Take me to your leader! Online Optimization of Distributed Storage Configurations

Artyom Sharov* Alexander Shraer Arif Merchant Murray Stokely

sharov@cs.technion.ac.il, {shralex, aamerchant, mstokely}@google.com

Google, Inc.

ABSTRACT

The configuration of a distributed storage system typically includes, among other parameters, the set of servers and their roles in the replication protocol. Although mechanisms for changing the configuration at runtime exist, it is usually left to system administrators to manually determine the "best" configuration and periodically reconfigure the system, often by trial and error. This paper describes a new workload-driven optimization framework that dynamically determines the optimal configuration at runtime. We focus on optimizing leader and quorum based replication schemes and divide the framework into three optimization tiers, dynamically optimizing different configuration aspects: 1) leader placement, 2) roles of different servers in the replication protocol, and 3) replica locations. We showcase our optimization framework by applying it to a large-scale distributed storage system used internally in Google and demonstrate that most client applications significantly benefit from using our framework, reducing average operation latency by up to 94%.

1. INTRODUCTION

Storage is changing from being mostly in-house and local to become a fully globally-distributed service. Cloud storage services such as Amazon S3, Microsoft Azure, and Google Cloud Storage, form the underpinnings of many Internet services with clients distributed all over the world. Typically, applications need continuous availability and reasonably good data access latency, which translates into storing multiple data replicas in different geographic locations.

Distributed storage systems usually provide consistency across replicas of data (or metadata) using serialization or conflict resolution protocols. Distributed atomic commit or consensus-based protocols are often used when strong consistency is required, while simpler protocols suffice when replicas need only preserve weak or eventual consistency. Such protocols typically define multiple possible roles for the replicas, such as a leader or master replica coordinating updates, and appoint some of the replicas to participate in the commit protocol. The performance of such a system depends on the configuration: how many replicas, where they are located, and which roles they serve; for optimal performance, the configuration must be tuned to the workloads.

Different applications using the same storage service may have completely different workloads, for example a logging system may use the storage mostly for writes and have relatively few clients, while an application responsible for access control may be read heavy. The workloads can be extremely variable, both in the short term — for example, load may come from different parts of the world at different times of the day for a social application — and in the long term — the administrators of the service may reconfigure their servers periodically, causing different load patterns. Long term workload variation could also be due to organic changes in the demands on the Internet service itself; for example, if a shopping service becomes more popular in a region, its demands on the underlying storage system may shift.

The cloud storage service must adapt to these changes seamlessly. In fact, elasticity is an integral part of cloud computing and one of its most attractive premises. Reconfiguring a distributed storage system at run-time while preserving its consistency and availability is a very challenging problem and misconfigurations have been cited as a primary cause for production failures [29]. Due to its practical significance the problem has received abundant attention in recent years both in academia and in the industry (see Section 7), focusing mainly on the design and implementation of efficient and non-disruptive mechanisms for reconfiguration. Yet, little insight exists on how to set policies and use reconfiguration mechanisms in order to improve system performance. For example, dynamic reconfiguration APIs have recently been added to the Apache ZooKeeper distributed coordination system [25, 19] and users have since been asking for automatic workload-based configuration management, e.g. [6]. At Google, site reliability engineers (SREs) are masters in the black art of determining deployment policies and tuning system parameters. However, hand-tuning a cloud storage system that supports hundreds of distinct workloads is difficult and prone to misconfigurations.

We describe the design and implementation of a workload-driven framework for automatically and dynamically optimizing the replication policy of distributed storage systems. We showcase our framework by optimizing a large-scale dis-

*Computer Science Department, Technion, Israel. Work done as part of a summer internship in Google's Distributed Storage Analytics team.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

tributed storage system used internally in Google. In this work we focus on operation latency as the main optimization objective. We assume that the various aspects of load distribution and balancing are addressed by the underlying storage system, as is the case for our storage system and for many other massively scalable systems.

Section 2 contains an overview of our storage model. In short, users define databases which are then partitioned and replicated [1, 10, 14, 15, 16, 17, 26]. The replication policy is defined by the database administrator, usually an SRE responsible for a specific client application. For example, if most writers are expected to reside in western Europe, the administrator will likely place replicas in European locations, and make some of them voters so that a quorum required to commit state changes can be collected in Europe without using expensive cross-continental network links. One of the voters (elected dynamically) acts as a leader and coordinates the replication.

Since leaders are involved in many of the operations, their location significantly affects latency. Section 3 describes the first tier of our optimization framework, which, given replica locations and roles, optimizes leader placement. Our algorithm uses a detailed characterization of the relevant operations supported by the storage system. Unlike many of the previously proposed methods, such as placing the leader close to the writers (previously suggested, for example, in the context of Google Megastore or Yahoo! PNUTS), which may work for one workload but not for another, we leverage Google's monitoring infrastructure to dynamically and accurately track the global workload of each database, as well as network latencies. Our evaluation, using both simulations with production workloads and production experiments, presented in Section 6, shows that our method is fast, accurate and significantly outperforms previously proposed "common sense" heuristics. We show that it results in a substantial speed-up for the vast majority of databases in our system, halving the operation latency for 17% of the databases and reducing the latency by up to 94% for others.

The goal of the second optimization tier, presented in Section 4 is, given replica locations, to determine the best replica roles in addition to the optimal leader placement. While the number of voter replicas is usually determined based on availability requirements, their locations are flexible. For example, if 2 simultaneous replica failures must be tolerated, 5 voting replicas will be used and our system determines which 5 replicas are to be voters based on the observed workload. Usually in such systems the leader is one of the voters and thus optimizing voter placement not only affects the latency of commits but also of other operations involving the leader. Our evaluation in Section 6 confirms the benefits of dynamic role assignment for both write and read heavy workloads, achieving a speed-up of up to 50% on top of the first optimization tier, for some databases.

Finally, in Section 5 we present two new algorithms used to determine replica locations (in addition to choosing replica roles and a leader), which constitutes our third optimization tier. For example, if a database suddenly experiences a surge in client operations coming from Asia, we will detect the change in workload and identify the best replica locations to minimize latency. The algorithm additionally determines which among the replicas should be voters and which voter should be leader. Section 6 shows that the two algorithms are near optimal and identifies workloads where each of the algorithms is preferable. We also show how these algorithms can be used to determine the desired number of replicas.

The optimization algorithms are dynamic – the best configuration for a given time interval is determined considering the workload in previous time intervals, weighted according to recency. The algorithms can thus react to workload changes without being overly sensitive to spikes.

In summary, the main contribution of this paper is the design, implementation and evaluation of an optimization framework for dynamically optimizing the replication policy of leader-based storage systems. Our system frees administrators from manually and periodically trying to adjust database replication based on the currently observed workloads, which is a very difficult task since such systems usually support hundreds of distinct workloads and multiple operation types each involving a series of message exchanges. Our system uses monitoring information to perform the optimization automatically, allowing a detailed and separate consideration of each workload. Our evaluation demonstrates dramatic latency improvements for a production distributed storage system used in Google.

2. STORAGE MODEL

We assume a common distributed storage model combining partitioning and replication to achieve scalability and fault tolerance: users (administrators) define databases, each database is sharded into multiple partitions, and each partition is replicated separately [1, 10, 14, 15, 16, 17, 26]. We call these partitions *replication groups*. A single replication policy, defined by the database administrator, governs all replication groups in a database and determines the configuration of each group – the number of replicas, their locations and their roles in the replication protocol[1].

A replica can be either *read-write* or *read-only*. Both are full replicas and can serve reads locally. Whereas *read-write* replicas vote on commits, *read-only* replicas are notified of state changes only after they occur. Such categorization exists in many systems, such as *acceptors* and *learners* in Paxos [21] or *participants* and *observers* in ZooKeeper [19]. One of the voting replicas in every replication group is chosen as the group *leader*. The leader is chosen dynamically and when it fails, a different leader is elected. Leaders are typically responsible for coordinating replication in their group, for locking and concurrency control to support transactions, and participate in transaction involving multiple groups.

We denote the voting replicas of a group g by $\mathcal{V}(g)$. Recall that all groups in a database share the same configuration, and hence are replicated across the same locations (i.e., clusters or datacenters). Since we are solely interested in the location of the replicas, we simply use $\mathcal{V}(db)$ to refer to $\mathcal{V}(g)$ for any group g in a database db. Similarly, $\mathcal{R}(db)$ refers to the set of all replicas (*read-write* and *read-only*). We omit the index db and g when it is clear from the context. A client is denoted by c, the replica closest to the client (in terms of network latency) by $nearest(c, \mathcal{R})$ (note that it is taken from the set $\mathcal{R}$), and the leader of a group g by $leader(g)$. Finally, we denote the set of client locations by $\mathcal{C}$ and the universe of potential replica locations by $\mathcal{S}$ (note that $\mathcal{R} \subseteq \mathcal{S}$).

[1]The storage system we used for evaluation supports multiple replication policies per database, however few databases make use of this feature and all such databases perform manual configuration tuning.

