previous. The *get from* orders reads with the update transactions they observe. The third one is *transitivity* where if an operation is causally dependent on a second, and the second on the third then the former is also causally dependent on the latter.

The original definition for causal consistency considers single operations. As transaction is a useful programming abstraction, recent works have proposed causally-consistent protocols with transactional support, namely Transactional Causal Consistency. A transaction that commits atomically and isolated creates a valid snapshot. If all the updates are applied in causal order then the result is a causally consistent snapshot. In consistency model is Transactional Causal+ Consistent if each transaction sees a valid snapshot and its own updates and if every snapshot is causally consistent.

For Transactional Causal Consistency, the semantics are enhanced with a forth rule called *transactional closure* where if one operation from a transaction causally depends on another operation from another transaction then all of the operations from the first transaction depends on all the operations on the second.

As mentioned earlier causal consistency is an eventual consistency level, meaning that at some point in time all the replicas will eventually converge. If a model provides a conflict resolution protocol that handles the conflicts in the same way then the model satisfies *convergent conflict handling* [35] and the transactional model becomes Causal+ Consistent.

2.1.1 SwiftCloud

SwiftCloud is a distributed storage technology that replicates data fully between data centers and partially in caches located client side, under Transactional Causal+ Consistency semantics, to improve performance. SwiftCloud improves the availability and mitigates temporary connectivity disruptions by allowing the user to cache locally information, making it especially useful for mobile clients.

The core is the set of DCs which fully replicate the data set and the clients that cache a subset of it on their side. Subsets of data types will reside here for a period of type, following a LRU strategy or unless explicitly specified. The data set is updated either by committing local transactions, called internal updates or as a result of synchronization with the DC.

To guarantee causal consistency, SwiftCloud implements different semantics for non-mergeable (or non-commutative) transactions and mergeable (commutative) transactions. Non-commutative transactions are order-dependent and execute server-side, under a two-phase commit protocol which ensures that at most one concurrent operation commits. Mergeable transactions (e.g. reads or CRDT and C-set writes) are carried out locally, on the client machine.

Replication under causal consistency is managed at two levels: between datacenters, over a fully replicated dataset, and between a datacenter and a client, over partially replicated datasets.

For the first case, the datacenter keeps track of every entry's causal predecessors in set called dependency set. When synchronizing, datacenters exchange logs containing updates. Upon receiving a log, the datacenter checks for causal gaps in the dependency sets. If none exist, all the updates are applied immediately. Conversely, a datacenter will wait until the entire dependency set is received. To avoid availability penalties or causal inconsistencies, operations performed during the synchronization phase will observe older but consistent snapshots (i.e. "read in the past" approach [9]).

On the client side, a SwiftCloud service is in charge with DC connectivity and data synchronization. Updates execute on a local copy and merged in the cache when finished. The scout will eventually synchronize the latest updates to the DC. In case of a cache miss, the DC will provide causally consistent snapshots.

2.1.2 COPS

COPS (Cluster of Order-Preserving Servers) is a wide-area causal+ consistent key-value store. COPS is the first system to formally define and use Causal + Consistency, adding a fourth rule, *convergent conflict handling.,* to causal consistency's original three (i.e. *execution thread, gets from* and *causal transitivity*).

Convergent conflict handling assumes handling all conflicting writes return the same results, at every replica. Normally, a commutative and associative function is used to handle the updates in the order in which they arrive at each partition and having the results converge.

The keyspace is distributed among the nodes using consistent hashing. Every key has a *primary* node in a datacenter and is replicated for fault tolerance, using chain replication. *Equivalent* nodes are all the nodes in other datacenters which are primary for a given key.

Locally, reads and writes are linearized over the replication chain. Reads are served from the tail of the chain while writes are always placed at the head and propagated downstream. After a write completes locally a new version is created, identified by a version number composed of a Lamport clock and unique node identifier. The new version is replicated across the chain and becomes available once it reaches the

tail. The write is placed in a queue and disseminated asynchronously to all the equivalent nodes in the remaining datacenters where it waits to be applied until all the causal dependencies are provided. COPS keep track if causality be means of version number as metadata.

COPS-GT is semantically enriched variant of COPS which allows causally-consistent multi-key transactions. COPS-GT uses a graph in which to store its causal dependencies. For writes, the logic is similar to COPS. For reads, the algorithm undergoes two rounds. First round retrieves all the keys in the parameter list together with their dependencies. Although all the local versions are consistent by themselves they may not be consistent with one another. The protocol cross-matches the dependency of each key and check for causal gaps. If any issues consistency violation is detected, the keys of proper version are fetched in a second round.

2.2 CONFLICT FREE REPLICATED DATA TYPES (CRDTS)

All large scale distributed data systems are subjected to Brewer's CAP theorem [22]. Strong consistency conflicts with the "availability" and "partitioning tolerance" properties, having a negative impact on scalability and performance due to its global characteristics. For overcoming this issue, a popular approach is weakening the consistency guarantees. Opposed to techniques such as serialization, seen in the strong consistency model, eventual consistency allows concurrent updates on data types, even in the case of temporary network partitioning. Furthermore, once the connection is reestablished, the replicas need not converge immediately. The ability to avoid consensus [10] circumvents an outburst of messages sent over the network at once. Unfortunately, updating regular data types concurrently has its trade-offs as well. An object being updated in parallel will require conflict resolution which in some cases may need consensus and rollback or in other cases it may use nondeterministic heuristics like last-write-wins.

The Conflict-free Replicated Data Types (CRDTs) allows concurrent updates while providing automatic conflict resolution and strong eventual consistency. In a conventional system, updating replicas concurrently results in an unsolvable conflict forcing the system to abort and rollback. Strong eventual consistency avoids this issue by strengthening the properties of the eventual consistency model. An object is said to be strongly eventual consistent if it is eventually consistent and if all objects that delivered the same states have the same update.

The CRDT family is composed of two members. The first one is the state-based CRDT, or convergent replicated data types (CvRDTs), as merging involves the replica's local state. The CvRDTs are modeled as an algebraic data type similar to a set, with partially ordered elements and a *least upper bound* (LUB) operator. The LUB is a commutative, associative and idempotent function that, provided the set and ordering function as parameters, outputs the smallest number, according to the pre-established ordering function, bigger than the maximum values of each set (i.e. supremum). Applying this operator on any two elements of the set yields a non-null result. The LUB operator is used by the state-based object's merge method to converge towards the LUB of the most recent values.

When converging, the states of the CvRDTs are compared between themselves. In case of a mismatch, the LUB operator is used to resolve the conflicts, generating a third state superset of both initial states.

State-based CRDTs do not require strong guarantees of its underlying communication channel, since the merge operation is commutative and idempotent. Messages may be lost, duplicated or out of order, as long as all the replicas receive all the updates, they will eventually converge.

The second type is the Commutative Replicated Data Types (CmRDTs), also called operation-based (op-based) CRDTs. In the case of the op-based CRDTs on the same replica, the updates need to be delivered in causal order by a reliable channel, whereas for *concurrent* updates, the operations applied need to *commute.* If a network partitions, all the connected replicas deliver each other's updates. Since delivery order is never stricter than causal delivery all the replicas will eventually converge.

State-based objects are easy to reason about since all information in captured in the state and require weaker channel guarantees. However, such CRDTs may grow large, making it much more complicated to be sent across the network. Recent work has proposed to only send the delta-states (i.e., states that were mutated) to alleviate the problem [43].

Op-based are more complex since it requires reasoning about the causal ordering of the operations. They have more expressive power with a smaller payload, but require stronger network guarantees.
The relation between the two types of CRDTs makes it possible to emulate one from the other, in any situation. Remember that a CvRDT is a monotonic semi-lattice with commutative properties, while the CmRDT are a partially ordered set of operations. Since the queries do not affect in any way the states, the only mapping needs to be done between the merge and update functions. In the case of a CmRDT emulated by a CvRDT, additional sets are used to keep track of concurrent updates, where no order is needed, as well as a set containing all the delivered updates, to avoid applying the same operation multiple times. The second case is achieved by making the op-based update method to compute a state-based update and perform merge downstream.

2.3 CLOCK-SI

In the following text, we describe Clock-SI, a transactional protocol that leverages the use of decentralized physical clocks to provide Snapshot Isolation. Its use of decentralized clocks has been integrated into Antidote, which is HoneyPot's target system, and has also inspired the design of our protocol.

Clock-SI uses the two-phase commit [24] where each involved partition checks if the transaction conflicts with any concurrent transaction, and proposes either *commit* or *abort* to the transaction coordinator. Conversely, instead of relying on a central authority to assign timestamps as in conventional protocols, Clock-SI dos not need a centralized clock authority instead it relies on the partition's loosely synchronized clocks for the assignation of snapshot and transaction commit timestamps. The implication of such an approach is eliminating the single point of failure and avoid having a single timestamp authority that may become bottleneck. Moreover, the communication latency is reduced by one round-trip for obtaining a timestamp in read operation (just the snapshot timestamp) and two in the case of an update transaction (for the snapshot and commit timestamps).

The targeted system is a multi-version key-value store having a partitioned dataset, with one partition per server. The servers' hardware clocks can be synchronized by clock synchronization protocols like Network Time Protocol (NTP), but the protocol does not assume the differences of clocks are bounded. Transactions consists of a set of simple operations (i.e. get, put, delete) which can be applied sequentially on any collection of data objects from any partition.

To lower the chances of such a scenario ever occurring, the timestamp of the transaction can be intentionally diminished, improving availability at the cost of possible data staleness.

2.3.1 Read Protocol

In Clock-SI, a read transaction will always commit. A performance penalty may be experienced due to clock skew or pending transactions. The former case occurs when the request's timestamp is greater than the remote server's. In this situation, the requests stalls until the targeted node's clock catches up and the snapshot becomes available. The later example the request needs to wait for the outcome of the pending transaction. The read will have to wait until the update finishes as there is no way of foreseeing beforehand the outcome of the updated (i.e. transaction succeeds or aborts).

The reading protocol insures a timestamp for a new transaction. The timestamp is equivalent to the value of the hardware clock at the originating partition. The protocol verifies if the read needs to be delayed due to pending transactions with a smaller committing timestamp. If case may be, the read is postponed so that the snapshot can contain the last update (to conform to the snapshot consistency properties). There are three cases, first in which the transaction is committing, the second in which the transaction waits for all the responses in the case of a two-phase commit protocol and the last case as a result of a clock skew. In any situation, delaying a read operation does not introduce deadlocks: an operation waits only for a finite time, until a commit operation completes, or a clock catches up.

