Portkey: Adaptive Key-Value Placement over Dynamic Edge Networks

Joseph Noor
University of California,
Los Angeles
jnoor@cs.ucla.edu

Mani Srivastava
University of California,
Los Angeles
mbs@ucla.edu

Ravi Netravali
Princeton University
rnetravali@cs.princeton.edu

ABSTRACT

Owing to a need for low latency data accesses, emerging IoT and mobile applications commonly require distributed data stores (e.g., key-value or KV stores) to operate entirely at the network's edge. Unfortunately, existing KV stores employ randomized data placement policies (e.g., consistent hashing) that ignore the client mobility and resulting variance in client-server latencies that are inherent to edge applications—the effect is largely suboptimal and inefficient data placement. We present Portkey, a distributed KV store that dynamically adapts data placement according to time-varying client mobility and data access patterns. The key insight with Portkey is to lean into the inherent mobility and prioritize rapid but approximate placement decisions over delayed optimal ones. Doing so enables the efficient tracking of client-server latencies despite edge resource constraints, and the use of greedy placement heuristics that are self-correcting over short timescales. Results with a realistic autonomous vehicle dataset and two small-scale deployments reveal that Portkey reduces average and tail request latency by 21-82% and 26-77% compared to existing placement strategies.

CCS CONCEPTS

• **Information systems → Key-value stores**; **Data federation tools**; *Distributed storage*; *Spatial-temporal systems.*

KEYWORDS

Distributed Data, Edge Computing, Data Locality, Mobility, Data Migration, Replica Assignment

ACM Reference Format:
Joseph Noor, Mani Srivastava, and Ravi Netravali. 2021. Portkey: Adaptive Key-Value Placement over Dynamic Edge Networks. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3472883.3487004

1 INTRODUCTION

The expansion of IoT and mobile systems has resulted in deployments of high-volume data producers and consumers residing at the network edge. Example applications include autonomous vehicular networks [4, 54, 59, 71, 73, 85], federated learning [10, 36, 89], drone-assisted disaster management [19, 24, 25], and AR/VR [12, 15, 23]. Key to these applications are the inherent mobility and resource constraints of clients and (potentially) servers, as well as the requirement of low-latency data accesses [62, 81, 89]. Consequently, these applications typically employ lightweight datastores (e.g., key-value (KV) stores such as Redis [65] and Apache Cassandra [39]) *entirely* at the edge [16, 45, 50, 63], in order to avoid high edge-to-cloud communication latencies [52].

Unfortunately, existing distributed KV stores are ill-suited for edge settings, and instead were designed for datacenter environments with relatively uniform client-server latencies, e.g., when servers reside on the same rack. Accordingly, KV stores typically opt for randomized KV assignment strategies [20] such as consistent hashing and hash slot sharding that prioritize load balancing and fault tolerance, but ignore client mobility and the resulting client-server latencies (§3.1). The result is that retargeting KV stores to the edge computing context can yield largely inefficient data placements. For example, using our dataset for an autonomous vehicular application (§2.2), we find that existing placement strategies yield 1.7-11.9× higher access latencies than an optimal strategy that explicitly incorporates client mobility (§3.2); Figure 1 illustrates the intuition behind this suboptimality.

To fill this void, we present **Portkey**, a new distributed KV store that explicitly incorporates the time-varying mobility and latency patterns experienced by edge applications. Portkey formulates data placement as an online optimization problem, whereby data access patterns and client locations/latencies are continuously tracked and used to tune KV placements in a manner that globally minimizes access latencies. To realize this, Portkey must overcome two challenges with regards to efficient data collection and fast placement decisions. The underlying insight to both of our solutions (described below) is that, due to the very nature of dynamic systems, an optimal placement now is likely to become stale in the near-future. Thus, Portkey prioritizes rapid but potentially suboptimal decisions over delayed optimal ones.

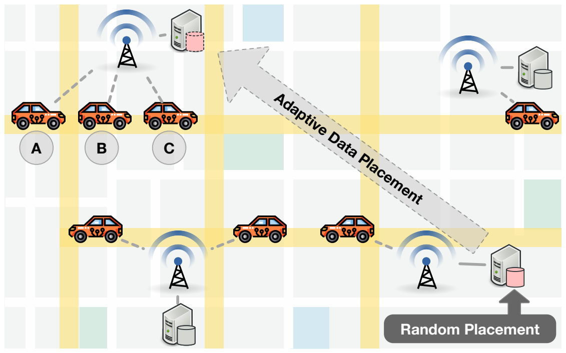

Figure 1: Example smart city that coordinates autonomous vehicles using datastore servers attached to distributed access points. The primary replica for a KV pair shared by clients *A*, *B*, and *C* should intuitively be placed at their nearest host, which may vary over time as the clients move. The randomized placement used in existing systems ignores this locality, resulting in potentially high access latencies.