Operations are invoked by clients and multiple operations may be executed together as part of transactions. Transactions may involve one or more groups within a database. Next, we identify five representative operations, most commonly supported (under various names) in replicated distributed storage systems. Our framework optimizes operation latency. As such it requires knowledge of the flow of each operation. Hence, for each operation, we give a possible message flow which we use to showcase our optimization framework in the following sections. This flow is simplified and actual implementations may use various optimizations, which may in turn require adjustments to the optimization formulation. However, such optimizations are orthogonal to the contributions of this paper.

weak read. A read from one of the replicas (typically the replica closest to the client). As replicas may lag behind the leader, the read can return stale values. Typically, a client c sends a request to $\mathit{nearest}(c, \mathcal{R})$, which replies with its local copy of the requested data.

bounded read. This read typically includes a bound on the staleness of returned values. If $\mathit{nearest}(c, \mathcal{R})$ is not sufficiently up to date, it forwards the request to the leader $\mathit{leader}(g)$, which responds with the latest version of the data. Once updated, $\mathit{nearest}(c, \mathcal{R})$ responds to the client.

strong read. This read returns the last committed value, typically returned by the leader. In some systems, clients read directly from the leader; in others, the read is relayed through $\mathit{nearest}(c, \mathcal{R})$.

For example, Yahoo! PNUTS [15] supports all three types of reads. Apache ZooKeeper, as well as Amazon SimpleDB [2], DynamoDB [1], MongoDB [4], and many others, support only the weak (sometimes called "eventually consistent") and the strong (sometimes called "consistent") read types. Often, instead of providing an explicit API for bounded reads, systems such as ZooKeeper make guarantees about the maximum allowed staleness of replicas. The choice among the reads may be explicitly made by the user or automatically done by the storage system based on higher level user preferences (such as in MongoDB or Riak [3]). Next, we define two state update operaitons:

single-group transaction. This is a state-changing operation or transaction that involves the data of a single replication group g. It is executed by $\mathit{leader}(g)$ on behalf of a client c, and, for high availability, requires $\mathit{leader}(g)$ to persist the state changes on a quorum (usually a majority) of voting replicas. To this end, $\mathit{leader}(g)$ sends messages to the voting replicas $\mathcal{V}(g)$ and waits for a quorum to respond. If the minimal required set of affirmative responses is collected, the transaction commits and a COMMIT message is sent to the client as well as to the group replicas.

multi-group transaction. When committing a transaction involving data from multiple groups, a distributed atomic commit protocol must be executed across the groups in order to either commit or abort the transaction atomically in all involved groups. Typically, this protocol is two-phase commit and the leader of one of the groups involved in the transaction acts as the coordinator while the others are participants. In order to be fault tolerant, every step of the protocol must be agreed-upon by the members of each group, and not only by its leader. For the the purposes of this paper, we assume the following simple protocol: the coordinator leader $\mathit{leader}(g)$ broadcasts a PREPARE message to participant leaders. Each participant leader $\mathit{leader}(g')$ checks locally whether the transaction may be committed and if so persists its intention to commit to a quorum of voters $\mathcal{V}(g')$. It then sends an ACK message to $\mathit{leader}(g)$. Once all participants have acked, $\mathit{leader}(g)$ commits the transaction by persisting it to a quorum of $\mathcal{V}(g)$ and responds to the client. It then sends COMMIT messages to the participant leaders which in turn commit the operation in their respective groups by persisting the state-changes to a quorum.

For example, VoltDB [26], HyperDex [17], DynamoDB and Microsoft Orleans [14] support both transactions types, whereas PNUTS [15], ZooKeeper [19] and many NoSQL stores only support variants of single-group transactions.

3. TIER 1: LEADER PLACEMENT

In this work, we focus on optimizing operation latency. From the description in Section 2, it is easy to see that operation latency is affected by the location of the client, location of the replica closest to the client, locations of the *read-write* replicas, and finally, by the choice of the leader among the *read-write* replicas. Our first algorithm, described in this section, optimizes leader placement without modifying server roles and locations ($\mathcal{V}$ and $\mathcal{R}$) given by the database andministrator.

Consider a database with one thousand groups and five *read-write* replicas per group, each of which can become leader. It may seem that in order to optimize leader placement for such database we have to consider 5000 different placement configurations. Unfortunately, this is not the case. Multi-group transactions involve several groups, whose leaders execute a distributed atomic commit protocol. Changing the placement of one group leader may therefore impact the optimal placement of the leaders of other groups. In fact, for our numerical example, in the worst case the number of different placement options is 5^{1000}.

In order to achieve a practical solution we must reduce the solution space drastically. We chose to optimize leader placement on the granularity of a database instead of a single group. Since all groups in a database are replicated in the same way, this method reduces the solution space to one of the *read-write* replica locations for this database. In practice, while our optimization algorithm outputs a single location, the storage system may place the different leaders of groups belonging to the database close to this location, taking various constraints related to load balancing, failure diversity, etc., into account. We denote the leader location produced by our algorithm for database db in the i-th time interval by $\lambda_{db}^{(i)}$.

For simplicity, we focus on optimizing the average operation latency; this metric is generalized in Section 3.1. Intuitively, to minimize average operation latency we compute it for every possible leader location (any *read-write* replica can potentially become leader), and then choose the location yielding minimum average latency. Assuming that different clients in the same cluster experience similar latencies when communicating with servers, we logically group clients within each cluster and consider "client clusters" rather than individual clients in our analysis.

Since different operation types may have different latency profiles, we compute a weighted average. Specifically, at interval i we (a) determine the average latency $\mathtt{t}_{\alpha,c}^{(i)}(\ell)$ of each type of operation α from every client cluster c, for each potential leader location ℓ ranging over the set $\mathcal{V}(db)$ of *read-*

write replica locations as defined for database *db*, and (b) quantify the number of operations $\mathsf{n}^{(i)}_{\alpha,c}$ of each type α for each client cluster c. Finally, we choose a leader $\lambda^{(i)}_{db}$ that minimizes the following expression (for simplicity we omit the denominator of the weighted average):

$$\lambda^{(i)}_{db} = \arg\min_{\ell \in \mathcal{V}(db)} \{score^{(i)}(\ell)\} ,$$

where $score^{(i)}(\ell) = \sum_{\alpha,c} \mathsf{t}^{(i)}_{\alpha,c}(\ell) \cdot \mathsf{n}^{(i)}_{\alpha,c}$. The location $\lambda^{(i)}_{db}$ can then serve as prediction for the optimal location $\lambda^{(i+1)}_{db}$ in the $(i+1)$-st time interval. Next, we calculate $\mathsf{t}^{(i)}_{\alpha,c}(\ell)$ by considering the flow of different operation types described in Section 2. We assume that leader identities are exposed to the clients, and hence clients send strong reads and transactions directly to the relevant leaders.

Denote the average roundtrip-time latency between nodes a and b in time interval i by $\mathtt{rtt}^{(i)}_{a,b}$. For weak reads, the average latency is simply $\mathtt{rtt}^{(i)}_{c,nearest(c,\mathcal{R})}$, and similarly for strong reads, given a candidate leader location ℓ, we get $\mathtt{rtt}^{(i)}_{c,\ell}$. For bounded reads, using the linearity of expectation, we have that the average latency is $\mathtt{rtt}^{(i)}_{c,nearest(c,\mathcal{R})} + \mathtt{rtt}^{(i)}_{nearest(c,\mathcal{R}),\ell}$, which corresponds to a roundtrip between the client and the nearest replica and another roundtrip between that replica and the leader. For simplicity, we do not distinguish here between bounded reads served locally versus those forwarded to the leader, but do implement the distinction. Calculating single-group transaction latency is slightly more complex, as it requires the leader to wait for a majority of voter responses, i.e., for the median fastest response. Unfortunately, the median and average operations don't commute in general. We found, however, that cross-cluster latency distributions tend to be very narrow around their respective mean values and overlap only for tail latencies. They can therefore be approximated as fixed values for computing the average of the median. The median of average `rtt` latencies is therefore a very good approximation for the average of median `rtt` latencies. Hence, the average operation latency of single-group transactions can be estimated as $\mathtt{rtt}^{(i)}_{c,\ell} + q^{(i)}_{\ell}$, where $q^{(i)}_{\ell} = median_{v \in \mathcal{V}(db)}\{\mathtt{rtt}^{(i)}_{\ell,v}\}$.

The computation for multi-group transactions is made complex by the fact that every multi-group transaction involves a different set of groups. Fortunately, our decision to optimize per database (rather than per group) considerably simplifies the calculation. First, since we are looking for the best single location $\lambda^{(i)}_{db}$ for all group leaders in the database, we only need to consider assignments which result in the same placement for all group leaders and the latency between these colocated leaders is effectively negligible. Second, since all groups in a database usually share the same configuration, the set of latency distributions between the leaders and their respective voters is the same for all group leaders in the database. To leverage this, let us recap the flow of multi-group transactions: The client sends a message to one of the group leaders and waits for a response. The average roundtrip latency between the client and the leader is simply $\mathtt{rtt}^{(i)}_{c,\ell}$. The leader (coordinator) then contacts other leaders (participants). Since all leaders are in the same location ℓ this latency is negligible. Each participant leader then sends a message to its voters and waits for a majority of responses, which takes $q^{(i)}_{\ell}$. Then, participant leaders contact the coordinator leader. Finally, the coordinator leader commits the transaction by sending it to the voters of its group and waits for a majority of responses, which again takes $q^{(i)}_{\ell}$, on average. Overall, the average latency of a multi-group commit can be estimated as $\mathtt{rtt}^{(i)}_{c,\ell} + 2 \cdot q^{(i)}_{\ell}$. To summarize, the score of a candidate leader location $\ell \in \mathcal{V}$, is given by Equation 1.