2.3.2 Commit Protocol

Transactions that have updated data items undergo certification check, i.e., commit protocol, to decide if they should be committed or not. A certification check verifies if there are other concurrent transactions trying to update the current write set. If the check passes the transaction's state changes from active to committing. The snapshot timestamp is assigned based on the physical clock's readings followed by persisting the changes to local storage. At this point, the status will be changed to "*committed"* and the effects can be seen by all future[1] transactions.

If the write-set targets multiple partitions, a coordinator oversees a two-phase commit protocol, insuring either an atomic commit at every involved partition or the transaction's abort. The coordinator runs at the originating partition and issues a certification check which is carried out locally at every affected partition A partition votes 'aborted' if it locally detects any concurrent conflicting transactions; otherwise, it writes the prepared record to local stable storage, proposes its current physical clock as the transaction's prepare timestamp and then replies 'prepared' to the transaction coordinator. The status is changed accordingly, from 'active' to '*prepared*'. The coordinator will receive prepared messages from each partition containing the *'prepared'* timestamp and will choose the maximum as the transaction's commit timestamp. Choosing the biggest timestamp enforces isolation as any future transaction will see all the updates inside its snapshot at the same time.

[1] Transactions having the timestamp bigger than this transaction's committing timestamp.

2.4 CACHING

Caching is technique that improves access time to highly requested resources found throughout all levels of abstractions, from low-level embedded systems to large cloud platforms.

Unlike HoneyPot, modern in-memory distribution caching solutions are design to provide either high availability or strong consistency. Under high availability, client code is required to contribute with a layer of semantics while replicas under strong guarantees are locked throughout the duration of a transaction, penalizing performance.

Memcached is a popular in-memory key-value store used by companies such as Facebook [7] to get better performance and lower latency by using it as a distributed cache tier. Memcached provides a simple CRUD API without any additional semantics for transactions, therefor Facebook has implemented its own logic, tailored to their specific workloads (read heavy), for data retrieval or consistency management.

Redis [37] is a similar solution providing slightly more advanced functionalities under eventual consistency. Unlike Memcached, Redis has a built-in disk persistence, supports replication and more advanced data structures. Replication in Redis is mainly used for high-availability and follows a simple master-slave protocol, with each slave being an exact replica of its master [38]. The default replication setting between a master and its slaves is asynchronous, sending stream of updates down the slave chain. Redis also provides a *Wait* procedure that is able to acknowledge that a specific datum has been replicated over some number of Redis instances.

3 HONEYPOT

HoneyPot is a fully distributed caching protocol that optimizes data locality by temporarily storing snapshots in memory and guarantees Transactional Causally+ Consistency semantics, the strongest available and convergent model [21] [36]. The protocol acts as a proxy, replying to read queries and performing update transactions on behalf of the key owner. By reducing the need to access the requested data via a costlier path, the overall performance of the target system is improved.

We first provide a high-level description of the two-tier architecture of HoneyPot. We continue with target platform and its underlying protocol, after which we highlight the challenges we faced by presenting a naive caching mechanism that improves performance but does not enforce causal consistency. Then we present our protocol in detail and finally give a sketched correctness proof.

3.1 OVERVIEW

In this section, we present a high-level overview of the HoneyPot's constituent abstractions.

HoneyPot encapsulated the logic using two segregated components present on each node in the cluster. These two abstractions interact via a data bus provided by the target system.

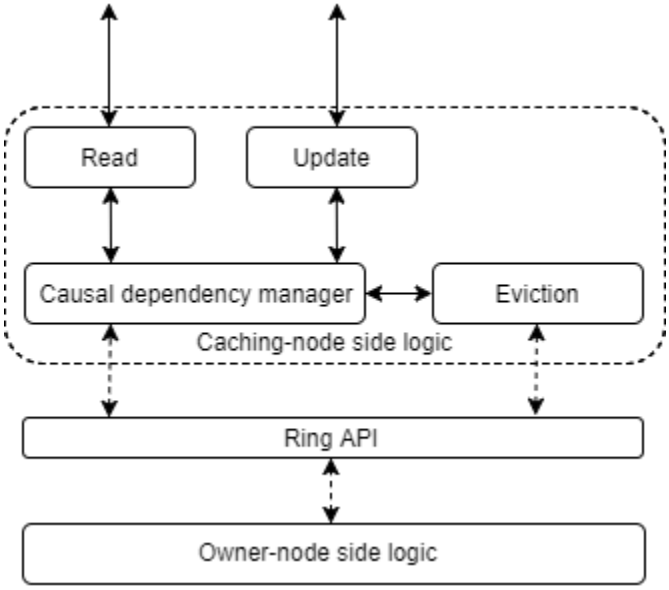

Fig. 2: Architecture of a Honeypot node

The caching-node logic is the actual caching layer, the entry-point of the system. By exposing a read and write functionality the cache provides a simple API which is used by the clients to interact with internal data. The API rests on a much more complex reasoning, represented abstractly as *Causal dependency*

manager and *Eviction* components. This underlying logic is the core of HoneyPot's guarantees. The *Causal dependency manager* component provides causally consistent snapshots to the *Read* component and creates causal ordering for the mutating operations received from *Update*. *Eviction* oversees that no key is kept in the cache longer than it should or that the in-memory table does not grow over the preconfigured threshold. When either one of these limits is exceeded, it ensures that all the causally related objects are properly evicted and merged at the owner.

The Ring API is the logical ring's data bus provided by the target system. It is a totally transparent layer used for message exchanges between nodes. Via this abstraction the two composing sides of HoneyPot are able to communicate using message-passing.

The remaining abstraction is logically placed at the owner-node side. It contains internal functionality used by the caching layer to fill causal gaps in the read-set or to atomically merge evicted writes, forming new causally consistent snapshots.

HoneyPot provides a simple API allowing bulk reading and writing through which a client is able to retrieve or update multiple keys at once:

- Read([key_1, key_2, …])
- Write([{key_1, [op_1, op_2, op_3]}, {key_2, [op_1, op_2, op_3, …]}, …])

3.2 ANTIDOTE DB – THE TARGET PLATFORM

Antidote is an experimental geo-replicated CRDT store platform developed on top of the Riak Core [30]. The keyspace is fully replicated across DCs, using static placement schemas where non-overlapping partitions are placed inside virtual nodes (v-nodes). Having the replication factor inside the cluster set to 1 (no replication) makes Antidote a perfect candidate for the caching protocol.

We briefly describe Cure, Antidote's underlying protocol. Cure is a transactional causal consistent protocol for highly available transactions [33]. Cure provides a friendly API which allows client code to issue reads and writes under the same transaction, called an interactive transaction.

Transactional causality is guaranteed by following two simple properties: (i) a transaction reads from a consistent snapshot (no snapshot will contain causal gaps, all causal dependencies are made visible) and (ii) transactions commit atomically (starting from a causally consistent snapshot, given that either all updates commit or none does, regardless of the outcome the resulting snapshot will still be consistent). For state confluence, where other systems adopt the last-write-wins strategy [31] [39] [38] [40], Cure relies on the rich semantics of CRDTs for conflict resolution and compensate for concurrency anomalies.

The protocol assumes that all partitions are fitted with physical clocks, loosely synchronized by a network protocol such as NTP. Clock skew between partitions does not undermine the correctness of the protocol, but it negatively impacts performance. Cure uses the clock to tag committed transactions with

monotonically increasing timestamps, as well as making snapshots visible in accordance with causality. The timestamps are placed in a vector clock in which each datacenter has one entry. Transactions originating from the local DC satisfy causality so its effects can be made visible right away. In contrast, transactions that originate from other DCs need to synchronize *globally stable snapshot* (*GSS*), the set of versions available at every local partition. Once all potential causal dependencies are made visible, i.e. the globally stable snapshot advances beyond the incoming transactions' commit timestamp, the transactions may progress. Computing the GSS and delaying foreign-originating transaction are part of GentleRain [42], Cure's inter-DC stabilization protocol.

The system's entry point provides an API that allows the user to register read and write operations, in interactive mode. As an additional parameter, a user may register a snapshot timestamp corresponding to the snapshot's commit timestamp (i.e. its version). This parameter will default to local time if not explicitly specified. The request is then forwarded to a transaction coordinator. It splits the keys into a read-set and a write-set and then sends appropriate snapshot request to the keys' primary locations.

When a partition receives a request (either read or prepare) of a transaction, it first compares the snapshot time of this request with its local physical clock. If it is greater than local clock or there is a pending write then the operation is postponed until the requested snapshot becomes available locally, that is until the local clock catches up with the timestamp. Cure stores multiple versions of the same key along with the metadata describing the causal dependencies. Periodically, the oldest versions are garbage-collected. The owner looks in its store for the version of the key identified by the biggest timestamp smaller or equal to the timestamp from the message. With all the keys retrieved, the coordinator applies the writes to any keys from the read-set overlapping the write-set and replies to the read request to the user. The updates are forwarded to the keys' owners. Committing implies a two-phase commit protocol in which the first phase all participant nodes propose as prepare timestamp equal to their local clock and in the commit atomically with the highest of the proposed timestamps in the first step.

3.3 PROBLEM FORMULATION

Workflows require a small fraction of the entire keyspace and key-value stores with static data placement schemas cannot adapt to different traffic patterns. This problem can be easily addressed by introducing a caching tier which can adapt to workload patterns in real-time, temporarily storing only the subset of keys which pose interest.

While solving a problem, caching introduces another: causally consistent data stores face the possibility of missing updates when partially replicating data [9]. The problem becomes even more cumbersome when multiple versions of the same key are replicated dynamically across multiple nodes. Data which allow a client-side cache [9] fall back on fully replicated datacenters. The client cache sees the data center as a single serialization point to which it delegates the task of supplying the causally consistent version of the missing keys.

As a straw man's approach, we can naively allow data items to be cached in any node they are requested, and allow reading and writing to cached data items without restriction. However, during the time spent in a cache, the object's state will be altered only by the local subset of writes. Since other transactions occurring at other nodes may be missing, no caching node may claim it has the latest version of the object.

Assuming three nodes, A, B and C as seen in the figure below, with node C being the owner of key 'c'. At timeframes 10 and 22, nodes A and B cache key 'c' with the value10. Now, in the system there are three identical replicas of key 'c'. After a while, three writes arrive at nodes B and C to update the key. Now the system has 3 replicas with different values. From the chronological standpoint, the ordered sequence of transactions, from a certain point in time onwards, are no longer the same. Their causal histories have diverged.

