FleetDB: Follow-the-workload Data Migration for Globe-Spanning Databases

Aleksey Charapko
University at Buffalo
acharapk@buffalo.edu

Ailidani Ailijiang
University at Buffalo
ailidani@buffalo.edu

Murat Demirbas
University at Buffalo
demirbas@buffalo.edu

Abstract. Globally distributed databases provide limited support for access locality optimization and fine-grained object migration. We present FleetDB: a WPaxos-backed strongly-consistent geo-distributed database that adapts to the locality of accesses quickly, safely, and at the most fine-grained level —any object can migrate freely across regions to be oriented for ideal performance. FleetDB achieves near-local average read and update latencies for workloads that show access-locality. By leveraging its core WPaxos protocol, FleetDB also implements transactions across objects and provides per-object linearizability and transaction serializability.

1. INTRODUCTION

Today several distributed databases span across many datacenters and geographic regions to serve users across the globe [7, 8, 5]. These systems often experience changing access patterns across the regions [17] and need to adjust to the workload locality in order to allow faster read/write operations and to eliminate/reduce the wide area network (WAN) latencies. Since cross-region communication typically takes up to two orders of magnitude more time than local/intra-datacenter communication, trading even a small fraction of remote operations for local ones can provide a significant speedup. By reducing cross-datacenter traffic, these systems can also save on operational costs.

Unfortunately, very few database systems provide native support for data migration in response to shifts in workload patterns. Some systems, like PNUTS, ZooNet, and Eiger replicate the data globally to allow local reads at any datacenter [6, 15, 16]. While this approach improves read performance, the writes remain slow, as a dedicated region is responsible for all updates and the traffic must go through that region. Spanner employs Paxos-groups across datacenters to replicate the data, and has the capability to relocate data from one Paxos-group in one region to a group in another region, using a movedir command [7]. CockroachDB adjusts the leaseholder node in each Raft-group to relocate an object to a nearby datacenter [5].

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Even fewer systems [12, 18] provide high-resolution, object-level data placement/migration strategies. The majority of databases, however, often chunk data into large blocks or partitions, limiting object/data-level relocation. For instance, in Spanner and CockroachDB data items sharded to a partition must follow the same geographical placement and replication rules as all other items of that partition. Datastores that adapt the locality at a per-object level often do not provide strong per-record/object consistency guarantees. For instance, selective replication in PNUTS [12] allows the system to replicate records to a sub-set of all datacenters based on the access pattern, but such replicas may still provide stale data to the client. However, some applications, such as metadata management for global-spanning file systems [3] or multi-user document sharing, require strong consistency guarantees all while providing low latency at the global scale.

Contributions. We present FleetDB, a globe-spanning, write-optimized, low-latency, strongly consistent key-value datastore with per object linearizability and transaction serializability. Unlike previous work, FleetDB can adapt to the locality of accesses quickly and at the most fine-grained level: any object can automatically relocate across regions for ideal performance. To manage object-locality, FleetDB supports a rich set of object-migration policies. A simple policy may move each object to the region where majority of accesses for that object originate. More sophisticated policies can take into account minimizing latency for multi-party clients simultaneously, exploiting the replication groups to provide quicker migration, and adhering to load balancing and resource capability concerns of the clusters. In addition to providing these guarantees, FleetDB supports atomic, serializable multi-object transactions as well. The transactions are implemented at the core protocol level, all within Paxos to ensure reliability and safety.

To provide assurance, reliability, and strong-consistency in FleetDB, we leverage our recent work on WPaxos [2], a WAN-optimized multileader Paxos protocol. WPaxos selects leaders (and maintains logs) to be per-object, and employs an object stealing protocol, with adaptive stealing improvements to match the workload access locality. Multiple concurrent leaders coinciding in different zones steal ownership of objects from each other using phase-1 of Paxos, and then use phase-2 to commit update-requests on these objects locally until they are stolen by other leaders. To achieve fast phase-2 commits, WPaxos appoints phase-2 acceptors to be close to their respective leaders within the same or adjacent columns in the grid topology overlaid across regions. This way WPaxos provides per-object linearizability in an efficient and fault-tolerant way, while obviating the need for another service for object relocation.