Both $\mathsf{n}^{(i)}$ and average $\mathtt{rtt}^{(i)}$ latencies are measured for the last observed (i-th) time interval and, as presented thus far, used to predict $\lambda^{(i+1)}_{db}$. Observe that there is a tradeoff between the chosen time interval length and the accuracy of prediction. By choosing a short interval (e.g., one minute) our solution becomes very sensitive to workload spikes, deteriorating prediction accuracy. By using long intervals (e.g., one day) the prediction may be more accurate yet it may average out potentially interesting workload changes (such as diurnal patterns). Instead of attempting to pick the "best" time interval length (which may vary across different databases), in Equation 2 we introduce a *decay* parameter τ and compute *score* based on multiple past intervals (and not just the i-th interval), weighting them according to recency using an exponential moving average (for simplicity we again omit the denominator of the weighted average).

$$\begin{aligned}
&\mathsf{score}^{(i)}(\ell, \mathcal{R}, q^{(i)}_{\ell}) = \sum_{c \in \mathcal{C}} [(\mathsf{n}^{(i)}_{weak\ read,c} \cdot \mathtt{rtt}^{(i)}_{c,nearest(c,\mathcal{R})} \\
&+ \mathsf{n}^{(i)}_{bounded\ read,c} \cdot (\mathtt{rtt}^{(i)}_{c,nearest(c,\mathcal{R})} + \mathtt{rtt}^{(i)}_{nearest(c,\mathcal{R}),\ell}) \\
&+ \mathsf{n}^{(i)}_{strong\ read,c} \cdot \mathtt{rtt}^{(i)}_{c,\ell} + \mathsf{n}^{(i)}_{single\text{-}group\ transaction,c} \cdot (\mathtt{rtt}^{(i)}_{c,\ell} + q^{(i)}_{\ell}) \\
&+ \mathsf{n}^{(i)}_{multi\text{-}group\ transaction,c} \cdot (\mathtt{rtt}^{(i)}_{c,\ell} + 2 \cdot q^{(i)}_{\ell})]
\end{aligned} \tag{1}$$

$$\begin{aligned}
&\mathsf{agg_score}^{(i)}(\ell, \mathcal{R}, q^{(i)}_{\ell}) = \\
&= \frac{1}{\tau} \cdot \mathsf{agg_score}^{(i-1)}(\ell, \mathcal{R}, q^{(i-1)}_{\ell}) + \mathsf{score}^{(i)}(\ell, \mathcal{R}, q^{(i)}_{\ell}) \\
&\lambda^{(i)}_{db} = \arg\min_{\ell \in \mathcal{V}} \{\mathsf{agg_score}^{(i)}(\ell, \mathcal{R}, q^{(i)}_{\ell})\}
\end{aligned} \tag{2}$$

Interval $i = 1$ is the first interval considered for our analysis and $\mathsf{agg\_score}^{(0)}(.,.,.) = 0$. Note that Equation 1 can be generalized to account for multiple replication policies (configurations) per database. However a straightforward extension is exponential in the number of configurations, as it has to account for multi-group transactions involving every possible subset of configurations, and every possible leader placement in each configuration. The design of a more efficient method is an interesting topic for future research.

3.1 Optimizing Tail Latency

For some users, optimizing tail latency is more important than optimizing for the mean. Although currently our system optimizes for average latency, in the future, we plan to extend it to allow database owners to specify the desired percentile and optimize for that percentile when determining the best configuration for the database.

When considering tail latency, we can no longer use the nice linearity properties we leveraged so far. Below we show how to extend the score calculation in Equation 1. As input, instead of the average roundtrip-time latencies, we now need to know the roundtrip-time latency distribution $H_{a,b}$ between each pair of locations a and b. We assume that these distributions are independent. For simplicity, assume that latencies are discretized as multiples of 1ms.

When computing the latency of each operation type, instead of summing up averages we should compute the distribution of the sum of random variables. As an example, consider the simple case of a bounded read, which travels from a client c to the closest replica $nearest(c, \mathcal{R})$, then from $nearest(c, \mathcal{R})$ to the leader ℓ and back all the way to the client. In order to find the latency distribution of this operation we perform a discrete convolution $H_{c,nearest(c,\mathcal{R})} * H_{nearest(c,\mathcal{R}),\ell}$ as follows:

$$Pr(\mathtt{t}^{(i)}_{bounded\ read,c}(\ell) = x) =$$
$$Pr(\mathtt{rtt}_{c,nearest(c,\mathcal{R})} + \mathtt{rtt}_{nearest(c,\mathcal{R}),\ell} = x) =$$
$$\sum_{k=m}^{x} Pr(\mathtt{rtt}_{c,nearest(c,\mathcal{R})} = k, \mathtt{rtt}_{nearest(c,\mathcal{R}),\ell} = x - k) =$$
$$\sum_{k=m}^{x} Pr(\mathtt{rtt}_{c,nearest(c,\mathcal{R})} = k) \cdot Pr(\mathtt{rtt}_{nearest(c,\mathcal{R}),\ell} = x - k),$$

where m denotes the minimum possible value of $\mathtt{t}^{(i)}_{bounded\ read,c}(\ell)$ and $\mathtt{rtt}$ is the random variable corresponding to the latency (rather than the average latency). Once the distribution of the sum has been computed, the required percentile can be taken from this distribution.

Computing the distribution of the quorum latency is more complex. The simplest numeric method is to perform a Monte Carlo simulation, repeatedly sampling the distributions $H_{\ell,v}$ for $v \in \mathcal{V}$ and computing the median latency each time. For an analytical solution, observe that the leader needs to collect $majority - 1$ responses from other servers, where $majority \leftarrow \lceil \frac{|\mathcal{V}|+1}{2} \rceil$ and assume that the leader's own response arrives faster than any other response. The CDF of the maximum response time from any set of read-write replicas is simply the product of the CDFs of response time for the individual replicas. For example, for 3 *read-write* replicas ℓ, v and w where ℓ is the candidate leader:

$$Pr(\mathtt{max}(\mathtt{rtt}_{\ell,v}, \mathtt{rtt}_{\ell,w}) \le x) = Pr(\mathtt{rtt}_{\ell,v} \le x, \mathtt{rtt}_{\ell,w} \le x)$$
$$= Pr(\mathtt{rtt}_{\ell,v} \le x) \cdot Pr(\mathtt{rtt}_{\ell,w} \le x)$$

We can therefore construct the CDF of maximum response time for every subset of the read-write replicas. From these, using the inclusion-exclusion principle [5], we can compute the probability of the event that at least one subset of the read-write replicas, of cardinality $majority - 1$, has maximum response latency less than x, for each x. But this event is equivalent to the event that the quorum's response time is less than x, hence it gives us the CDF of the quorum response time. Continuing our example for 3 *read-write* replicas, we get:

$$Pr(q^{(i)}_{\ell} < x) = Pr(\mathtt{rtt}_{\ell,v} \le x) + Pr(\mathtt{rtt}_{\ell,w} \le x)$$
$$- Pr(\mathtt{max}(\mathtt{rtt}_{\ell,v}, \mathtt{rtt}_{\ell,w}) \le x)$$

4. TIER 2: LEADER AND REPLICA ROLES

Our tier-1 optimization algorithm, described in the previous section, optimizes leader placement while keeping $\mathcal{V}$ and $\mathcal{R}$ fixed. In this section, we introduce our tier-2 optimization algorithm that determines the best voter locations $\mathcal{V}$ from $\mathcal{R}$ as well as the best leader location from $\mathcal{V}$. This algorithm does not modify $\mathcal{R}$ (this is the topic of Section 5). As we explain next, for reasons of efficiency we do not directly use the method described in Section 3, but rather the optimization objective in Equation 2.

In order to evaluate and compare different configurations we must take into account the best leader location possible with the configuration. Given a configuration with $|\mathcal{R}|$ replica locations, a brute-force approach (shown in Algorithm 1) is to enumerate $\binom{|\mathcal{R}|}{|\mathcal{V}|}$ configurations, corresponding to different possible subsets of $num\_voters = |\mathcal{V}|$ *read-write* replicas, and for each one find the optimal leader using the algorithm given in Section 3. Recall that our tier-1 algorithm evaluates every *read-write* replica as a potential leader by considering the resulting cost for every client cluster. For a typical database in the production system considered in Section 6, this algorithm would result in more than 26 million computations per database and time interval.

Algorithm 1 Brute-force algorithm for tier-2.

1: **procedure** *tier-2-brute-force($\mathcal{R}$, num_voters)*
2: *for each* set $V \in \mathcal{R}^{num\_voters}$
3: *for each* replica $\ell \in V$
4: $q^{(i)}_{\ell} = median_{v \in V}\{\mathtt{rtt}^{(i)}_{\ell,v}\}$
5: $score_{\ell} \leftarrow \mathsf{agg\_score}^{(i)}(\ell, \mathcal{R}, q^{(i)}_{\ell})$ (Equation 2)
6: $\lambda_V \leftarrow \arg\min_{\ell \in V}\{score_{\ell}\}$
7: $score_V \leftarrow \mathsf{agg\_score}^{(i)}(\lambda_V, \mathcal{R}, q^{(i)}_{\lambda_V})$
8: $V_{opt} = \arg\min_V\{score_V\}$
9: **return** $(\lambda_{V_{opt}}, V_{opt})$ // Optimal leader and quorum

We propose a much more efficient alternative (Algorithm 2): instead of picking *read-write* replicas first and then the leader among them, we reverse the decision order and eliminate configurations that are clearly sub-optimal due to their poor leader score. More precisely, for every candidate leader ℓ, out of all the replica locations $\mathcal{R}$ (not just *read-write*), we find the k-th smallest $\mathtt{rtt}$ to other replicas, where $k = \lceil \frac{num\_voters+1}{2} \rceil$, and use it to calculate $score^{(i)}$. We then choose the leader $\lambda^{(i)}_{db}$ for which $score^{(i)}$ is minimized. Finally, we compute the set of voters for leader $\lambda^{(i)}_{db}$ by picking the *num_voters* replicas with minimum $\mathtt{rtt}$ from the leader (in fact, we could take just the k fastest replicas and pick the remaining $num\_voters - k$ replicas arbitrarily since quorum latency is determined by the fastest majority of votes).