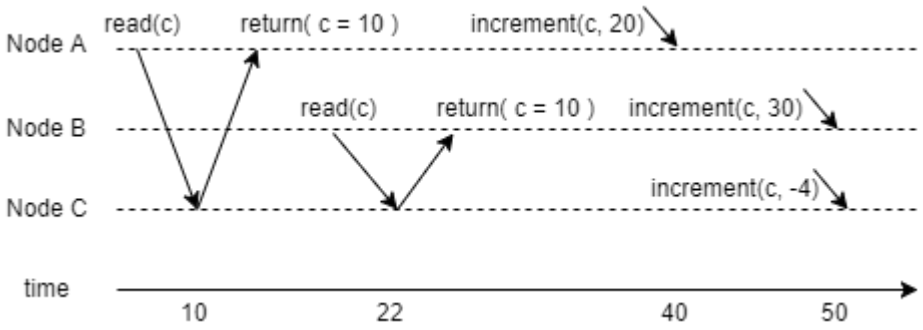

Fig. 3: History divergence for key x cached by 2 nodes concurrently

Keeping track of every snapshot's location and dependencies without coordination is hard. The protocol must take into consideration all the causal dependencies of existing snapshots and insure causal consistency of newly created ones.

3.4 HONEYPOT PROTOCOL

In this section, we describe the HoneyPot protocol and provide some intuition about the underlying logic used to provide the guarantees.

3.4.1 HoneyPot Read and Write

In an environment where multiple versions of the same object coexist concurrently, providing causally consistent snapshots is not trivial, especially while trying to avoid fat metadata or coordination due to performance considerations. HoneyPot provides the user with a simple API which allows him to manipulate data in a transactional causally+ consistent way.

HoneyPot is able to boost performance by temporarily storing in its memory table keys for a predetermined amount of time. This period of time of time is called a *time lease.* In the literature, a lease is defined as a contract between a datum's storage site and its temporary holder, granting the latter rights over the object for a limited period of time. More formally, a lease consists of a pair *(timestamp; duration)* where the timestamp represents the starting time of the lease and the duration its temporary span.

To provide causal consistency, every snapshot must be causally consistent and every transaction must see its own updates. Reading from the same cache provides session guarantees (i.e. read your writes) for free. However, when creating causally consistent snapshot, the protocol needs to take into consideration the recent versions of the stored object. In this context, "recent history" refers to all the versions at the owner with commit timestamps no older than "lease", relative to the current local time. Older than "lease" versions should not exist since all the timers would have expired and the updates have been consolidated as monotonic version of the same object, owner side and all the borrowed snapshots will contain them.

In this example, we assume nodes A owning 'a', B owning 'b', C owning 'c' and D owning 'd'. Initially keys 'a', 'b', 'c' and 'd' have version 0. Each update creates new versions of the key identified by the commit timestamp. The first update, T_1, creates new versions for keys 'a' and 'd' with timestamp 15. After T_2, key 'b' and 'd' will both have a newer version. The read at node C will cache keys 'a' and 'b' with their latest versions available at the owner, respectively 15 and 20.

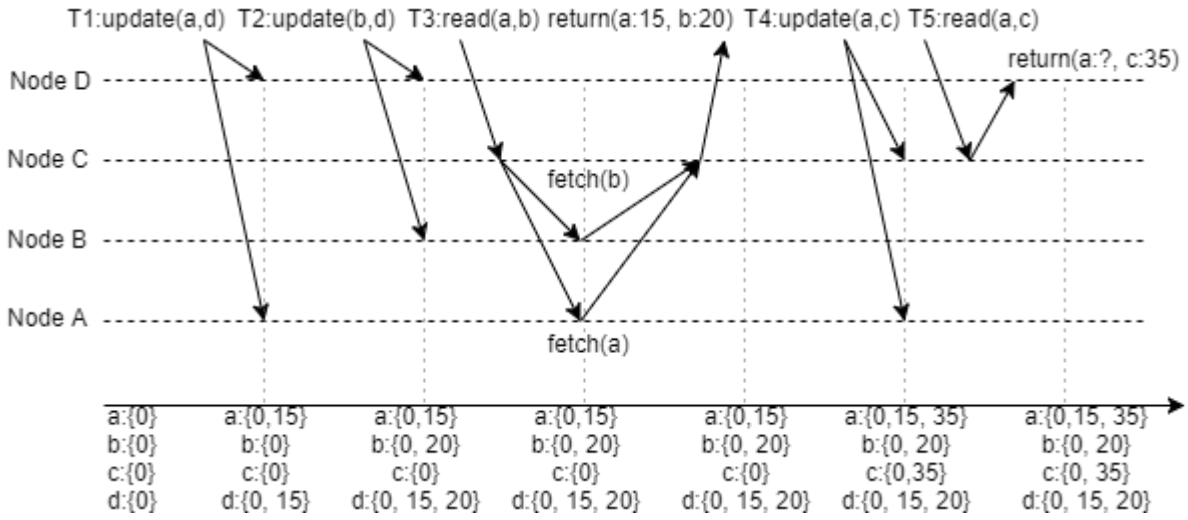

Fig. 4: Selecting the right version

The third update, T_4, commits at nodes C and A, creating newer versions identified by timestamp 35. Now, the latest versions of key 'a' has diverged from the versions cached at node C. A second read at node C is interested in keys 'a' and 'c'. Key 'a' is already cached with version 15.

The problem shown in the above example is that when reading from cache, a transaction does not know if the version in the cache is still up to date. For instance, while T_5 is reading the cached *'a'*, it does not immediately know if there is a newer version of *'a'* (which is *'a_{35}'* in this case). While T_5 could certainly query Node A to verify if *'a'* is still consistent, performing this type of check too frequently would diminish

the benefit of caching. To this end, HoneyPot employs a light-weight mechanism that helps to identify whether it is 'safe' to read a local version, or if it is not safe such that it is necessary to query the owner node. This is achieved by means of embedding metadata for each version to specify the period of logical duration that the current version is visible. In case a transaction is trying to read a version out of its visible duration, the read may violate causality and has to be forwarded to the owner of this version.

We define the visibility limit of a key's version as the biggest timestamp smaller than the one identifying the next consecutive version of that key. A visibility limit of a key's version is the continuous sequence of timestamps between a version's commit timestamp and its visibility limit.

> *Assuming key 'a' with versions a_0, a_{56} and a_{140}, created by three writes committed at respective times, and a timestamp resolution of 1, a_1's visibility limit is 55 with visibility range [0, 55], a_{56}'s visibility limit is 139 with the visibility range [55, 139] and a_{140}'s visibility limit is not yet defined and instead, the owner node's clock time at which the key was read/cached will be used as a limit. Each version will have the same state thought its visibility range.*

Honeypot uses the visibility ranges of key versions to prune out already cached but inconsistent versions of a read-set and to fetch consistent ones. For a set of keys to be consistent, each two keys need to be consistent relative to one another. For two keys to be consistent, the visibility limit of the key with the smallest version needs to be higher than the other keys' version. In other words, a set of keys are consistent with each other of their visibility ranges overlaps.

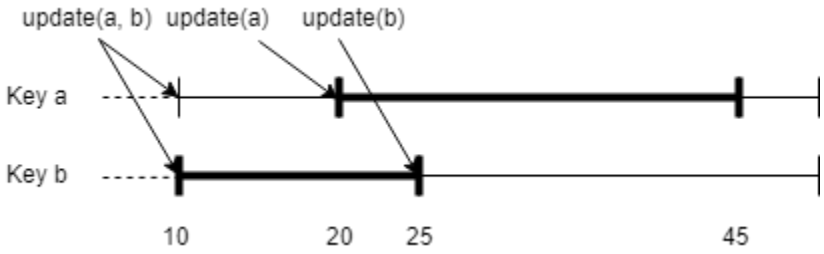

Fig. 5: Overlapping visibility ranges, emphasizing the cached versions.

As will be discussed in the handover section, the version of any given key reflects a change in state generated by an atomically committed update operation at the time specified by the commit timestamp. All the keys updated by the same write will have the same version and two updates never commit on the same key with the same timestamp. If the visibility limit of one key is greater than the commit time of the other then all the transactions that updated both keys are reflected in their versions (Fig. 5). Conversely, if the visibility ranges do not overlap then there is a chance that in the gap at least one transaction updating both key could have committed and appear only in the key with the highest version (Fig.6) leading to a causal gap and finally to an inconsistent snapshot.

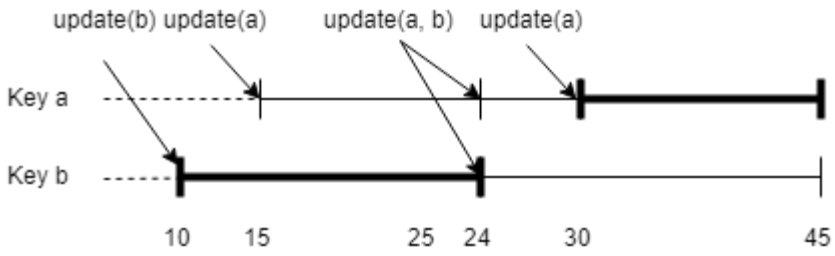

Fig. 6: Non-overlapping visibility ranges, emphasizing the cached versions.

During a read, HoneyPot computes missing keys as well as the set of already cached (lines 3 and 4 of the *Read* specification).

The next step is to compute a meaningful read timestamp, and then fetch both missing keys and cached keys that may be inconsistent with this timestamp. If no key from the parameter list is cached the local clock is provided as timestamp so that the protocol can fetch the most recent versions of the keys (line 6 through 10).

If, however, some keys are cached, the protocol start verifying key consistency and filtering the keys whose visibility limit does not exceed the maximum timestamp (line 12). For each of these keys, *FetchKey* is called together with the snapshot timestamp (line 13).

At the owner, *FetchKey* waits for pending updates to finish or for the local clock to catch up the timestamp received as parameter, i.e., waiting until the required snapshot is available at that partition (lines 55 and 56 of the *FetchKey* specification). As soon as these two conditions are satisfied, the function checks in its logs for the largest version of the key smaller or equal to the provided causal clock (line 57) and returns it together with the visibility limit. The visibility limit corresponds to the next version of the key, if there is one, or the local time otherwise (lines 58 through 61). This local timestamp is used as the next version of the current version does not exist yet, and will commit with a timestamp larger than the local clock of that node. Thus, the current local time serves as a conservative lower bound on the visibility limit of this version.