Challenge 1: efficient data collection. At its core, the ideal placement for a given KV pair (or simply KV) at any time is impacted by two factors: which clients access that KV, and what is their latency to each available datastore server. The former can be logged by transparent request proxying, while the latter involves generating a logical network distance matrix across distributed clients. Though conceptually straightforward, collecting latency information is difficult under the tight resource constraints imposed by edge networks (i.e., bandwidth [86]), as well as edge devices such as IoT gateways and sensor nodes (i.e., energy and memory [74]).

To solve this, Portkey generates succinct latency sketches [28, 44] using a series of lightweight techniques inspired by Network Tomography [84]. First, to generate a holistic view, Portkey profiles the end-to-end latency of the datastore from the perspective of each client by (1) passively profiling application-generated requests, and (2) judiciously inserting active probes to servers that are not accessed by client workloads. Then, in subsequent time windows, Portkey employs *locality-aware reprofiling* such that latency information is recollected only if there exists sufficient evidence that a client's motion *might* affect KV placements. Importantly, due to the short time windows that Portkey operates over, reprofiling decisions consider only proximal servers whose client-server latencies would be most affected by short-term mobility. The same servers should already house the KVs that a client accesses, resulting in low reprofiling overheads (i.e., few active probes).

Challenge 2: fast placement decisions. Even with the necessary information, making placement decisions involves solving a computationally hard optimization problem (reducing to the NP-hard Partition problem [8, 35]) that incorporates all of the influencing factors including client location, server capacity, network state, and workload patterns (§4.2). Worse, the ideal placements can change as any of the aforementioned properties change, which can occur at very short time-scales in settings with high client mobility. For example, in our mobility trace of taxis moving through Rome [11], there is a 72% probability that at least one client will switch access points every 10 seconds.

To generate real-time placement decisions in response to changing system dynamics, Portkey's adaptive solver operates on keys independently, and handles host storage constraints by using a greedy assignment that prioritizes KVs with the largest marginal impact on overall datastore performance, i.e., balancing storage requirements with access frequency. Overheads are further reduced by having non-contentious keys, e.g., those that are accessed frequently but only by a single client, skip the formal solver. This greedy heuristic foregoes optimal placement in exchange for rapid placement of the most important KVs at any time. However, suboptimalities (e.g., from ignoring the impact of colocating KVs) only persist for short time scales, and subsequent profiling and solver iterations will reveal the missed latent effects, allowing for timely readjustment.

We implement Portkey as an immediately deployable modular extension to the Redis KV store that can *transparently* adapt data placements to arbitrary workloads and network characteristics [53]. Unfortunately, to our knowledge, there does not exist a public dataset for our target edge applications that includes the associated client mobility. Thus, to evaluate Portkey, we first developed representative datasets for an autonomous vehicular application that incorporate *real* taxi mobility traces over public distributed KV benchmark workloads (§2.2); our datasets and testbed cover a wide range of values for parameters that affect datastore performance including data locality, network topology, and client-server latencies. We also deployed Portkey in two (small-scale) smart building and crowd sourcing applications that use Raspberry Pis and live mobile networks. In comparison to the predominant randomized placement policy and a variety of locality-aware heuristic strategies, Portkey reduced average and tail (95th percentile) request latencies by 21-82% and 26-77%, while delivering low network (1-3%) and memory (< 1MB for thousands of servers and KVs) overheads.

2 TARGET APPLICATIONS

In this section, we first describe our target edge applications and their intrinsic properties (§2.1), and then describe the representative workloads used throughout the paper (§2.2).

2.1 Edge Applications and Goals

Autonomous Vehicular Networks. Applications over vehicular networks commonly require rapid access (10s-100s of ms) to a distributed data management platform. Use cases include the creation and dissemination of 3D feature maps, enhanced line-of-sight augmentation, real-time traffic prediction, route planning, customer-vehicle matching (e.g., for ride-hailing services), and model sharing [4, 34, 54, 59, 71, 85, 88]. Data locality and mobility characteristics range from location-specific (e.g., marking hazards) to highly-mobile (e.g., current line-of-sight feature map). Similarly, the sets of accessing clients vary over time depending on current client locations or the set of neighboring vehicles. Regardless, given the inherent need for rapid decision making, reliable low-latency access to shared data is required on the order of tens of milliseconds [9, 33, 41].

Disaster Management. Embedded technologies have dramatically impacted our ability to predict and respond to natural disasters at the edge. These data processing pipelines are fundamentally latency-sensitive; faster detection provides a better response opportunity. For example, early earthquake and wildfire detection sensor systems leverage crowdsourced data across mobile clients (e.g., IMU or inertial measurement unit values, locations) to determine disaster areas (e.g., earthquake hotspots) and offer advanced notification enabling vital preparation [37, 48, 57]. Incorporating UAV and drone deployments further enhances situational awareness and augments disaster response, ranging from firefighting to search & rescue [19, 24, 25]. Available infrastructure supporting distributed data storage in these scenarios varies from purely peer-to-peer coordination over highly mobile servers (e.g., drones acting as both clients and servers) to static edge deployments, e.g., at shared access points.