Algorithm 2 Efficient algorithm for tier-2.

1: **procedure** *tier-2-efficient($\mathcal{R}$, num_voters)*
2: *for each* replica $\ell \in \mathcal{R}$
3: $q^{(i)}_{\ell} \leftarrow \lceil \frac{num\_voters+1}{2} \rceil$-th smallest $\mathtt{rtt}^{(i)}_{\ell,r}$, $r \in \mathcal{R}$
4: $score_{\ell} \leftarrow \mathsf{agg\_score}^{(i)}(\ell, \mathcal{R}, q^{(i)}_{\ell})$ (Equation 2)
5: $\lambda = \arg\min_{\ell \in \mathcal{R}}\{score_{\ell}\}$
6: $max\_voter\_rtt \leftarrow num\_voters$-th smallest $\mathtt{rtt}_{\lambda,r}$, $r \in \mathcal{R}$
7: $U_{\lambda} \leftarrow k\text{-}closest(\lambda, num\_voters, max\_voter\_rtt, \mathcal{R})$
8: **return** (λ, U_{λ})

9: **procedure** *k-closest(ℓ, num_voters, max_latency, $\mathcal{R}$)*
10: $P_{<} \leftarrow \{r \in \mathcal{R} \mid \mathtt{rtt}_{\ell,r} < max\_latency\}$
11: $P_{=} \leftarrow \{r \in \mathcal{R} \mid \mathtt{rtt}_{\ell,r} = max\_latency\}$
12: **return** $P_{<} \cup \{(num\_voters - |P_{<}|) \text{ elements from } P_{=}\}$

To see why Algorithm 2 returns the optimal solution, let us consider the leader $\lambda_{V_{opt}}$ and set of voters V_{opt} returned by Algorithm 1. Algorithm 2 evaluates $\lambda_{V_{opt}}$ as candidate leader (line 2). Optimality follows from the fact that Algorithm 2 chooses voters for $\lambda_{V_{opt}}$ from a larger set of candidates (since $V_{opt} \subseteq \mathcal{R}$) and therefore quorum latency for

$\lambda_{V_{opt}}$ is necessarily smaller or equal in Algorithm 2 (line 3) compared to Algorithm 1 (line 4).

Complexity of Algorithm 2. It takes $O(\mathcal{R})$ to consider every replica as candidate leader and, for each candidate, another $O(\mathcal{R})$ to find the k-th smallest `rtt` to other replicas (using a worst-case linear time selection algorithm). Invocation of *k-closest* in the last step of the algorithm takes $O(\mathcal{R})$, overall yielding $O(\mathcal{R}^2)$ complexity, clearly better compared to the exponential complexity of Algorithm 1. For a typical database in our storage system, Algorithm 2 requires roughly 2.8 thousand computations per database and time interval, which is 4 orders of magnitude less than brute force.

Failure Diversity. Since progress in quorum-based replicated systems is usually guaranteed only as long as a quorum of *read-write* replicas is available and can communicate in a timely manner, *read-write* replicas must be located in different failure domains. To account for this fact, we slightly modify our algorithm as follows: instead of choosing the k-th smallest `rtt` for each leader candidate out of a set of $\mathcal{R}$ replicas, we pre-process the set for each candidate leader by bucketing replicas according to the different failure domains and choosing the replica with smallest `rtt` from the candidate leader as a representative replica from each domain (bucket), filtering out the remaining replicas in each domain. We then select the k-th smallest `rtt` from the reduced set of replicas. The linear time pre-processing does not increase the complexity of our algorithm and may potentially speed up the execution of *k-closest*. Note that in practice there may be multiple different diversity constraints that one may want to consider, and further adjustments may be required. Our system recommends an alternative set of replicas for a given database preserving the failure diversity level currently met by the database configuration. In the future, we plan to offer clients several alternative configurations trading off increased fault tolerance and operation latency.

5. TIER 3: REPLICA LOCATIONS, ROLES, AND LEADER

In this section we expand the scope of our optimization and present two efficient algorithms to select the best set of replicas $\mathcal{R}$ from the possible locations $\mathcal{S}$, a set of voters $\mathcal{V} \subseteq \mathcal{R}$ and the best leader from $\mathcal{V}$ (one of the algorithms makes direct use of Algorithm 2). Unlike the algorithms in Sections 3 and 4, which find the optimal solution, the algorithms in this section are heuristics and we compare the achieved solutions with the optimum in Section 6.4. The algorithms in this section take the desired number of replicas as a parameter. How to determine this number is further explored in Section 6.4.

Similarly to Section 4, a brute force algorithm is straightforward, but exponential in complexity and highly impractical: such an algorithm could consider every possible subset of replicas $\mathcal{R} \subseteq \mathcal{S}$, execute *tier-2-efficient*($\mathcal{R}$, $|\mathcal{V}|$), and choose the best combination of leader, voters and replicas.

Recall that in Section 4 we reduced the search space by finding the best score for every possible leader in linear time. This approach yielded an optimal solution because the locations of all replicas were fixed and thus, for each client, $nearest(c, \mathcal{R})$ and hence also $\texttt{rtt}_{c,nearest(c,\mathcal{R})}$ did not depend on the choice of the set of voters or the leader. Our goal was to minimize the rest of the expression in Equation 1. Here, on the other hand, optimizing the function *nearest* is part of the problem. Given the location ℓ of a candidate leader, we cannot, for example, greedily choose the closest replicas to ℓ to be voters since it may be better to trade off quorum latency for decreasing the latency between the clients and their closest replicas (e.g., for a read-heavy database).

Our problem is a variant of non-metric facility location, which is known to be NP-Complete. We present two efficient heuristics for choosing replica locations, both make use of Algorithm 3, a variant of the weighted K-Means algorithm. The algorithm assigns a weight w_c to each client cluster c based on the total number of operations performed by c:

$$w_c = \sum_{\alpha} \mathsf{n}^{(i)}_{\alpha,c} = \mathsf{n}^{(i)}_{weak\ read,c} + \mathsf{n}^{(i)}_{bounded\ read,c} + \mathsf{n}^{(i)}_{strong\ read,c} + \mathsf{n}^{(i)}_{single\text{-}group\ transaction,c} + \mathsf{n}^{(i)}_{multi\text{-}group\ transaction,c}$$

The goal of Algorithm 3 is to find a set of servers G ($G \subseteq \mathcal{S}$) such that $\texttt{cost}(G)$ is minimized:

$$\texttt{cost}(G) = \sum_{c \in \mathcal{C}} w_c \cdot \texttt{rtt}^{(i)}_{c,nearest(c,G)} \quad (3)$$

Algorithm 3 gets an initial set of replica locations (centroids) L_{fixed} and the total desired number of locations *num_replicas* as parameters. First, we choose initial locations for the remaining centroids (lines 6-10) by placing them close to the "heaviest" client clusters (according to w_c). Each centroid location g defines a set of client clusters $\mathcal{C}_g$ for which g is the nearest centroid (line 15). The remainder of Algorithm 3 tries to adjust the position of each centroid g in a way that minimizes cost (weighted roundtrip-time) for clients in $\mathcal{C}_g$. Note that the centroids in L_{fixed} are not being moved. The algorithm terminates returning the set of centroids G once there is no sufficient improvement in the total cost, i.e., $\texttt{cost}(G)$.

Algorithm 3 Weighted K-Means for choosing replica locations.

```
1: // L_fixed: set of fixed replica locations, which can't be moved
2: // num_replicas: total number of replicas to be placed
3: procedure weighted-k-means(L_fixed, num_replicas)
4:    // pick initial centroids
5:    G ← L_fixed
6:    sort all client clusters c ∈ C by descending w_c
7:    while |G| < num_replicas and more client clusters remain
8:       c ← next client cluster in C
9:       if nearest(c, S) ∉ G then
10:         add nearest(c, S) to G
11:   new_cost ← cost(G)
12:   repeat
13:      prev_cost ← new_cost
14:      // cluster clients according to nearest centroid
15:      ∀g ∈ G let C_g ← {c | g = nearest(c, G)}
16:      // attempt to adjust centroids
17:      for each g ∈ G \ L_fixed
18:         g' ← v ∈ S s.t. Σ_{c∈C_g} w_c · rtt^(i)_{c,v} is minimized
19:         update centroid g to g'
20:      new_cost ← cost(G)
21:   until new_cost − prev_cost < threshold
22:   return G
```

Recall that our goal is not only to find good replica locations, but also find a quorum and a leader. Our two new algorithms differ in the order in which they perform these tasks. Algorithm 4 first places all replicas in "strategic" locations using Algorithm 3 and then invokes Algorithm 2 to determine the leader and voters from within the replicas.