The new versions received back at the caching node will override the existing ones together with the new visibility ranges (line 14) and all the locally cached updates will be applied over each key (line 16). The operations are kept separately from the actual snapshot value to avoid handover. If the updates would have been handed back to the owner, new versions would have been created with a commit timestamp greater than any version previously cached, and implicitly than TS_{max}, making them temporarily unavailable, creating large latency. Moreover, the handover triggers a two-phase commit protocol, which is very expensive as we will see in the next section.

For cache-misses, the logic is similar, except that there are no cached operations to be reapplied (lines 19 through 22).

Once the consistent snapshot is created, all the keys are linked together (line 23) and a timer is started (line 24) (more details in the next section).

HoneyPot Read specification

```
1: Read(ListofKey keys)
2:      TS_max = 0
3:      CachedKeys = all key in keys that are cached locally
4:      RemoteKeys = Keys - CachedKeys
5:      ReadSet= []
6:      If CachedKeys is empty
7 :             MaxTs = local_clock()
8:      Else
9 :             For key in CachedKeys
10:                     TS_max = max(TS_max, key.ts)
11:     For key in CachedKeys
12:             If key.vl < MaxTS
13:                     Value, visibility_limit = FetchKey(key, TS_max)
14:                     Add {key, visibility_limit .ts, visibility_limit .vl} to cache
15:                     Apply cached op list
16:                     ReadSet.add(value)
17:             Else
18:                     ReadSet.add(value)
19:     For key in RemoteKeys
20:             Value, ts, vl = FetchKey(key, TS_max)
21:             Add {key, ts, visible_limit=vl} to cache
22:             ReadSet.add(value)
23:     Link(ListOfKeys)
24:     StartTimer(ListOfKeys)
25:     Return ReadSet
```

Spec. 1 Read

HoneyPot FetchKey specification

```
54: FetchKey(key, st)
55:     Wait until st <= local_clock()
56:     Wait until key has no prepared record with pt <= st
57:     Value, ts = max version of key with ts <= st
58:     If ts is the max version
59:             Reply Value, ts, current clock
60:     Else
61:             Reply Value, ts, ts of the next version
```

Spec. 2 FetchKey

The *Write* logic is somewhat simpler. As mentioned earlier, updates are stored locally regardless of the fact that the key is actually cached or not. If the key is cached, the write is applied to the object and the operation is appended in an ordered list, to maintain *thread of execution,* i.e. the order in which updates

coming from the same client are applied form causal ordering and needs to be respected (line 34 and 35 of the *Write* specification). If the key is not yet cached, a new entry is created, having the version flagged with -1 (line 37) and the operations are stored in order. The next step is to causally link the keys and start the timer. When the timer expires, all operations for the causally linked keys are sent back to their owners to be merged. If a read will concern a key marked with snapshot version -1 (line 37), the *Read* algorithm will treat this key as a normal case of inconsistency (line 12 of the *Read* specification), it will fetch the causally consistent version and apply the all the cached operations before serving it to the user.

HoneyPot Write specification

```
31: Write(ListofUpdates updates)
32: For key, op in updates
33:     If key is cached locally
34:             key.op_list.append(value/op)
35:             key.snapshot.apply(op)
36:     Else
37:             Add {key, -1, -1} to cache
38:             key.op_list.append(value/op)
39:     key.snapshot.apply(op)
40: ListOfKeys = getKeys(ListOfUpdates)
41: Link(ListOfKeys)
42: StartTimer(ListOfKeys)
```

Spec. 3 Write

3.4.2 Handover protocol

The handover protocol has two major contributions to the protocol. First contribution is making sure that no key is cached for more than the predefined amount of time. The second contribution makes sure that all causally related dependencies commit atomically under the same commit timestamp, necessary for creating new causally consistent snapshots.

The handover protocol has been developed with the purpose of scheduling evictions such that causal consistency is satisfied. While further development could provide adaptive strategies for different read or written frequencies, in the current version of the protocol the time lease is fixed and will act as a possible upper-bound on staleness.

There are three main reasons for a key to be evicted:

- A timer expires
- If the cache is full and the triggered eviction selects the keys based on some strategy (LRU, LFU, etc.)
- If a cached key is causally connected by local updates to a key that is about to be evicted.

The first case is the nominal use-case scenario. Each time a set of keys is cached, with no other dependency on existing cached keys, a callback function with the timer is spawn in a different thread. The timer has the equivalent value of the lease. When it expires the callback is called. The callback's mission is to trigger a function that gets the updates of each key and send them to the owner to be merged.

The second case is triggered when the high-memory watermark is exceeded. Honeypot relies mainly on the in-memory table. The size of the memory used by the table needs to be kept in check, if the size of the table exceeds a preconfigured size, the protocol starts evicting keys based on some specified strategy (least frequently used, least recently used, etc.), until the new data will fit.

The last and very frequent scenario of the eviction mechanism is needed for maintaining causality, as we will see from the fallowing example.

We assume four nodes A, B, C and D with A and B owning keys 'a' and 'b', respectively. Initially, the versions of the keys are 0. At some point, Node C caches keys 'a' and 'b' sequentially, starting a timer each time. Before key 'a' 's timer expires, 'a' transaction updates both keys creating versions 'a_{20}' and 'b_{20}'.

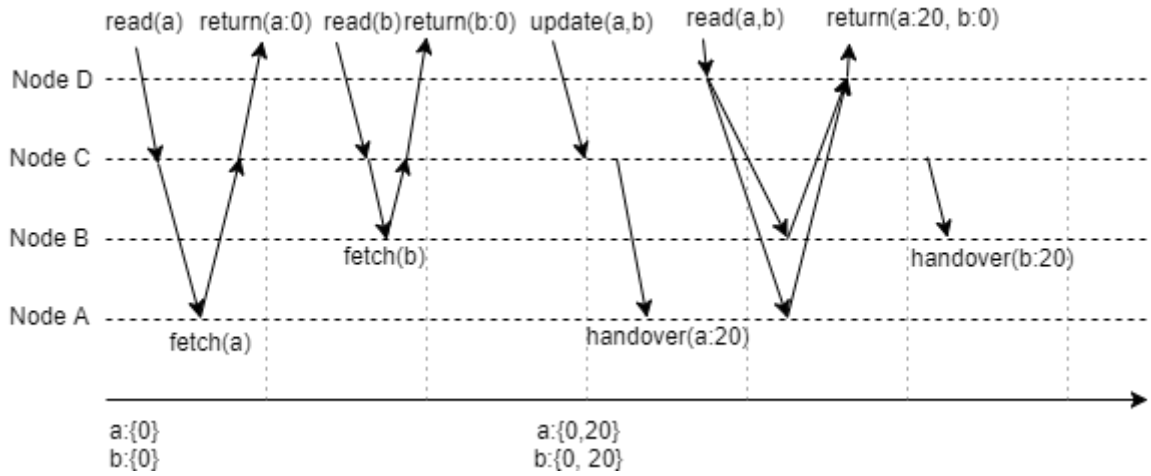

Fig. 7: Transaction isolation violation

After the updated version has been sent back to its owner, node D will try to read both keys. Since D's cache is empty, it will fetch 'a_{20}' and 'b_0', causing a causally inconsistent view

To be able to keep track of the causal dependencies between interleaved operations (be it *thread of execution, gets from, transitive dependency* or *transactional closure*), an in-memory graph is kept alongside the in-memory object store, similar solutions have been found in systems like COPS-GT [35].

Initially, the graph is empty. When queries are received at the API level, a new strongly connected component is created. The vertex set is made of all the new keys together with, possibly, an additional vertex containing the reference to a timer. The algorithm tries to minimize the number of redundant timers in the system to reduce the overhead and memory size. This happens frequently, in the case of a partial hits when only a subset of the keys in a transaction are locally cached. To do so, for every key in a

transaction its hit-count is checked. Normally, the hit-count is used as a discriminator for eviction strategies. When an object is freshly barrowed its hit-count defaults to 0. When a transaction touches the object, the hit-count is automatically bumped. A positive hit-count means that the key has been already pre-cached by a previous transaction and is connected to a timer reference vertex in its connected component. To keep track of causality between already cached keys, edges are added between vertices, connecting keys to strongly connected components or components among themselves.

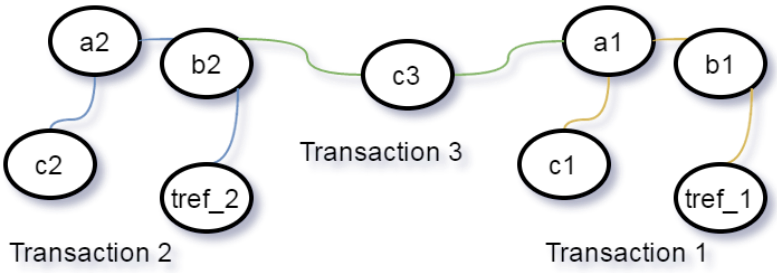

Fig. 8: Liking transactions

Initially a read transaction fetches keys a_1, b_1 and c1. They have been fetched together so they need to get evicted at the same time. A connected component is created (yellow edges) and single timer is added, tref_1. The same goes for keys a_2, b_2 and c_2 (blue edges). After some time, a transaction wants to update keys a_1, b_2 and c_3. Keys a_1 and b_2 are already cached so when c_3 is fetched, no additional timer is needed and will be linked directly to the right keys (green edges) created connected components creating a bigger key chain.
Since the yellow component is the oldest its timer will expire first. All the keys in the same connected component will be handed over and the rest of the timers (tref_2) deactivated.

When a timer in a connected component expires, the entire key-chain will be evicted and remaining timers deactivated. A dictionary is computed using the hashing function applied on the keys to determine where to send the updates. Precomputing the write-sets may reduce the network load since the updates for a particular node could be sent in just one message.

After computing the write set, a two-phase commit (2pc) protocol, similar to that of Clock-SI, is fired. The caching node which starts a 2pc session is named a coordinator. During the first step, the cached operations are sent to each owner by calling the *Prepare* function (*HandOver* specification, line 47). Upon receiving the *prepare* message, the owner-node logic waits for pre-existing updates to commit (line 64 of the *Prepare* specification), buffers the current updates (line 65) and proposes a commit timestamp (line 65), representing the local clock. When the coordinator receives all the replies of the participant nodes, it elects the highest timestamp and sends it back. At the owner node, the commit timestamp received from the coordinator is used as a version identifier for the new snapshot of the key. All other updates on the same key are kept in a FIFO buffer and blocked until the current 2pc has finished. Once committed, the oldest pending *prepare* message for that key is popped and a bigger-than-previous-version timestamp is proposed. Because a commit timestamp is at least as recent as the one proposed by an owner, and concurrent updates are serialized using a FIFO buffer, no two transactions will commit on the same key with the same timestamp. In other words, the commits are monotonic.