In FleetDB we adopt WPaxos in a novel way to provide dynamically configurable partial replication capability. By providing configurable fault tolerance, FleetDB allows users to strike a balance between desired levels of performance and fault tolerance. In the fastest configuration, FleetDB can only tolerate some node failures within each zone, but not entire zone crash. More resilient setups, however, can handle full datacenter outages. Moreover, unlike the generic WPaxos, FleetDB does not perform full replication to all nodes, unless it is configured to do so. It replicates the objects within a *replication group*, consisting of one or more availability zones.

Since a naive application of WPaxos's object stealing may lead to an imbalance in the system, FleetDB controls the global distribution of data in the system and take actions to prevent disproportional load. FleetDB is designed to halt object stealing to a leader upon reaching a certain misbalance threshold at that node. Instead, FleetDB resorts to one of the following measures: stealing to another leader within the region, stealing to another leader in a nearby region, or performing object swap —explicitly giving up the leadership of one object in exchange for another.

While allowing for a fine-grained, object-level locality control, FleetDB manages to support transactions and enables atomic multi-object operations *all within the Paxos protocol*. By staying within Paxos, the FleetDB transactions are guaranteed to be safe/correct to the face of concurrency and faults, and FleetDB does not incur extra complexity of overlay protocols. This is achieved by acquiring the leadership of all objects involved in a transaction under the same node. While this disrupts the normal data placement procedures —as the objects must be transferred to a single region regardless of their access locality—, it is often the case that the same objects participate in a transaction more than once, and this alleviates the relocation cost.

We implemented FleetDB in Go as an opensource project https://github.com/acharapko/fleetdb and evaluated the system to showcase its dynamic object migration and transaction capabilities. We show that FleetDB exhibits a bimodal latency pattern for the client requests: for the local items the system may finish in under 10 milliseconds, while accessing remote objects incurs additional WAN RTT. Despite having to pay WAN costs occassionally, with good locality the system achieves an average latency of just 30 milliseconds. For workloads that include transactions, the locality also benefits the performance when the system is configured to have a dedicated leader per zone.

2. FLEETDB

FleetDB is a globally distributed, write-optimized, strongly consistent key-value datastore with per object linearizability and transaction serializability. FleetDB keeps track of object-locality and uses an object-migration policy to move each item to the region of majority access. The system uses REST API to expose basic client operations, such as Get, Put, and Delete.

We illustrate the high-level overview of FleetDB architecture in Figure 1. The system consists of nodes, arranged in a grid manner; each column in the grid represents the

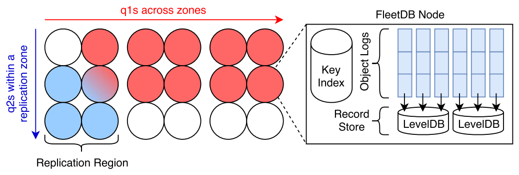
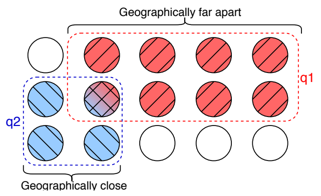
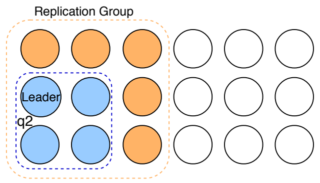


Figure 1: Overview of FleetDB architecture.



(a) WPaxos $q1$ and $q2$ quorum and their intersection.



(b) A $q2$ within a replication group in FleetDB.

Figure 2: WPaxos quorums and FleetDB replication groups

machines in the same datacenter or availability zone. Every node of the datastore is identical, and consists of logs for each object owned by the node, key cache and sharded persistent storage. FleetDB uses WPaxos [2] for object log replication.

2.1 WPaxos

WPaxos is a multileader WAN-optimized Paxos protocol [2]. Similar to other Paxos [14] variants, WPaxos operates in three distinct phases: promise (phase-1), accept (phase-2) and commit. Phase-1 establishes some node in the system as a leader with the follower nodes promising to accept leader's commands. Phase-2, typically repeated many times, tells the followers to accept a command, and a commit phase finalizes the commands and allows replicas to execute them.