Algorithm 5, on the other hand, first sets the leader and a quorum of voters and then invokes Algorithm 2 to place

Algorithm 4 Algorithm KQ.

procedure *KMeans-Quorum*(*num_replicas*, *num_voters*)
  $G \leftarrow$ *weighted-k-means*($\emptyset$, *num_replicas*)
  $(\lambda, V) \leftarrow$ *tier-2-efficient*(G, *num_voters*)
  // Return the leader, set of voters and set of replicas
  return (λ, V, G)

the remaining replicas close to the clients. More specifically, we go over all possible leaders locations in $\mathcal{S}$ and find the best quorum for this leader. This quorum is then considered as centroids during the invocation of Algorithm 3 but these centroids are pinned down and not moved by the algorithm.

Algorithm 5 Algorithm QK.

1: **procedure** *Quorum-KMeans*(*num_replicas*, *num_voters*)
2: $majority \leftarrow \lceil \frac{num\_voters+1}{2} \rceil$
3: $minority \leftarrow num\_voters - majority$
4: *for each* replica $\ell \in \mathcal{S}$
5: $q_\ell^{(i)} \leftarrow majority$-th smallest $\mathtt{rtt}_{\ell,s}^{(i)}, s \in \mathcal{S}$
6: $Q_\ell \leftarrow$ *k-closest*$(\ell, majority, q_\ell^{(i)}, \mathcal{S})$
7: $G_\ell \leftarrow$ *weighted-k-means*$(Q_\ell, num\_replicas)$
8: $score_\ell \leftarrow \mathsf{agg\_score}^{(i)}(\ell, G_\ell, q_\ell^{(i)})$ (Equation 2)
9: $\lambda = \arg\min_{\ell \in \mathcal{S}}\{score_\ell\}$
10: $O \leftarrow$ any *minority* locations from $G_\lambda \setminus Q_\lambda$
11: // Return the leader, set of voters and set of replicas
12: **return** $(\lambda, Q_\lambda \cup O, G_\lambda)$

Note that in Algorithm 5, unlike in Algorithm 4, we know both the leader and the quorum latency when invoking Algorithm 3 and therefore in line 18 of Algorithm 3 actually use the cost function given in Equation 1 (with the change that the summation is done only over clients in $\mathcal{C}_g$) instead of the simplified cost model given in Equation 3. For simplicity, this is omitted from the pseudo-code.

6. EVALUATION

In this section we describe the evaluation of our optimization framework with one of Google's large-scale distributed storage systems. This particular system supports the five representative operation types described in Section 2, which made it the perfect candidate for optimization.

We implemented a system consisting of three tools: a data collection pipeline, an optimizer, and a simulator. The data collection pipeline fetches relevant inputs on the number and latencies of relevant operations from Google's monitoring tools, as well as relevant data from the database schemas such as the network QoS class used by each database, and then prepares it for consumption by the optimizer. The data is broken down into several nonoverlapping time intervals, within each interval — by database, and within each database — by client cluster and by operation type. The optimizer generates scores for each one of the requested optimization tiers on each one of the time intervals $1..i$ reported by the collection pipeline, using an exponential moving average as demonstrated in Section 3 (Equation 2) with $\tau = 2$ as the decay parameter. It then gives a placement recommendation for each interval based on the previous ones. Finally, the simulator compares the optimizer's recommended placement strategy for each interval with other reasonable placement heuristics as well as with the optimal placement for the time interval.

Our experiments were carried out on machines with 12-core 3.50GHz Xeon(R) CPU and 32 GB RAM. The running times of our tools for tiers 1 and 2 for 48 time intervals on all production databases combined were under 1.5 minutes. In what follows we present experiments dedicated to each of the optimization tiers.

6.1 Leader Placement

In this section we show experimental results demonstrating the benefit of optimizing leader placement for the vast majority of databases in our storage system.

Speedup potential. In the following experiment, we scored the current configuration of each database and compared it with the configuration proposed by our optimizer. We analyzed the average operation latency of databases during one typical workday partitioned into 48 nonoverlapping intervals of 30 minutes each.

In our storage system, a database administrator can specify an optional "preferred leader" location, and the storage system picks a location close to it (it may not always be possible to use the preferred location due to lack of available resources or ongoing maintenance). When assigning a score to the current database configuration, we need to distinguish databases with and without the "preferred leader" setting. To each database *db* with preferred leader set to location $\ell^\star \in \mathcal{V}(db)$ we assign the score $\mathsf{score}_{db}^{(i)} \triangleq \mathsf{score}^{(i)}(\ell^\star, \mathcal{R}, q_{\ell^\star}^{(i)})$ at each interval $i = 1, 2, \ldots, 48$. For databases *db* without a specified preferred leader, the group leaders are assumed to be spread uniformly across $\mathcal{V}(db)$ (according to our observations, this assumption closely models real deployments in our system). Accordingly, the score of such database configuration is the average of scores $\mathsf{score}^{(i)}(\ell, \mathcal{R}, q_\ell^{(i)})$ across $\mathcal{V}(db)$:

$$\mathsf{score}_{db}^{(i)} \triangleq \frac{1}{|\mathcal{V}(db)|} \sum_{\ell \in \mathcal{V}(db)} \mathsf{score}^{(i)}(\ell, \mathcal{R}, q_\ell^{(i)}) \ . \qquad (4)$$

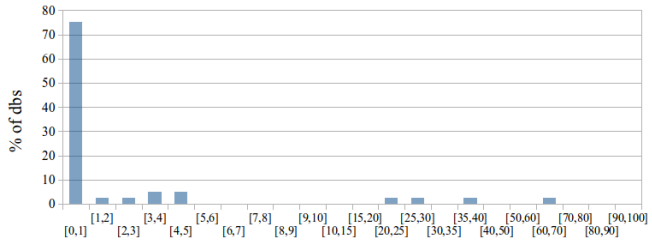
For each interval $i = 1, 2, \ldots, 48$, we calculate

$$1 - \mathsf{score}^{(i)}(\lambda_{db}^{(i-1)}, \mathcal{R}, q_{\lambda_{db}^{(i-1)}}^{(i-1)}) / \mathsf{score}_{db}^{(i)},$$

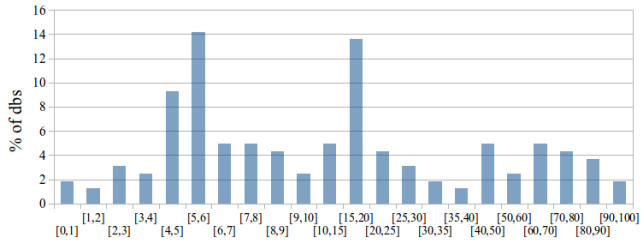
the potential reduction in latency when following the placement recommendation of our optimizer, which places the leader in interval i based on the preceding intervals $1 \ldots i-1$. For each database *db*, we calculate the *average latency reduction* over all the values of i. Figures 1(a) and 1(b) demonstrate the effectiveness of optimizing leader placement for databases with and without an existing preferred leader setting, respectively.

Observe that there is a significant divide between databases that manually set the preferred leader and those that do not in terms of latency reduction when following our recommended leader placement. We can see that typically, administrators that choose to set the preferred leader, set it in a way matching the recommendation of our optimizer; this can be seen in Figure 1(a) which shows that over 75% of the databases of this kind are found in the first bucket $[0, 1\%]$, i.e., they cannot further benefit from our recommendation. This serves as a validation that our model matches the intention of database administrators in all these cases. We see, however, that for some databases the manual setting is sub-optimal, as evidenced by the existence of ∼10% outliers, the score of which is off by at least 10% from the optimum. Our recommendations can help speed-up such outliers.

We found, however, that only 25% of all databases specify a preferred leader and with more new databases created, this



(a) Latency reduction (in % to current) for databases with preferred leader



(b) Latency reduction (in % to current) for databases without preferred leader

Figure 1: Histogram of latency reduction for databases with and without a preferred leader setting. Bins on the x-axis denote % latency reduction compared to current placement. The height of each bin (y-axis) is the percent of databases (with or without a preferred leader setting) for which the corresponding reduction in latency was measured.

percentage diminishes further. This surprising fact further motivates the need for automatic optimizations. For the remaining 75% of the databases our tool provides significant latency improvements, as shown in Figure 1(b). The average operation latency in many of the cases can be reduced by tens of percent. Over 17% of such databases can halve their average operation latency by following our placement recommendation. For some databases, latency is reduced by more than 90%.

Optimizer output and recommendation oscillation. Figure 2 shows a sample output of our optimizer, which outputs the best leader location every 30 minutes and the latency overhead for alternative locations, compared to the best one (for brevity, we show only 2 additional locations). Notice the oscillation in recommendations between clusters ℓ_1 and ℓ_3 caused both by their similar scores and by workload changes between 00:30–02:00 and 03:00–05:00. Our algorithm mitigates minor workload spikes by using a decay parameter τ which counters the spikes with historic scores. A second level of defence should be deployed which considers the costs and benefits of moving the leader to a different location. For example, moving the leader may not be worth while if the optimizer predicts a 2% latency reduction.