HoneyPot HandOver specification

```
44: HandOver(ListofUpdates updates)
45:     TS_max = 0
46:     For key, ops in updates
47:                 TS = Prepare(key, ops) at owner node of key
48:                 TS_max = max(TS_max, TS)
49:         For key, _ in updates
50:                 commit(key, TS_max)
```

Spec. 4 HandOver

HoneyPot Prepare specification

```
63: Prepare(Key, ops)
64:     Wait until no record of key is being prepared
65:     Add a prepare record of key
66:     Reply  local_clock()
```

Spec. 5 Prepare

Back again at the owner node, the *Commit* function will remove the operations from the prepare record (line 69) allowing pending prepares to proceed and apply the operations in an ordered manner creating a new causally consistent snapshot with the versions equal to the provided timestamp.

HoneyPot Commit specification

```
68: Commit(Key, CommitTime)
69:     Remove prepare record of key
70:     NewSnapshot = latest_snapshot.apply(stored ops)
71:     NewSnapshot.timestamp = CommitTime
```

Spec. 6 Commit

3.4.2.1 Operation compaction – HoneyPot Optimization

The compaction mentioned earlier is a simple way to reduce network load, memory space and overhead. Compaction takes leverage over the commutativity property of some object types and creates an equivalent, more compact state.

> *Assuming a counter CRDT and x different increment operations summing up to S from Actor1 and y decrements from Actor2 adding up to D, a compaction act actor level would return one operations for each actor, namely increment(s) for Actor1 and decrement(D) for Actor 2, or an increment(S-D) for a global compaction.*

3.4.3 Crash recovery

Crash recovery is a crucial feature of the Honeypot protocol which can guarantee transactional semantics under failure.

> *Let's assume some clients connected to some node in the cluster and updated a set of keys. Before the updates are sent back to the owners, the node crashes. Without any recovery mechanism in place the updates are lost and any user connecting to any node hereafter won't be able to see the writes, violating the causal consistency guarantee.*

Persisting each update to disk is useful in the context of a crash. If a node goes offline without any damage to the physical storage, once back on line all the updates are fetched from the log and handed to the owner.

Since the version of the target platform against which the test have been carried out did not commit data to disk, this feature has been intentionally left out of the implementation but can be very easily added by replacing the data storage abstraction from an in-memory table to a disk-persisted log.

3.5 CORRECTNESS

It this section we will prove the correctness of the algorithm with respect to its transactional guarantees.

A model is transactional causally+ consistent if every snapshot contains all its causal dependencies applied in causal order, and every transaction observes a consistent snapshot and its own updates [20]. A snapshot having these properties is said to be consistent while missing causal dependencies form a *causal gap.*

Proposition1: Reads return values from a causally consistent snapshot.

Proof:

We assume that a read transaction T reads data items [k_1, k_2, … k_n] with snapshot time s_t, which returns items with versions [v_1, v_2, … v_n]. By contradiction, among all the keys that were read, there exists k_m that has a version v_x, such that $v_m < v_x \leq s_t$. v_x may be fetched either from cache, if the coordinator node of this transaction has a consistent version cached, or from the original node of the key if no consistent version is cached.

We first consider that the version is fetched from cache. As described in section 3.4.1, for T to read v_m from cache, the visibility limit of v_m, i.e., v_{lm}, must be no less than s_t ($v_{lm} \geq s_t$). However, since our protocol (line 12 of the *Read* specification) ensures that $v_{lm} < v_l$, this means $v_x > st$, which contradicts the assumption.

Then we consider that v_m is fetched from its owner node. As such, v_x may either be created before or after the read is performed. In the former case, the read operation would fetch v_x instead of v_m. In the latter,

after the read is finished, the clock of the owner partition, t_{clk}, must be at least as large as s_t, and since v_x is created after the read operation, it cannot obtain a timestamp less than s_t (due to our commit protocol). As such, $v_x \geq t_{clk} > s_t$, which contradicts the assumption.

Proposition 2: Reading from a snapshot respects atomicity.

Proof:

We assume that an update transaction T updates data items [k_1, k_2, … k_n] and it commits with a timestamp c_t, which creates a corresponding new version for each updated key.

By contradiction, a snapshot read does not respect atomicity if it only reads partial updates of T but misses others. Without loss of generality, we assume a transaction T_r that issues read to k_1 and k_2, and it reads k_1 with version c_t, but k_2 with version v_2 such that $v_2 < c_t$.

Having proved Proposition 1, it is trivial to see that our protocol always enforces atomicity for snapshot reads: since we provide causally consistent read, T_r can only read v_2 if $v_2 \leq T_r < c_t$; however, then T_2 would not be able to read version c_t of T_1.

4 PERFORMANCE

In this section, we will assess Honeypot's performance increase by using plain-vanilla Antidote as a baseline. The load test has a concurrency factor of 50, meaning 50 threads will concurrently query the key-value store and measure the response time of each request as well as the number of successful operations fulfilled by the target.

To emulate different cases of keyspace have been generated using uniform and Pareto distribution laws over ranges of 10, 100, 1000 and 10000 values. Adjusting the range of possible values and the distribution law impacts the cache hit probability. The uniform distribution of keys can be treated as normal use case while using the Pareto distribution would emulate an interest in some specific subset of keys.

We will present our findings a series of figures describing the transaction ratio (read or write) with respect to the key value range, distribution law and lease time. Each transaction concerns exactly ten distinct keys. The memory threshold for the cache is set to 10 Mb.

The target cluster was created in a dockerized environment inside a virtual machine on top of a bare-metal machine with i5-760 CPU @ 2.8 Ghz (max 3.3 Ghz), 16GB of RAM and SSD storage.

4.1 THROUGHPUT AND LATENCY

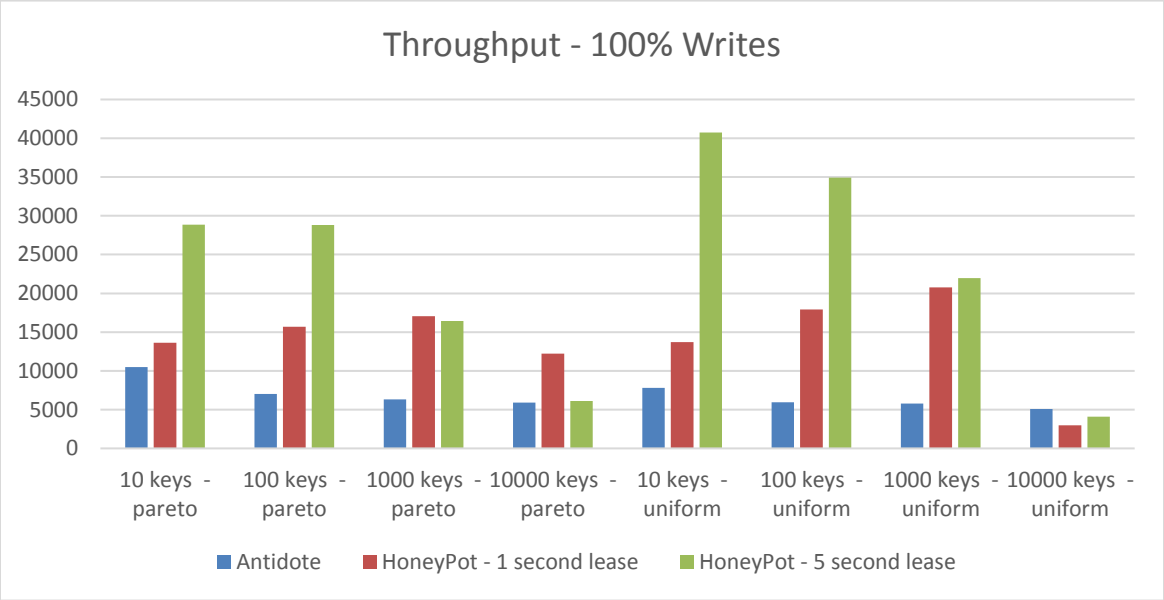

Fig. 10: Throughput for 100% write workload.

We start by isolating the write part of the protocol under different contexts to measure the impact of lease size over different cache-hit rates. As we can see, overall Honeypot outperforms plain vanilla Antidote, regardless of the lease size. However, there is a counter-intuitive throughput difference

between a one second and five second lease. This is due to the fact that the write-protocol creates a dependency tree which grows over time, especially in the case of wide ranges of keys. Here, the overhead of the 2-phase protocol outweighs the benefit of caching.

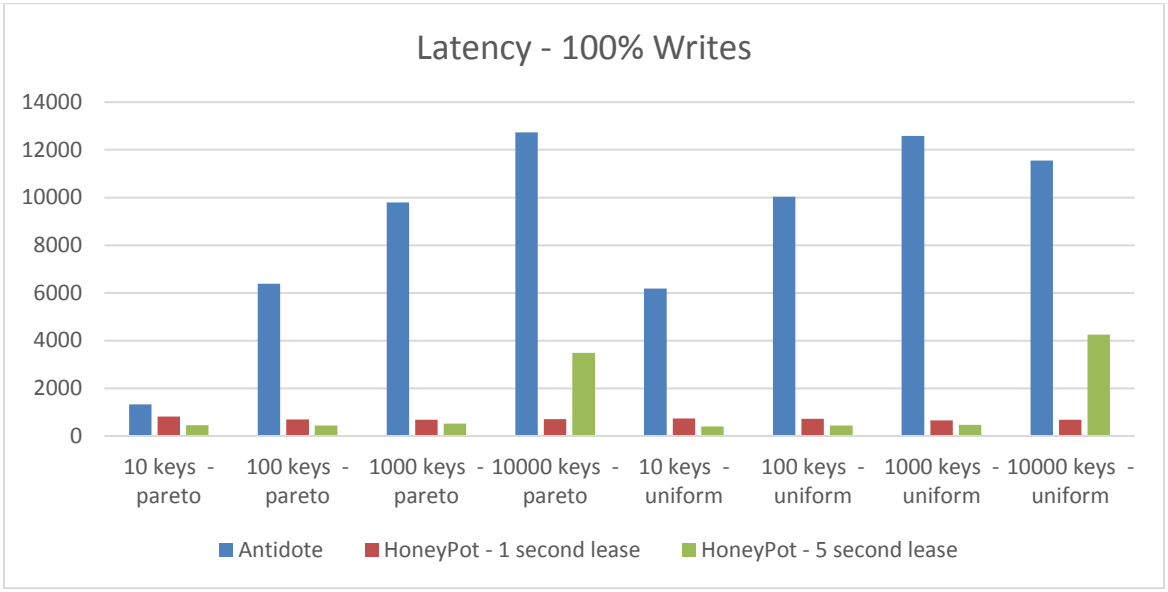

Fig. 11: Latency for 100% write workload.