The requirement for majority quorums means that in WAN settings, both phase-1 and phase-2 incur significant latency due to the geographical distances between the nodes. Recent flexible quorums [11] result shows that the majority quorums requirement in Paxos can be relaxed to the following: any potential quorum $q1$ used for phase-1 of Paxos must intersect with any potential quorum $q2$ used for phase-2. As shown in Figure 2a, WPaxos leverages this finding to trade off a large but rare phase-1 quorum that spans most or all regions, with a frequently used phase-2 quorum of followers located within the region or nearby regions.

WPaxos can move an object between regions to optimize for access locality. The object migration is safe, as it is carried out entirely within the Paxos protocol. Upon moving an object from one region to another, the leader in new region needs to start the phase-1 of Paxos on some large $q1$ with higher ballot number. In this phase, new leader learns of the object's log since it is guaranteed to intersect the $q2$ quorums that was responsible for accepting and committing object's values. After successful completion of phase-1, new leader may use a nearby $q2$ to commit quickly. Since not all $q1$ quorums intersect in WPaxos, it is possible to for multiple potential leaders to complete phase-1 at the same time. However, only one leader will be able to successfully complete phase-2.

2.2 Replication

FleetDB takes advantage of WPaxos to drive the replication and guarantee safety. In FleetDB, every object has its own WPaxos log, allowing unrelated objects to have different leader nodes and fast $q2$s. Such per-object replication allows for optimal geographical placement of data to minimize the latency.

A replica or FleetDB node is a basic unit of replication in the system. Each replica holds a copy of some objects and may act as a leader for a subset of them. Replicas belong to availability zones, and availability zones belong to some geographical region. Many zones may coexist within the same region.

Unlike many other databases, FleetDB does not have to perform full replication to every region, although it is still an option if so configured. Instead, every object is kept within a *replication group*, spanning one or more availability zone. At the very minimum, the replication group has enough nodes and zones to form a valid $q2$ quorum. However, for increased redundancy, faster failure recovery and faster migration speed, a replication group may span additional nodes and zones, as shown in Figure 2b.

FleetDB uses WPaxos to replicate the log for each object by running a phase-2 in a *replication group* and executing the operations against the datastore. The replication round requires some valid $q2$ within the group to reply back to the object leader. Typically, the fastest replying $q2$ is the one closest to the item owner, however if this quorum is not available due to the fault or network congestion, some other $q2$ may complete the operation. This makes it very important for a replication group to span nearby zones to achieve low operation latency.

FleetDB provides fault tolerance that is configurable by adjusting the number of zones required for each quorum type. For instance, the configuration tolerating no zone failures requires a $q2$ spanning just a single zone, and $q1$ spanning all zones to guarantee the intersection requirement between phase-1 and phase-2 quorums. Such configuration can complete phase-2 operations very quickly but cannot tolerate zone failure. On the other hand, a two-zone $q2$ paired with $n-1$ zone $q1$, where n is the total number of zones, can tolerate one entire availability zone failure while continuing the normal operations.

FleetDB benefits from having a replication group larger than the $q2$ when it comes to failure recovery and ownership transfer. In case of a replication region spanning the same number of zones as the $q2$, a zone failure will prevent the only possible $q2$ from replying to the leader, making the operation time-out before the leader has a chance to try on some other $q2$. Having some quorum redundancy in the replication group eliminates the need for time-out wait and system silently shifts to a second fastest $q2$ in the group.

Even in the case of critical zone failures beyond the tolerance limit, FleetDB provides some availability for disaster tolerance. For instance, in a single-zone $q2$ deployment (i.e., no zone-failure tolerance), after a zone failure, only objects owned by nodes in the crashed zone become unavailable, whereas operations on all objects in non-affected zones can proceed. However, since no $q1$ exists in such configuration, this limits both object migration and new object creation.

2.3 Data Migration Policies

FleetDB aims to provide optimal placement of data within the system at the object granularity. In order to achieve this, we use some object migration policy to define the optimal data placement, which may differ depending on the application or workload.

For example, a policy may optimize for fast object migration. In that case the migration is preferred to happen within the replication group, as this avoids the object's full log copy and reply at the new leader, since all nodes in the replication group are largely caught-up with the object's log.