22:30: opt ℓ_1, 2^{nd} best $\ell_2 = 4\%$, 3^{rd} best $\ell_3 = 9.1\%$
23:00: opt ℓ_1, 2^{nd} best $\ell_2 = 5.77\%$, 3^{rd} best $\ell_3 = 9.77\%$
23:30: opt ℓ_1, 2^{nd} best $\ell_3 = 5.2\%$, 3^{rd} best $\ell_3 = 23.53\%$
00:00: opt ℓ_1, 2^{nd} best $\ell_2 = 5.24\%$, 3^{rd} best $\ell_3 = 7.68\%$
00:30: opt ℓ_3, 2^{nd} best $\ell_1 = 5.59\%$, 3^{rd} best $\ell_2 = 13.07\%$
...
02:00: opt ℓ_3, 2^{nd} best $\ell_1 = 14.32\%$, 3^{rd} best $\ell_2 = 23.42\%$
02:30: opt ℓ_1, 2^{nd} best $\ell_3 = 7.38\%$, 3^{rd} best $\ell_2 = 9.16\%$
03:00: opt ℓ_3, 2^{nd} best $\ell_1 = 22.6\%$, 3^{rd} best $\ell_2 = 33.09\%$
...
05:00: opt ℓ_3, 2^{nd} best $\ell_1 = 11.49\%$, 3^{rd} best $\ell_2 = 23.46\%$
05:30: opt ℓ_3, 2^{nd} best $\ell_1 = 3.3\%$, 3^{rd} best $\ell_2 = 15.08\%$
06:00: opt ℓ_1, 2^{nd} best $\ell_3 = 0.92\%$, 3^{rd} best $\ell_2 = 11.73\%$

Figure 2: Sample output of the optimizer.

Comparison with other placement strategies. We use our simulator to compare four placement policies using historical storage activity data from one typical day, discretized into 48 intervals of 30 minutes each. For $i = 2, 3, 4, \ldots, 48$ and each one of the strategies, the simulator sets the leader at time interval i, based on the prediction provided by the placement strategies on interval $i-1$ and assigns score $s^{(i)}$ to that prediction based on the actual workload data for interval i.

We considered four strategies for each database db: (*optimized*) placing the leader at $\lambda_{db}^{(i)}$ (with decay $\tau = 2$), as predicted by the optimizer using data from intervals preceding i, whose score on interval i is $\mathsf{score}^{(i)}(\lambda_{db}^{(i-1)}, \mathcal{R}, q^{(i-1)}_{\lambda_{db}^{(i-1)}})$, (*closest-to-writes*) placing the leader statically in a cluster $\ell^\dagger$, wherefrom most of the transactions in interval $i = 0$ originated, with score $\mathsf{score}^{(i)}(\ell^\dagger, \mathcal{R}, q^{(i)}_{\ell^\dagger})$, (*smallest-quorum*) placing the leader in a cluster $\ell^\diamond = \ell^\diamond(db)$, where the average round-trip-time latency $median_{v \in \mathcal{V}(db)}\{\mathtt{rtt}^{(i)}_{\ell^\diamond, v}\}$ from the leader to the majority of voters is minimal, with score $\mathsf{score}^{(i)}(\ell^\diamond, \mathcal{R}, q^{(i)}_{\ell^\diamond})$, (*average*) random leader location across all the groups of the database db, achieving the average score as in (4). We compare with the closest-to-writes and smallest-quorum strategies, since they are sometimes employed by database administrators when setting the preferred leader, and with the average strategy, since it reflects the performance of databases without the preferred leader setting, as explained in the previous experiment. The closest-to-writes strategy is a common heuristic used also in other systems (see Section 7). Our baseline is the optimal "oracle" strategy which sets the leader for interval i at $\lambda_{db}^{(i)}$ (considering the i-th interval workload when aposteriori determining the best leader location for interval i, using $\tau = \infty$). Latency overhead $s^{(i)}/\mathsf{score}^{(i)}(\lambda_{db}^{(i)}, \mathcal{R}, q^{(i)}_{\lambda_{db}^{(i)}}) - 1$ with respect to the optimum $\mathsf{score}^{(i)}(\lambda_{db}^{(i)}, \mathcal{R}, q^{(i)}_{\lambda_{db}^{(i)}})$ is calculated for each strategy score $s^{(i)}$ on each interval $i \geq 2$.

Figures 3(a) and 3(b) demonstrate latency reductions (in percent) for two databases with no manual preferred leader location. The optimized strategy perfectly predicted the optimum for both databases. In general, for all the databases, predictions were nearly perfect, with small deviations from optimum due to sudden workload spikes. More than 90% of operations belonging to the database in Figure 3(a) were bounded reads, that is why the closest-to-writes and the smallest-quorum policies, both of which disregard the locations of readers, underperform compared to the optimized strategy which considers client locations and all operation types. The smallest-quorum policy is slightly better than closest-to-writes due to its choice of a well-connected replica as leader. In Figure 3(b), 38% of operations are strong reads and 60% of operations are weak reads. Once again the optimized strategy prefectly predicts the optimum strategy and outperforms the average and the smallest-quorum strategies by a large margin (more than 60% on average); the closest-to-writes strategy is not applicable in this case, as there were virtually no transactions in the considered database.

6.2 Evaluation in Production

We are working directly with customers to validate our models in production. We present the results of one such experiment in Figure 4. For simplicity, in this experiment

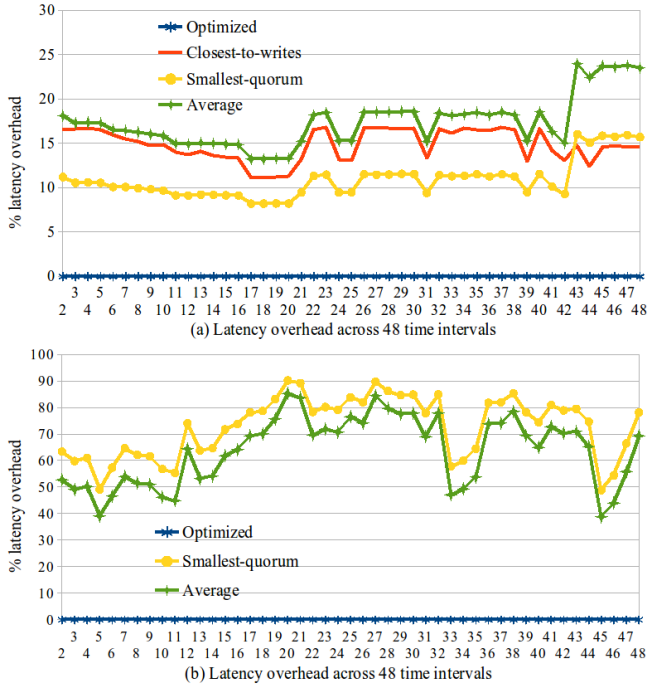


Figure 3: Latency overhead of various placement strategies compared to the "oracle" optimum score $\text{score}^{(i)}(\lambda_{db}^{(i)}, \mathcal{R}, q^{(i)}_{\lambda_{db}^{(i)}})$.

we only reconfigured the chosen database once, even though the optimizer outputs recommendations continuously. We monitored the 50-th percentile latency (solid line) as the database is reconfigured (at 16:00) causing all leaders to migrate to the location λ recommended by our optimizer (dashed curve shows the percent of leaders in location λ). We observe a reduction of ∼70% in latency when the migration completes (around 16:15), after which the latency slightly increased and stabilized at ∼40% of its initial value, exceeding the predicted improvement by a factor of 2. Even though our model currently optimizes mean latency, it is interesting to note that in this experiment we saw a reduction of 30% in 99-th percentile latency (however 90% latency did not improve). In another experiment with a different database, we measured a ∼33%, ∼25% and ∼15% speedup

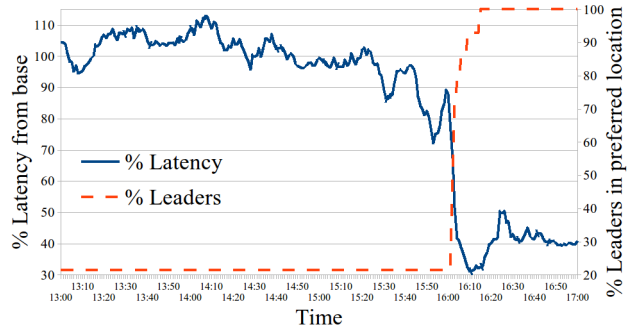


Figure 4: Production experiment with one database. Figure depicts drop in 50-th percentile latency (solid line) along with migration of leaders to recommended location (dashed line). Latency base (100% mark, left y-axes) is chosen as average latency over 3 hours preceding start of experiment (1pm–4pm).

in 50-th, 90-th and 99-th percentile latencies, respectively. Note that our tool predicted a reduction of 39.7% in average latency, which is fairly close to what was observed.

The discrepancy between the predicted and the actual reduction can be ascribed to the fact that at any given point in time the number of leaders at the different locations from $\mathcal{V}(db)$ is not exactly the same (though across a longer period of time on average, it is close to uniform). We found that for the first database mentioned above, one of the locations in $\mathcal{V}(db)$ was taken down for maintenance at the time of the experiment (leaders were evenly spread across the remaining locations). For the second database, the predicted latency reduction was calculated under the assumption that all the leaders have an equal probability of 20% to be in any one of the 5 possible locations, but in reality about one-third of all the leaders were found in the same location. In the future we intend to measure the actual leader distribution across $\mathcal{V}(db)$ dynamically and encorporate it in our model.