As we can see, the latency of write-only transactions is consistent (inversely proportional) to the throughput measured.

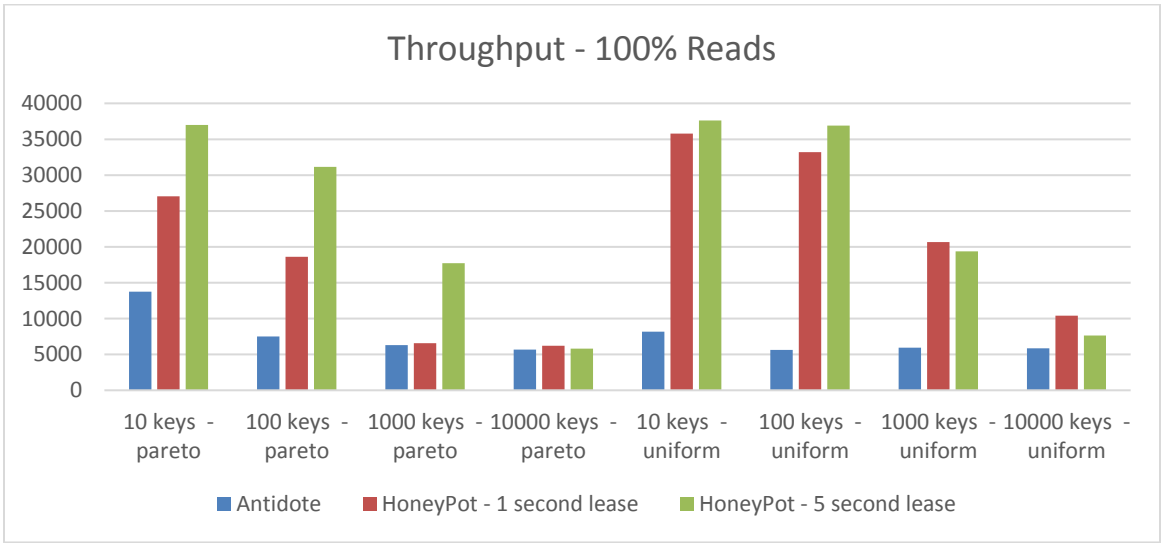

Fig. 12: Throughput for 100% read workload.

When measuring the read mechanism in isolation we observe similar results. Although there is no 2-phase commit logic, the dependencies are still kept in a tree, which will only increase with time. Having to detect

version consistency between more snapshots and fetching the correct ones is detrimental to the overall performance.

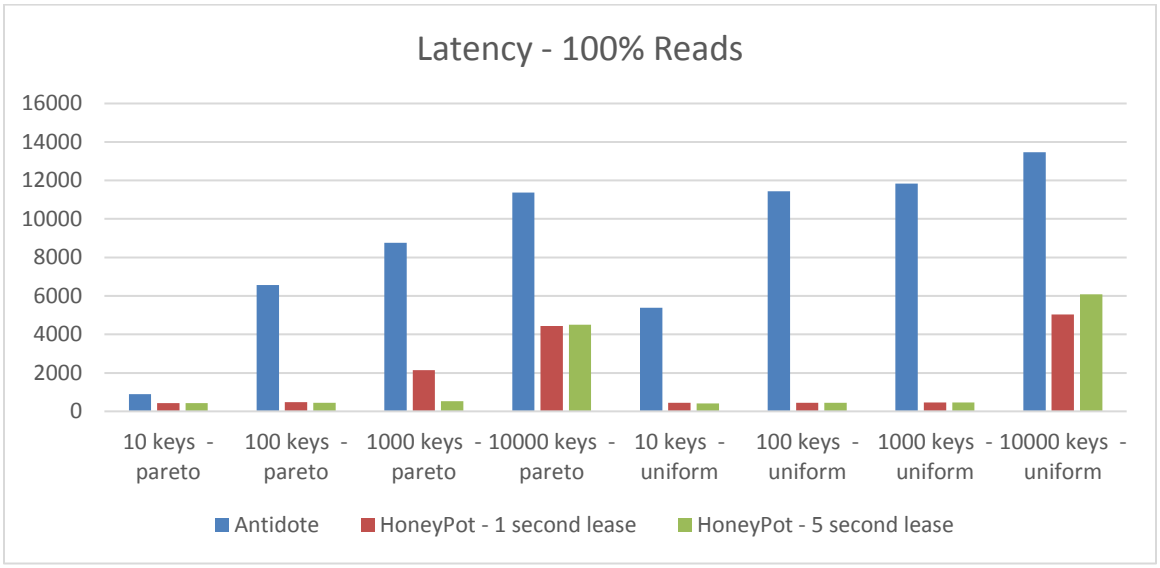

Fig. 13: Latency for 100% reads workload.

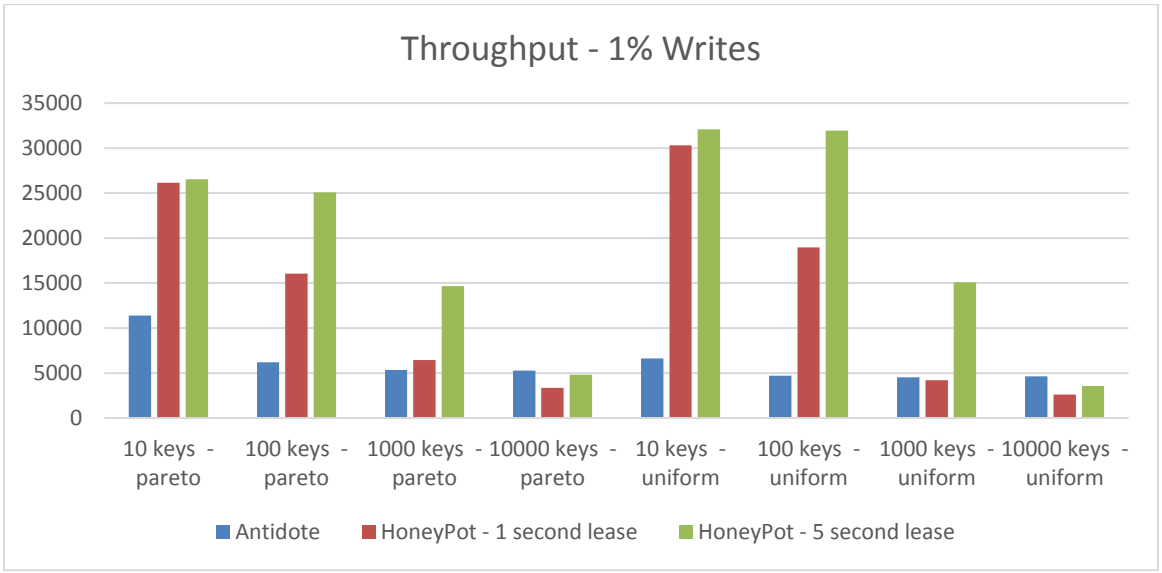

Fig 14: Throughput for 1% write – 99% read workload.

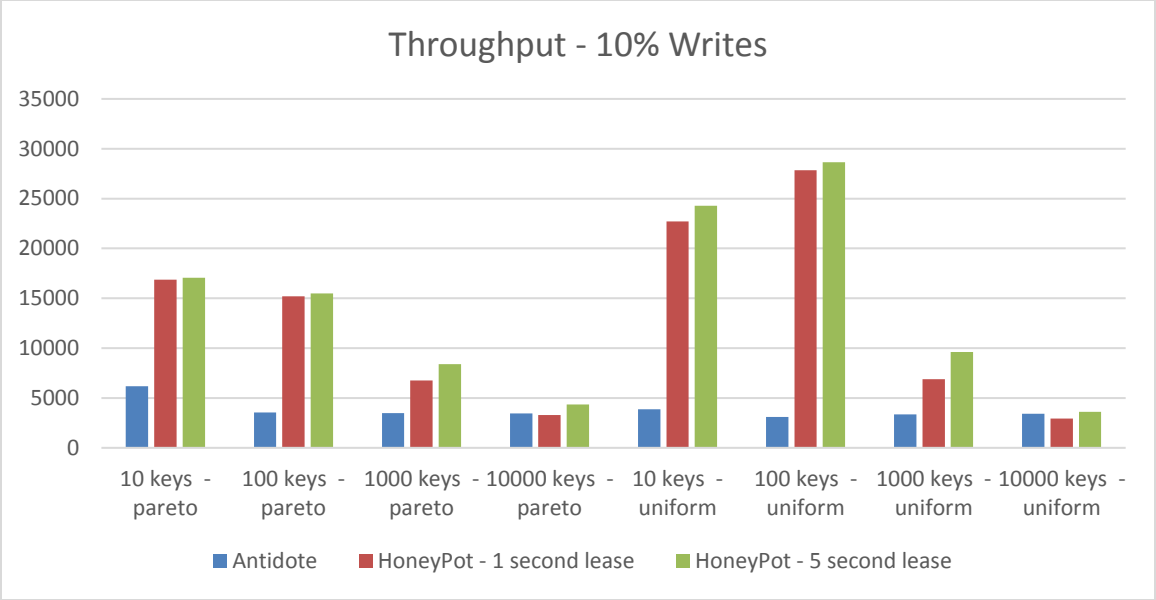

Fig. 17: Throughput for 10% write – 90% read workload.

The 10% write - 90% read ratio is arguably the closest of the test suits to the real use-case scenarios found in practice [37]. The test show again the impact of a lease over different cache-hit frequencies. We can observe that in the case of a near 100% cache-hit rate, the size of the lease does not create a huge gap between throughputs. In the best-case scenario, HoneyPot offers three to four times the performance of the target system, in average just twice and the worst-case scenario approximately the same.