Some applications with highly region-biased access patterns may benefit from a policy that minimizes latency at the majority access region. One example may be a social application where a user and majority of user's friends are in the same region. With such policy FleetDB migrates the object over to the region of majority access, allowing quick read and write in the region and progressively slower access from further regions.

Some applications, however, may require a more strategic placement of data in the system. For example, a metadata service for a globe-spanning file system, such as SCFS [3] requires both strong consistency to orchestrate the access to some file and optimal latency for users frequently accessing the file. FleetDB can achieve this by placing the metadata object in some central location for the clients, minimizing the access latency for every participant.

Object migration policies also take the data-balance in the system into the consideration. A policy will never pick an overloaded node for migration, even if that node was the most optimal candidate. Instead, the policy will find the next best non-overloaded node for the object placement. FleetDB makes a decision about the load on each replica through a periodic gossip about the number of objects owned by each replica.

The objects migrate when a chosen policy decides that current location is not optimal. To that order, we collect some access statistics to perform a running calculation on the optimal placement, given the policy rules and constraints. Once the current owner of the objects decides to move it to another location, it sends a handover message to the better suited node. Upon receiving the handover message, the new leader must run phase-1 of WPaxos against a $q1$ quorum to establish itself as the new owner of the object. At this point all regions are aware of the leader change and normal operation resumes. FleetDB keeps a hash index of all objects in the system at every node to facilitate both object migration and client requests.

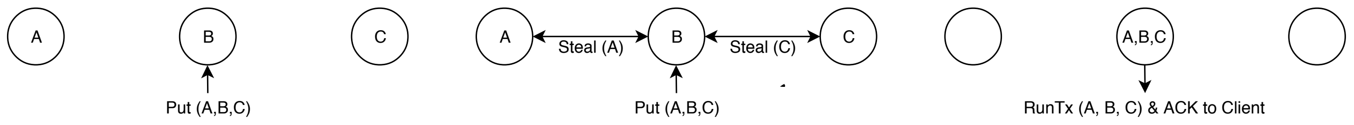
In a situation when some node becomes overloaded, FleetDB employs a backpressure protocol that triggers an overloaded replica to evict some of its least-used objects to other nodes. The rate of eviction depends on the severity of replica's overload.

3. TRANSACTIONS

In addition to key-value point access commands, FleetDB supports minitransactions [1] operating on a set of items. Our minitransactions are serializable atomic write, read, and read-write operations on multiple objects.

3.1 Transaction Protocol

FleetDB implements transactions natively at the core protocol level by leveraging WPaxos. The core idea of transaction execution is to consolidate object ownership at one node and perform transaction commit and execute on all items in the transaction atomically, as shown in Figure 3. We give



(a) Transaction begins in the region owning object B.

(b) A and C are moved to a transaction holder.

(c) Transaction holder commits A, B, C in their logs before atomic execution.

Figure 3: High-level overview of a transaction protocol.

a description of the transaction protocol in Algorithm 1.

Algorithm 1 Minitransaction

```
 1: TxCommands ← request from client
 2: for each $c \in \mathcal{T}xCommands$ do
 3:     if Own($c.key$) = TRUE then            ▷ if own an object
 4:         Put TxLease on $c.key$ with phase-2
 5:     else
 6:         Acquire ownership of $c.key$ with phase-1
 7:         Put TxLease on $c.key$ with phase-2
 8:     end if
 9:     if HasLease($c.key$) = FALSE then
10:         Abort TX                           ▷ Abort Tx if lease failed
11:     end if
12: end for
13: Send TxAccept to $q2$
14: if $q2$ reply OK then
15:     Send TxCommit                          ▷ commit and execute
16:     Commit TxCommands
17:     Atomically Execute TxCommands
18: else
19:     Abort TX                               ▷ Abort Tx due to a q2 NACK
20: end if
```

A replica receiving a transaction request from a client becomes a transaction holder and assigns a timestamp to the transaction. At this point the transaction holder needs to obtain the ownership of all objects involved in the transaction (line 6) by running a Paxos phase-1 for any items it does not have. The holder also tries to acquire *transaction leases* on all transaction objects as shown on lines 4 and 7.