6.3 Replica Roles

Next, we evaluate our tier-2 algorithm, that determines the optimal replica types in addition to leader placement. Before conducting our experiments, we intuitively expected to find databases with workflows exhibiting the "follow-the-sun" phenomenon.[2] For example, we expected to see clients in the US and in Europe with intense activity during daytime and reduced activity at night, such that the overall "center" of activity oscillates between US and Europe every ∼12 hours. We found, however, that often the traffic originating from US-based clients is greater than that originating from non-US clients even during night time in the US, therefore the center of activity always remains in the US. This is demonstrated for one database in Figure 5, which shows that European traffic amounts to ∼35% of US traffic during 120 consecutive hours.

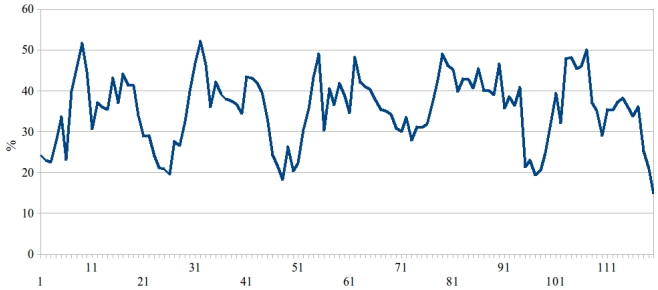


Figure 5: Europe traffic as a percentage of US traffic over 120 hours, for a single database.

Nevertheless, we discovered diurnal patterns between US East Coast and West Coast, as shown in Figure 6, where we plot the ratio between the number of operations originating in the East Coast and the number of operations from clients on the West Coast across 48 consecutive hours with one database, overlaid with the leader locations as suggested by our optimizer. Delineated by vertical lines are points at which our optimizer suggested to switch leader placement from a cluster on one coast to a cluster on the other coast. The reader can readily notice the correlation between ratios larger than 1 and optimizer recommendations for leader placement on the East Coast (as well as between ratios smaller than 1 and recommendations for the West Coast).

[2]Apache ZooKeeper users have a similar intuition [6].

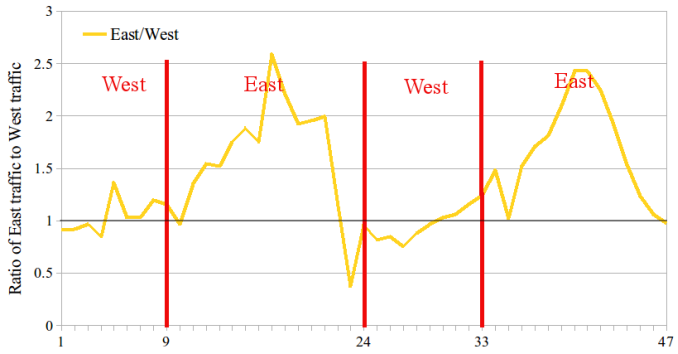


Figure 6: Ratio of East Coast to West Coast traffic, for a single database. Vertical lines denote times at which recommended leader placement changed from East Coast to West Coast or vise versa.

The charts in Figure 7 show the reduction in latency of tier-2 and tier-1 optimizations versus the current score, for two different databases. Figure 7(a) features a database which does not set the preferred leader, ∼98% of operations in which are strong reads, for which the optimization of tier-2 was considerably better than that of tier-1. The leader placement in our tier-1 optimization was chosen in Central US, whereas in tier-2 it migrated to the Pacific Northwest. This reduction in latency looks paradoxical at first, considering the fact that the locations of the voters and quorum are only supposed to affect the latency of transactions, which are virtually nonexistent in this database. This phenomenon is readily explained by the fact that our optimization in tier-2 allows us to consider all the replicas in $\mathcal{R}$ as potential leader candidates, instead of just the pre-determined set of *read-write* replicas considered by our tier-1 optimization. Indeed, the Pacific Northwest replica was initially configured as a *read-only* replica and thus could not function as leader, whereas in tier-2, where we can convert it to a *read-write* replica, it has become a legitimate candidate (and an eventual "winner"), thereby bringing about the surprising reduction in latency.

Figure 7(b) shows a different case, where both tier-1 and tier-2 optimizations suggested the same leader placement, but tier-2 chose a different quorum, due to which the average operation latency was cut by an additional ∼15% compared to tier-1. This database also does not set a preferred leader. About 57% of operations are strong reads and additional ∼42% are multi-group transactions; the latter operations significantly benefited from a new, better connected quorum of replicas. In this case tier-2 approximately doubles the reduction in latency achieved by tier-1.

Note that in all the experiments above we first analyzed the level of failure diversity currently preserved by the database configuration and only suggested alternative configurations maintaining the same diversity level.

6.4 Replica Locations

In tier-3 of optimization, we experimented with the performance of the KQ and QK heuristics (see Section 5).

We start by comparing their performance with that of the exhaustive search preserving the failure diversity constraints of the current database configuration. Figure 8 shows the average ratio between the score given by the optimizer to the exhaustive search and the scores of QK and KQ heuristics, as a function of the total number of replicas, across 12 largest (by the amount of traffic) databases in our system, when the number of voters $|\mathcal{V}|$ was fixed at 3. On the same chart, we also plot the average ratio between the score of the exhaustive search and the best of two heuristics across the same 12 largest databases. For some databases, KQ is better than QK, whereas for others the QK outperforms KQ, resulting in a perhaps surprising fenomena where the average Best(QK,KQ) score is better than both the average KQ and average QK score. For $|\mathcal{R}| \in \{6, 7\}$, KQ was consistently better than QK, that is why Best(QK,KQ) coincides with KQ at that point. The performance of QK on the chart is worse on average that than of KQ because of the fact that most of the considered databases are read-heavy and the relatively small number of replicas considered, of which 2 are "wasted" by QK on the quorum. For such databases it is worthwhile to spread out the replicas to place them as close to most of the clients as possible, which is where KQ excels in comparison with QK.

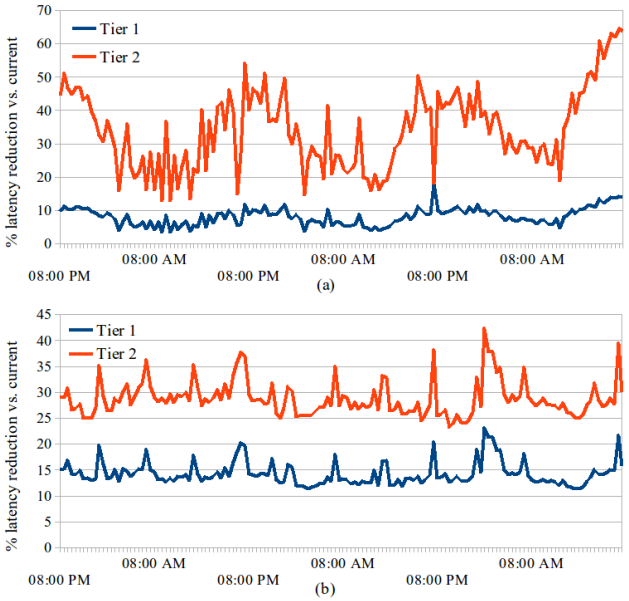


Figure 7: Latency reduction due to tier-1 and 2 optimizations across 3 days of workload data for two selected databases.

We notice that already with $|\mathcal{R}| = 5$ replicas, the best of the two heuristics performs within 5% margin of the optimum produced by exhaustive search, with the added benefit of being substantially faster. For $|\mathcal{R}| = 7$, both QK and KQ, which are polynomial (in $|\mathcal{R}|$), generated results for all 12 databases within seconds, whereas the exponential exhaustive search took several orders of magnitude longer.

Next, we compare KQ and QK, specifically interested in identifying workloads where each of the two algorithms should be preferred over the other. In the following experiment, run with $|\mathcal{V}| = 5$ and $|\mathcal{R}| = 7$ we broke down all databases into buckets by the percentage of transactions among all operations and compared the two algorithms for databases in each bucket. Figure 9 shows a positive correlation between the percentage of transactions and the superiority of QK, which, for databases with more than 60% transactions performs better by more than 80% compared with KQ. A second experiment in which the databases were broken down into buckets by the percentage of weak reads

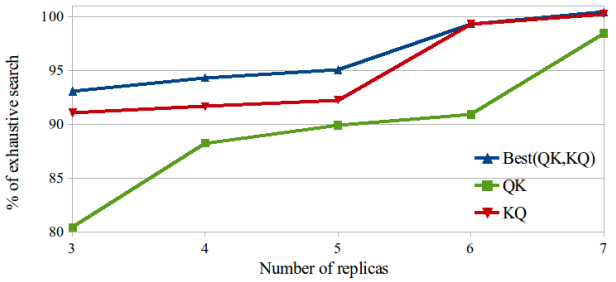


Figure 8: Score of the exhaustive search in percent of the score of KQ, QK and best of two heuristics (with $|\mathcal{V}| = 3$), as a function of $|\mathcal{R}|$.

shows a strong correlation between that percentage and the superiority of KQ; results of this experiment, which appear in Figure 10, demonstrate that for read-heavy databases the speedup of KQ versus QK can be as high as ∼23% on average. Similar experiments with breakdowns of databases by percentages of bounded reads and strong reads did not yield a conclusive outcome.

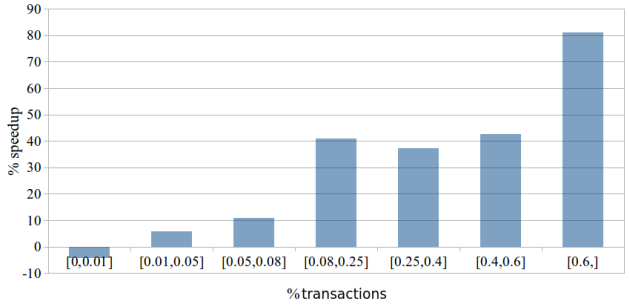


Figure 9: Speedup of QK vs. KQ heuristic as a function of the percentage of transactions in a database.