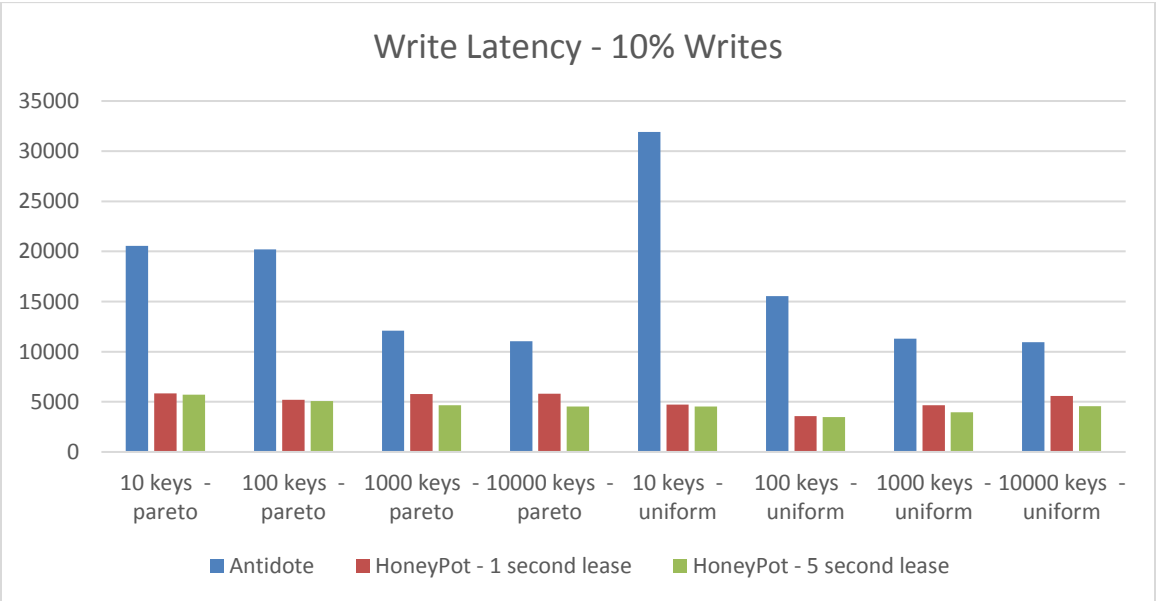

Fig. 18: Latency for writes, 10% write – 90% read workload.

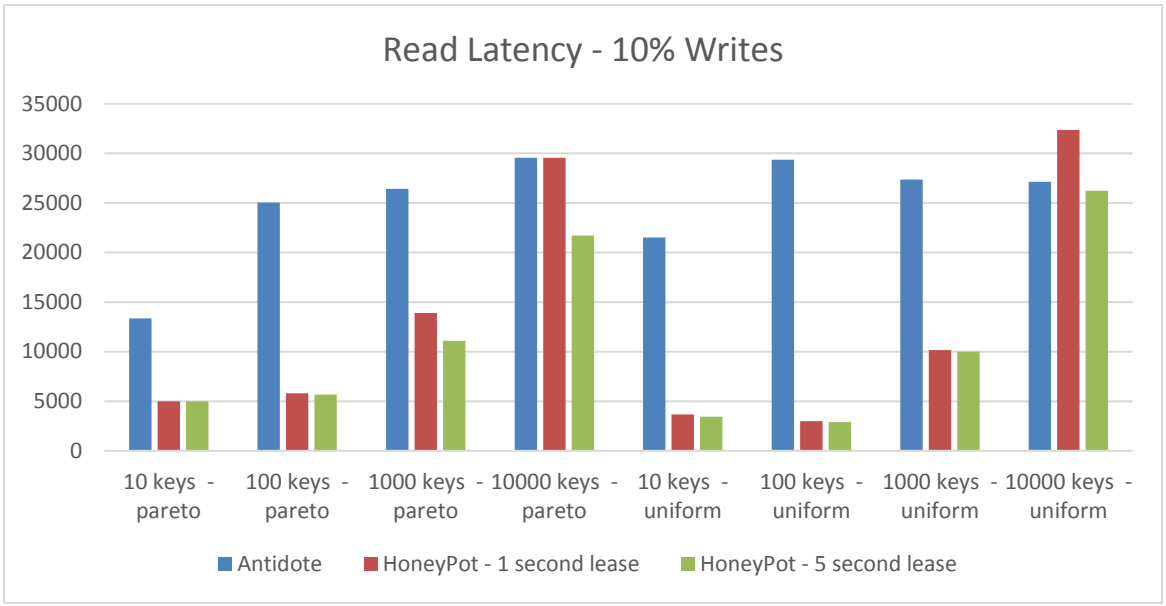

Fig. 19: Latency for writes, 10% write – 90% read workload.

Updates are stored locally, compacted and evicted once the lease has expired. Once evicted, the 2pc protocol takes over and tries to hand over the list of operations for each key to their owner, possibly having to wait for concurrent pending commits to finish. On the other hand, reads are most of the time non-blocking (read from the past) and the local application of operations is O(1) since all the updates have been compacted as soon as they have arrived at the caching node. These considerate explain why the read latency is, at its worst, still better than the write latency. However, unlike the write protocol whose latency is invariant to the cache-hit rate or the existing state of the cache, the read protocol varies in this sense. If keys are missing or cached but with incompatible versions, the algorithm generates extra overhead, and it is impossible to estimate ahead of time.

4.2 Load Balancing

Each node in the cluster is responsible for at least one partition. In the current setup, Antidote has twelve non-overlapping partitions distributed over ten nodes, rendering two nodes with two partitions each. In our case, nodes named Master and Node 2. Managing twice the number of partitions doubles the chance that any given transaction will be interested in a key residing at that particular node.

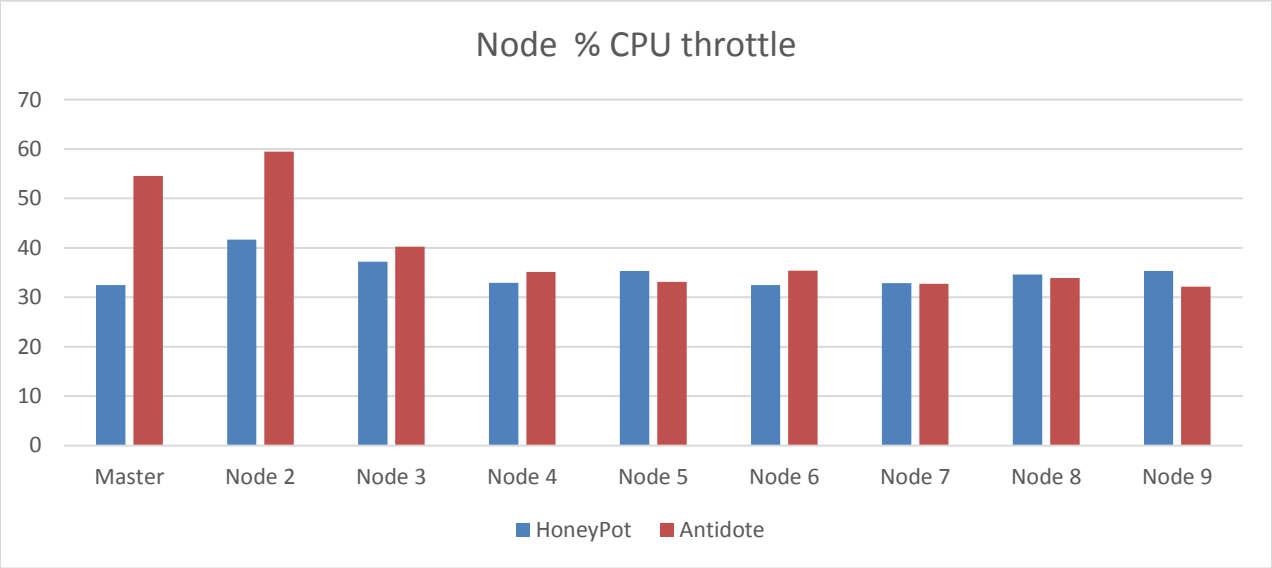

Fig. 20 Cluster node CPU throttle

Fig. 20 illustrate the average CPU throttle of each node in the cluster. Master and Node 2 have a higher load than the rest of the nodes in the Antidote cluster while for HoneyPot, the difference is significantly smaller. This proves that HoneyPot is able to absorb a part of the load by handling operations on behalf ow the owner (in this case nodes Master and Node 2).

In Fig. 21, we can see how HoneyPot is able to distribute the load across the entire cluster, whereas for Antidote, network usage of nodes Master and Nodes 2 standout.

Fig. 22 depicts the overall usage during the period of the test. It allows us to observe the amount of traffic generated by each node. Since HoneyPot does not need to contact the owner for each incoming request, the network resources needed are significantly smaller.