Transaction leases allow FleetDB to reduce the object contention when some other node tries to get the lead of the transaction object before the transaction has been completed. To acquire a lease for an object, a transaction holder commits the lease to the object's log using $q2$. The lease prevents the object migration in case of contention: if some other node with a higher ballot tries to run the phase-1, the node with a lease will reject the phase-1 with a lease time and duration. At this point the incumbent leader will back-off and retry later. Note, that leases do not completely solve the problems associated with object contention, but help minimize some negative performance impacts.

After acquiring all leases, the transaction holder runs the accept phase (phase-2) of WPaxos to get the slots for objects in a transaction and put the values into these slots (line 13). Only after a $q2$ has succeeded, the holder sends a commit message (line 15) and executes the transaction. The system ensures that the transaction execution is atomic by blocking at the transaction slot the logs of the involved objects until all commands in the transaction have been committed. This blocking prevents any newer commands on any of the transaction objects from executing before any of the commands in the transaction. We illustrate the steps for running transactions in Figure 4.

3.2 Deadlock Prevention

In some rare cases, multiple transactions may attempt to operate on some overlapping set of objects. This makes a deadlock situation possible as each transaction may successfully steal and place leases on some items but not the others. FleetDB prevents the deadlock with a wound-wait scheme: younger transaction cannot steal from the older and has to wait, while older transaction can steal from the younger, causing it to abort. We use Hybrid Logical Clocks (HLC) [13] for transaction timestamping.

3.3 Transaction Recovery

A transaction may fail due to the transaction holder crash or network partition. In such circumstances, it is important to either finish the transaction when possible or safely abort. Since our minitransaction protocol heavily relies on Paxos, the recovery from the transaction leader failure is also based on Paxos. In particular, the recovery starts when some node steals an object involved in the transaction. At this point that new leader must also acquire the remaining transaction objects. If the transaction was successfully anchored by the previous leader, then all transaction objects will have slots and values available at the new leader, allowing to finish the transaction. In case some or all transaction commands are not anchored and the new leader is not able to learn these commands, the transaction will safely abort.

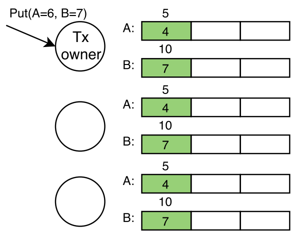
4. EVALUATION

We have implemented a prototype FleetDB in Go [10] version 1.10 on top of our earlier WPaxos implementation. We use goleveldb [9] for the underlying key-value storage engine. The database exposes a client API accessible over HTTP protocol using which we implement a REST client and a benchmark.

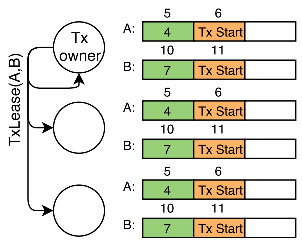
4.1 Throughput and Latency

We've looked at the overall performance of FleetDB under the access locality conditions. We used a majority access migration policy that transfers the object ownership over to the region with most object requests. We deployed the system in 3 AWS regions: California, Oregon and Virginia.

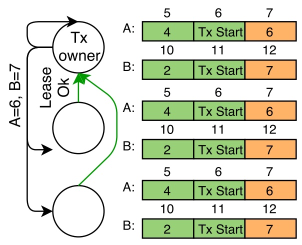
FleetDB is deployed with a pool of N objects with each object having a unique key in the range $[0, N)$. Every client can access any object in the system. The probability of a client accessing a particular object is determined by a Normal distribution $\mathcal{N}(-\frac{N}{6} + \frac{Nr}{3}, \sigma^2)$, where r is client's region ID and σ is a distribution's standard deviation. This access probability rule allow us to have some objects to be predominantly accessed in a single region, and some objects shared between the adjacent regions. 1 We ran transactional and non-transactional workloads with 50% read operations and $N = 10,000$ objects and $\sigma = 1,200$. Transactional workload



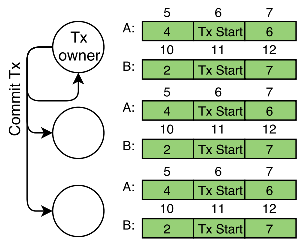
(a) Transaction holder starts multi-write transaction, all objects have been moved.



(b) Transaction holder tries to commit transaction leases on A and B.