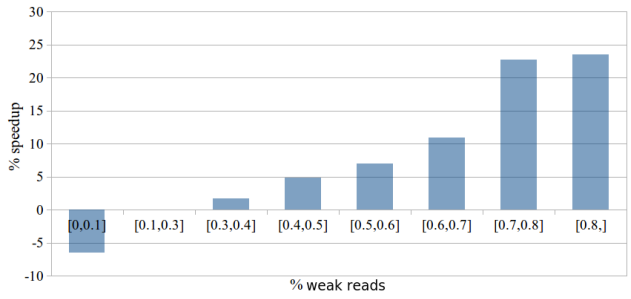


Figure 10: Speedup of KQ vs. QK heuristic as a function of the percentage of weak reads.

How many replicas do you need? Whereas the number of read-write replicas is usually set by an administrator to meet certain fault tolerance goals, the total number of replicas is usually more flexible. The cost of adding / moving / maintaining a replica is often significant as it requires allocating resources, copying data, and potentially deploying other relevant services if collocation dependencies exist. At minimum, the number of replicas should be sufficient to withstand the expected database load. But often, additional replicas are added close to the clients in order to reduce latency. Our framework can help explore the cost/benefit tradeoff of adding such replicas by examining the potential latency gains, and can determine their locations.

In the following experiment, we set $|\mathcal{V}| = 3$ and let $|\mathcal{R}|$ range between 4 and 13. We then measure the scores of both QK and KQ heuristics using workload from one day for one database. Figure 11 demonstrates, for each heuristic and $|\mathcal{R}| \in \{4, 5, 6, \ldots, 13\}$, its average latency slowdown in percent versus the score obtained with 13 replicas, which equals the optimal score for any tier-3 optimization with 3 voters (obtained using an exhaustive search). We can readily see that both heuristics flatten out very soon; specifically, with $|\mathcal{R}| = 11$, both are within ∼13.5% margin from the optimal score. This demonstrates the deminishing returns of adding more replicas – initially each new replica halves the average operation latency, while adding the 12th or 13th replica barely makes any difference.

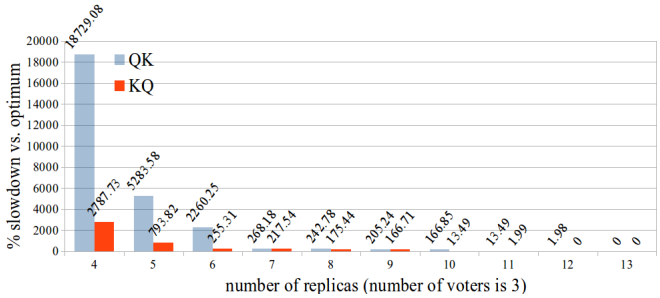


Figure 11: Slowdown of QK and KQ heuristics in percent from the optimum.

7. RELATED WORK

It has long been realized that distributed systems need to be dynamic, i.e., adjust their membership and other configuration parameters over time. Many storage systems [11, 18, 24] use an auxiliary coordination service such as Chubby [13] or ZooKeeper [19] to coordinate reconfiguration while others use the system itself [23, 25]. See [22, 12] for a tutorial on different approaches for reconfiguration of replicated state-machines (i.e., Paxos-like systems) and [8] for a survey on reconfiguring strongly consistent key-value stores. Much fewer works provide insights on how to determine the "best" storage configuration at runtime. Since LAN and WAN environments pose very different challenges, below we focus on storage systems that dynamically reconfigure in WAN.

PNUTS [15] and Megastore [10] place master/leader replicas close to the writers. Earlier works propose other heuristics, e.g., that the current leader should hand off leadership to another replica if that replica forwards more requests to the leader than it receives from elsewhere [28]. These heuristics may work well for some workloads but not for others. For example, in Section 6.1 we show that placing the leader close to the origin of the majority of writes performs poorly on our production workloads, which are mostly read dominant (and yet involve the leader). Furthermore, unlike in [28], we consider network latencies and instead of looking at the aggregate number of requests (or just one request type, such as writes), we consider the detailed flow of each request type and perform an optimization for the entire workload. In this work we formally state optimality criteria and our solution achieves optimal leader placement.

Adaptive replication mechanisms in PNUTS [20] and Nomad [27] dynamically create replicas based on locally observed reads. In Nomad, for example, a replica is created

at a given location when an object is read more than a certain number of times from that location, over a certain period of time, or at a certain rate. Authors of [20] state that they considered more exact methods but decided to use local heuristics since efficiently acquiring, tracking and collecting access statistics from around the world is a complex and expensive process. In this work we leverage Google's monitoring infrastructure to dynamically and accurately track the workload of each database, as well as network latencies. We demonstrate that a solution optimizing the entire workload can be both fast and practical.

Volley [7] proposes a heuristic for placing application data across data centers while minimizing client latency as well as synchronization latency arrising from data inter-dependency. The Volley algorithm does not support data replication and was not evaluated with replicated state. The authors briefly propose to model replicas as distinct data items that may have a certain amount of inter-item communication. Note, however, that with replication each client request is only sent to one of the replicas; unlike Volley, our tier-3 algorithm takes such workload partitioning into account when placing the replicas. Furthermore, unlike Volley, our cost model considers muliple types of client requests with different flows and we compare our replica placement heuristics with the optimum achieved by an exhaustive search using production workloads.

Tuba [9] is an extension of Microsoft Azure Storage that provides geo-replicated key-value store and automatically reconfigures its master and set of replicas based on the workload. Unlike Tuba, our algorithms do not require any changes to the storage system. Tuba uses exhaustive search to enumerate all placement options and choose the best one. It was evaluated with three storage locations using a synthetic workload. We tried exhaustive search, but it was not practical for our "Google scale" storage system. A highly optimized exhaustive search algorithm for replica placement (Section 6.4), akin to the exhaustive search in Tuba, took more than a day to complete and was only slightly better than our heuristic: up to 5% better for 5 replicas per group and less than 1% for larger configurations. In contrast, our optimal algorithms for choosing leader and replica roles (tiers 1 and 2) and heuristic methods for replica placement (tier 3) terminated in less than 2 minutes for all the databases combined.

8. CONCLUSION

Although mechanisms exist for changing the replication policy of distributed storage systems at runtime, system administrators are usually entrusted with determining the "best" configuration manually. We developed a new workload-driven optimization framework that dynamically and automatically determines the optimal configuration for leader and quorum based systems. Our system optimizes three aspects of the configuration: 1) leader location, 2) roles of different servers in the replication protocol, and 3) replica locations. We show that by just applying the first optimization tier to a large-scale distributed storage system used internally in Google, we can reduce the latency of 17% of the databases by more than half, including some databases with a speed-up over 90%. We demonstrate that the second optimization tier further reduces latency by up to 50% in some cases. Finally, we evaluated and compared different strategies for selecting replica locations and showed that they are close to optimal.

9. REFERENCES

[1] Amazon dynamodb. `http://aws.amazon.com/dynamodb/`.

[2] Amazon simpledb. `http://aws.amazon.com/simpledb/`.

[3] Basho riak. `http://basho.com/riak`.

[4] Mongodb. `http://www.mongodb.org/`.

[5] Inclusion exclusion principle. `http://en.wikipedia.org/wiki/Inclusion-exclusion_principle#In_probability`, Retrieved February 10, 2015.

[6] Zookeeper feature request. `https://issues.apache.org/jira/browse/ZOOKEEPER-2027`, Retrieved February 10, 2015.

[7] S. Agarwal et al. Volley: Automated data placement for geo-distributed cloud services. USENIX NSDI, Berkeley, CA, USA, 2010.

[8] M. K. Aguilera, I. Keidar, D. Malkhi, J.-P. Martin, and A. Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bul. of EATCS*, 102, 2010.

[9] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. USENIX OSDI, pages 367–381, Oct. 2014.

[10] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. CIDR, 2011.

[11] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4):10, 2013.

[12] K. Birman, D. Malkhi, and R. van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report 151, MSR, Nov. 2010.

[13] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350, 2006.

[14] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *ACM SOCC*, 2011.

[15] B. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2), Aug. 2008.

[16] J. Corbett et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), Aug. 2013.

[17] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. In *ACM SIGCOMM*, 2012.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, pages 29–43, 2003.

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

[20] S. Kadambi et al. Where in the world is my data? *PVLDB*, 4(11):1040–1050, 2011.

[21] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[22] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, Mar. 2010.

[23] J. Lorch et al. The smart way to migrate replicated stateful services. In *EuroSys*, 2006.

[24] J. MacCormick et al. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.

[25] A. Shraer, B. Reed, D. Malkhi, and F. Junqueira. Dynamic reconfiguration of primary/backup clusters. USENIX ATC, 2012.

[26] M. Stonebraker and A. Weisberg. The voltdb main memory DBMS. *IEEE Data Eng. Bull.*, 36(2), 2013.

[27] N. Tran, M. K. Aguilera, and M. Balakrishnan. Online migration for geo-distributed storage systems. USENIX ATC, pages 15–15, Berkeley, CA, USA, 2011.

[28] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, June 1997.

[29] Z. Yin et al. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2011.