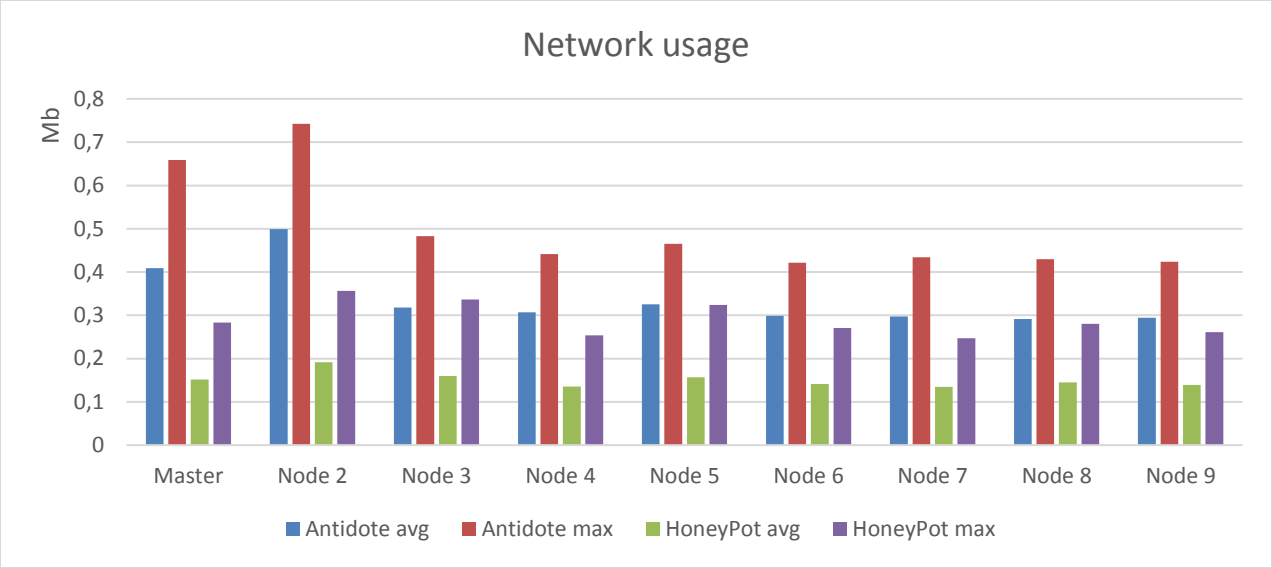

Fig. 21 Cluster per node network statistics

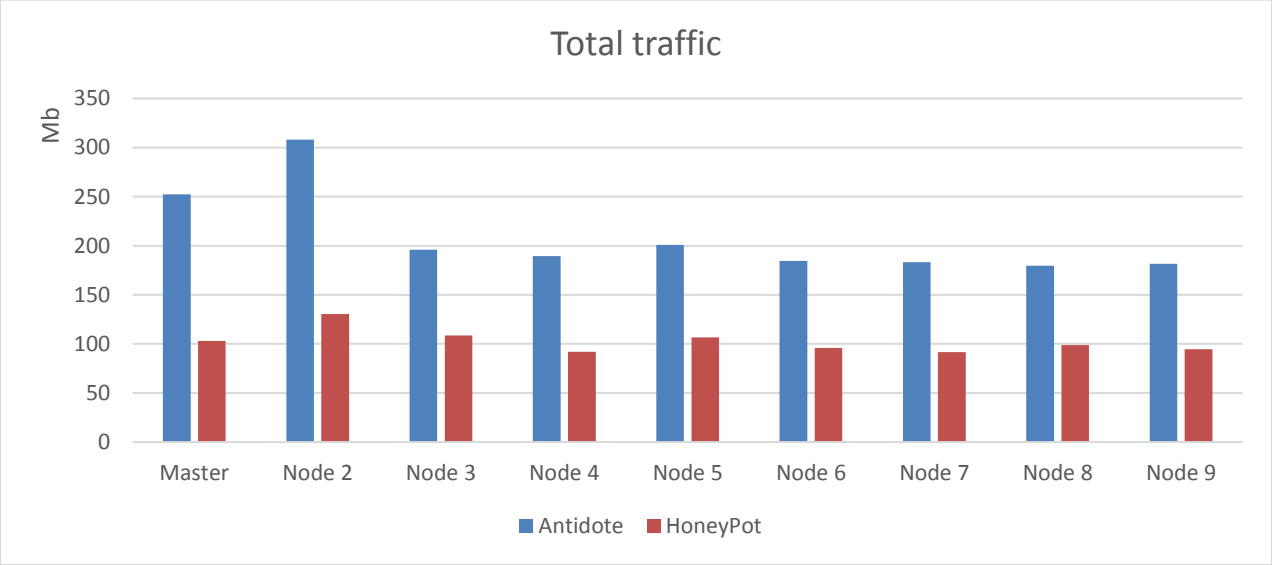

Fig. 22 Cluster node total bandwidth consumption

5 CONCLUSION

Honeypot is a lightweight caching protocol with Transactional Causal+ semantics. HoneyPot boosts performance even under small network latency. From the performance profile, we can clearly see that in a real-life scenario, throughput is increased two-fold for a lease of one second, and has little drawback in very low cache-hit rates scenarios.

Honeypot provides internal load balancing, preventing appearance of hot-spots or bottlenecks that may appear due to sudden increase in interest for a specific set of keys.

5.1 FURTHER WORK

5.1.1 Adaptive leases

Depending on an object's life cycle, update frequency or some caching node's affinity, its lease duration may be tailored to best suit these usage parameters. Some leases, called adaptive lease [34] are able to adapt on the fly to best suit the system's requirements. These dynamic policies are able to conform to fluctuating environmental parameters on the fly by periodically re-computing its duration. The end result may lead to improved availability, smaller staleness and less network traffic.

5.1.2 Crash-recovery – improvement and testing

Even with a disk persisting strategy, a full crash-recovery might be impossible due to mechanical or electronical failure. To prevent amnesia caused by damage to the physical drives, another strategy could be disseminating the updates to neighbor nodes. They could hand over the writes if the initial caching node fails.

Once the crash-recovery are in place, the system needs to be benchmarked under different loads and failure conditions.

6 REFERENCE LIST

[1] Gilbert, Seth, and Nancy Lynch. "Perspectives on the CAP Theorem." *Computer* 45.2 (2012): 30-36.

[2] Nayak, Ameya, Anil Poriya, and Dikshay Poojary. "Type of NOSQL databases and its comparison with relational databases." *International Journal of Applied Information Systems* 5.4 (2013): 16-19.

[3] Cattell, Rick. "Scalable SQL and NoSQL data stores." *Acm Sigmod Record*39.4 (2011): 12-27.

[4] Karger, David, et al. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web." *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997.

[5] Souri, Alireza, and Amir Masoud Rahmani. "A survey for replica placement techniques in data grid environment." *International Journal of Modern Education and Computer Science* 6.5 (2014): 46.

[6] Cristina L. Abad, Yi Luy, Roy H. Campbell, “DARE: Adaptive Data Replication for Efficient Cluster Scheduling” University of Illinois at Urbana-Champaign.

[7] Nishtala, Rajesh, et al. "Scaling Memcache at Facebook." *nsdi*. Vol. 13. 2013.

[8] Du, Jiaqing, et al. "Gentlerain: Cheap and scalable causal consistency with physical clocks." *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014.

[9] Zawirski, Marek, et al. "Write fast, read in the past: Causal consistency for client-side applications." *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015.

[10] Brewer, Eric. "CAP twelve years later: How the" rules" have changed." *Computer* 45.2 (2012): 23-29.

[11] Berenson, Hal, et al. "A critique of ANSI SQL isolation levels." *ACM SIGMOD Record*. Vol. 24. No. 2. ACM, 1995.

[12] Fekete, Alan David. "Serialisability and Snapshhot Isolation." In *Australasian Database Conference*, pp. 201-209. 1999.

[13] Fekete, Alan, et al. "Making snapshot isolation serializable." *ACM Transactions on Database Systems (TODS)* 30.2 (2005): 492-528.

[14] Bailis, Peter, et al. "Bolt-on causal consistency." *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013.

[15] "J. Liddle. Amazon Found Every 100ms of Latency Cost Them 1% in Sales." Retrieved August 15th from http://goo.gl/BUJgV .

[16] Lakshman, Avinash, and Prashant Malik. "Cassandra: a decentralized structured storage system." *ACM SIGOPS Operating Systems Review* 44.2 (2010): 35-40.

[17] DeCandia, Giuseppe, et al. "Dynamo: amazon's highly available key-value store." *ACM SIGOPS operating systems review* 41.6 (2007): 205-220.

[18] Terry, Doug. "Replicated data consistency explained through baseball." *Communications of the ACM* 56.12 (2013): 82-89.

[19] Cooper, Brian F., et al. "PNUTS: Yahoo!'s hosted data serving platform." *Proceedings of the VLDB Endowment* 1.2 (2008): 1277-1288.

[20] Zawirski, Marek, et al. "Write fast, read in the past: Causal consistency for client-side applications." *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015.

[21] Mahajan, Prince, Lorenzo Alvisi, and Mike Dahlin. "Consistency, availability, and convergence." *University of Texas at Austin Tech Report* 11 (2011).

[22] Brewer, Eric A. "Towards robust distributed systems." *PODC*. Vol. 7. 2000.

[23] Akkoorath, Deepthi Devaki, et al. "Cure: Strong semantics meets high availability and low latency." *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE, 2016.

[24] Lechtenbörger, Jens. "Two-phase commit protocol." *Encyclopedia of Database Systems*. Springer US, 2009. 3209-3213.

[25] Kleppmann, Martin. "A Critique of the CAP Theorem." *arXiv preprint arXiv:1509.05393* (2015).

[26] Herlihy, Maurice P., and Jeannette M. Wing. "Linearizability: A correctness condition for concurrent objects." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990): 463-492.

[27] Bernstein, Philip A., Vassos Hadzilacos, and Nathan Goodman. *RRENCY CONTROL AND RECOVERY IN DATABASE SYSTEMS*. Addison− Wesley, 1987.

[29] Ahamad, Mustaque, et al. "Causal memory." *International Workshop on Distributed Algorithms*. Springer, Berlin, Heidelberg, 1991.

[30] Klophaus, Rusty. "Riak core: Building distributed applications without shared state." *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, 2010.

[31] Almeida, Sérgio, João Leitão, and Luís Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication." *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013.

[32] Merkel, Dirk. "Docker: lightweight linux containers for consistent development and deployment." *Linux Journal* 2014.239 (2014): 2.

[33] Bailis, Peter, et al. "Highly available transactions: Virtues and limitations." *Proceedings of the VLDB Endowment* 7.3 (2013): 181-192.

[34] Duvvuri, Venkata, Prashant Shenoy, and Renu Tewari. "Adaptive leases: A strong consistency mechanism for the world wide web." *IEEE Transactions on Knowledge and Data Engineering* 15.5 (2003): 1266-1276.

[35] Lloyd, Wyatt, et al. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS." *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011.

[36] M. Ahamad, J. E. Burns, P. W. Hutto, and G. Neiger. Causal memory. In Proc. 5th Int. Workshop on Distributed Algorithms, pages 9–30, Delphi, Greece, Oct. 1991.

[37] Redis. Extracted form https://redis.io/ August the 15th.

[38] Redis Replication. Extracted from https://redis.io/topics/replication August the 15th .

[39] Du, Jiaqing, et al. "Gentlerain: Cheap and scalable causal consistency with physical clocks." *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014.

[40] Du, Jiaqing, et al. "Orbe: Scalable causal consistency using dependency matrices and physical clocks." *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013.

[41] Lloyd, Wyatt, et al. "Stronger Semantics for Low-Latency Geo-Replicated Storage." *NSDI*. Vol. 13. 2013.

[42] Du, Jiaqing, et al. "Gentlerain: Cheap and scalable causal consistency with physical clocks." *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014.

[43] Almeida, P. S., Shoker, A., & Baquero, C. (2015, May). Efficient state-based crdts by delta-mutation. In *International Conference on Networked Systems* (pp. 62-76). Springer International Publishing.