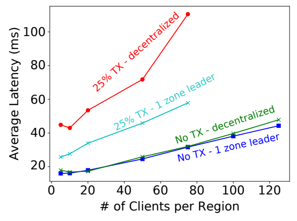


(c) $q2$ accept phase on A and B after leases committed and executed.

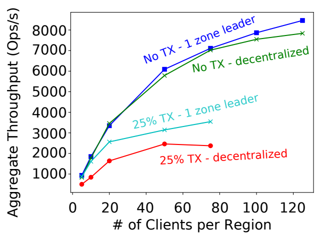


(d) When phase-2 succeeded on $q2$, transactions gets committed and executed.

Figure 4: Transaction commit protocol within a replication group.

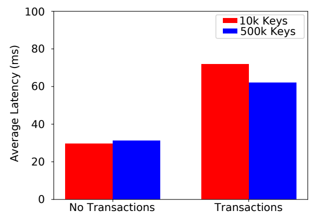


(a) Latency

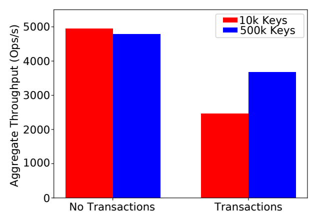


(b) Throughput

Figure 5: Throughput and latency in locality workload.

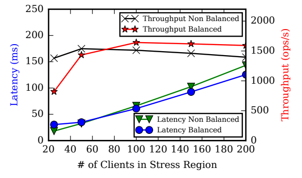


(a) Latency

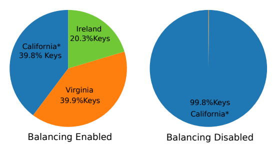


(b) Throughput

Figure 6: Latency and throughput for different key-space sizes



(a) Balanced and non-balanced FleetDB with biased region



(b) Balance in the workload with biased region (California)

Figure 7: Data balancing in FleetDB

had 25% of all operation to be write-only minitransactions on 3 objects. We used FleetDB in two configurations: first one had a dedicated leader node per zone, while the other one was fully decentralized with every node assuming the ownership of some objects. Figure 5 shows how throughput and latency changed as we increased the number of clients in each region. We measure the throughput in operations per second, and each transaction carries multiple operations.

Workload with 25% transactions shows overall lower performance largely due to the need to acquire leadership of all items in the transaction. This is especially a problem with fully decentralized FleetDB deployment, where every node can act as a leader over some keys. Despite the locality, transaction objects may be on different replicas in the same region, causing the system to undergo expensive phase-1 operation to perform a migration. The transactional performance is much better when each region has only one leader, as it completely eliminates the need to move objects within the region. Transactions also break locality adaptation of the system, as the object may have to move to the transaction owner against its access locality. Additionally, minitransactions require more messages, and tend to saturate the system faster, as we observe in Figure 5b. With this FleetDB experiment we show that decentralization may hurt the performance, as it requires too much communication to achieve our consistency guarantees. Having one leader per regions strikes a good balance between enough decentralization to provide region-local latencies for most operations without incurring unnecessary communication penalty when running transactions.

Small key-space in the above experiment also creates high object contention for transactions, further degrading the performance. When multiple concurrent transactions require the same item, it is likely for one transaction to wait or even get aborted. As we show in Figure 6, larger keys-space with identical locality is beneficial for transaction workloads, while having small impact on non-transactional performance.

4.2 Data Distribution

In this experiment we study how the system performs under the heavily skewed workload: California had 5 times more clients than Virginia and 10 more clients than Ireland. We measured how the performance observed by clients in the biased region changes as the number of clients increases. We run this experiment with uniformly random access patterns and high misbalance threshold of 10% to make sure the backpressure does not start data migration too soon. Figure 7a shows latency and throughput as the workload increases.

Balanced deployment is at disadvantage under light load, since California region can sustain the load despite the misbalance. As we add more concurrent clients, California gets saturated and slows down significantly, allowing balanced setup to perform better despite some WAN operations. Keeping proper balance of data prevents premature saturation of some nodes, by shifting the work elsewhere.

In Figure 7b we show the difference in object distribution between balanced and unbalanced deployment. In both cases, we ran a highly-skewed workload in California region and measured the data balance at the end of an experimental run. For this experiment we used a tighter, 3% misbalance threshold to achieve better overall data balancing.

5. FUTURE WORK

Improving Transaction Performance. FleetDB transaction performance is limited by the need to acquire leases and in many cases steal remote objects, breaking the access locality. Preserving the access locality is the key for improving transactional performance. One optimization involves adjusting the data-migration procedures to make it difficult to separately migrate objects that often appear in the same transaction. Another approach to improve locality in transactional workloads is to have read-only transactions (RoT) that do not force data-migration. We can achieve this with the help of hybrid logical clocks (HLC) coupled with a log or multi-version datastore. This allows the system to obtain consistent snapshots at some arbitrary HLC timestamp [4] without requiring any additional communication or synchronization.

Multi-universe FleetDB. Many databases [8, 7, 5] partition the data into independent units of storage and replication to achieve better horizontal scalability. Multi-universe FleetDB consists of such independent and partitioned deployments of FleetDB. The naïve implementation of multi-universe FleetDB loses the ability to perform transaction across universes. However, classical solutions such as two-phase commit can be placed on top of FleetDB to facilitate cross-universe transactions. Alternatively, it may be possible to compose FleetDB instances in a way to guarantee the intersection between them and extending our Paxos based protocol to perform transaction across universes by stealing the objects into the shared regions.

6. CONCLUDING REMARKS

We presented FleetDB, a globe-spanning strongly-consistent key-value datastore. Unlike previous work, FleetDB can nimbly adapt to dynamic workload changes in a fine grained manner. Migrating the ownership of an object is automatic, fast, safe (done using WPaxos phase-1), and allows reads and writes to be served locally in the migrated region. FleetDB also supports sophisticated migration policies that take into account load balancing, replication groups, and multiple simultaneous requesting regions. In addition to providing these guarantees, FleetDB manages to support atomic, serializable multi-object minitransaction.

We implemented FleetDB in Go and made it accessible as opensource project at https://github.com/acharapko/fleetdb. We experimentally showed the effectiveness of dynamic object migration by achieving average request latencies as low as 10 milliseconds for workloads with good object access locality. We also illustrated that fully decentralized solutions may not always benefit the performance. In our case, the transaction performance is better when we limit each zone to have a single dedicated leader node as it helps avoid unnecessary object migrations across leaders in the same region.

7. REFERENCES

[1] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[2] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar. Wpaxos: Ruling the archipelago with fast consensus.

[3] A. N. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin, and P. Verissimo. Scfs: A shared cloud-backed file system. In *USENIX Annual Technical Conference*, pages 169–180, 2014.

[4] A. Charapko, A. Ailijiang, M. Demirbas, and S. Kulkarni. Retrospective lightweight distributed snapshots using loosely synchronized clocks. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 2061–2066. IEEE, 2017.

[5] Cockroachdb: A scalable, transactional, geo-replicated data store. http://cockroachdb.org/.

[6] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[7] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. *Proceedings of OSDI*, 2012.

[8] Cosmos db - globally distributed, multi-model database service. https://azure.microsoft.com/en-us/services/cosmos-db/, 2018.

[9] Leveldb key/value database in go. https://github.com/syndtr/goleveldb, 2018.

[10] Google. The go programming language, 2018. https://golang.org/.

[11] H. Howard, D. Malkhi, and A. Spiegelman. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[12] S. Kadambi, J. Chen, B. F. Cooper, D. Lomax, R. Ramakrishnan, A. Silberstein, E. Tam, and H. Garcia-Molina. Where in the world is my data. In *Proceedings International Conference on Very Large Data Bases*. VLDB Endowment, 2011.

[13] S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *Principles of Distributed Systems*, pages 17–32. Springer, 2014.

[14] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[15] K. Lev-ari, E. Bortnikov, I. Keidar, A. Shraer, E. Engineering, and M. View. Modular Composition of Coordination Services. *2016 USENIX Annual Technical Conference*, 2016.

[16] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, volume 13, pages 313–328, 2013.

[17] Z. Shen, Q. Jia, G.-E. Sela, W. Song, H. Weatherspoon, and R. Van Renesse. Supercloud: A library cloud for exploiting cloud diversity. *ACM Transactions on Computer Systems (TOCS)*, 35(2):6, 2017.

[18] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)*, 22(2):255–314, 1997.