Federated Learning. Federated learning is a novel approach to distributed machine learning over multiple edge devices that does not need to exchange or centralize local data samples [10]. Typical model sizes range from 10-100 MBs, with latency requirements to parameter servers on the order of ~10ms [38, 40, 81]. As it involves continual data updates between edge devices and shared model parameters, early implementations have been shown to suffer from inefficient network communication [36]. These delays are further exacerbated by the mobile and disparate nature of clients in edge settings. Accordingly, solutions have noted the importance of low-latency datastore accesses for federated learning [89].

Augmented and Virtual Reality. Avoiding "VR sickness" is a difficult challenge in AR/VR applications. Although a number of sources contribute to this effect, reducing round-trip latency is a primary factor impacting application integrity and user discomfort [12, 15]. To this end, researchers have noted the need for a low-latency data management platform for AR/VR applications for maintaining a consistent user experience [14] and synchronizing shared world state [87], with a gold standard of 15–20ms [23].

2.1.1 *Key Workload Properties.*

Although emerging edge applications are diverse in nature, they share key workload characteristics that distinguish their datastore requirements from traditional cloud applications.

(1) **High Latency Sensitivity.** Low-latency data access (i.e., no more than tens of milliseconds) is critical for all of the aforementioned applications. As application data is commonly produced and consumed at the edge, this motivates data storage to also occur entirely at the edge (to avoid costly cloud-edge network latencies [52]). Note that this is true even for data streaming systems that use a data management broker for pub/sub messaging with data persistence (e.g., for fault tolerance and recovery) [64].

(2) **Large Client-Server Latency Discrepancies.** The networks onto which these applications are overlaid do not offer the relatively uniform client-server latencies of a cloud cluster. Instead, geo-distribution results in certain edge datastore servers residing "closer" to a client than others (with respect to latency).

(3) **Device Mobility.** A unique aspect of these applications is the inherent mobility of the client devices. From smartphones to autonomous vehicles, locations change over time. The movements of a given client can also directly affect its client-server latencies. Accounting for and adjusting to this volatility is essential in optimizing for access latency.

2.2 Representative Dataset and Testbed

To the best of our knowledge, there are no openly available datasets representative of the workloads in the aforementioned edge applications. Thus, to provide a baseline for testing and evaluating improvements to data placement policies, we developed an in-house edge-KV dataset. Our dataset is inspired by the autonomous vehicular network application, and incorporates realistic client mobility patterns and distributed data access characteristics. More specifically, our dataset leverages *real* mobility traces of taxis moving through Rome [11], and its data accesses are derived from the YCSB benchmark (augmented with its distributed extensions [17, 56]). Of course, precise data access/locality patterns and network latencies play a large role in edge datastore performance; we next describe how our setup and datasets cover wide ranges of values for these different properties.

Network Setup. Although Portkey's benefits are most pronounced when there exists a large latency discrepancy between datastore nodes (§6.2), it is designed to operate seamlessly across many of the conditions that the aforementioned edge applications could encounter. Thus, we opted for the general-purpose network setup depicted in Figure 2. As

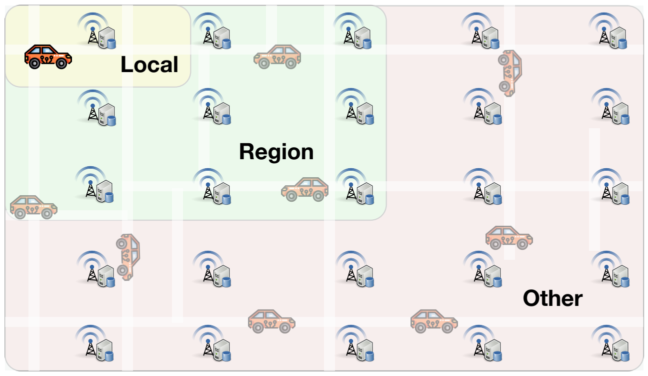

Figure 2: Overview of our experimental testbed.

shown, 25 Wi-Fi access points (APs), each housing a datastore server, are overlaid onto a 5x5 grid with an inter-AP distance of 400m (approximately 2× the outdoor range of 802.11n [2]). Vehicles act as datastore clients as they move throughout the region based on their corresponding taxi mobility trace. For portability, flexibility, and ease of replication, we emulated this network using Mininet [27, 47], with datastore clients and servers running on a shared server. Note that our setup is focused on scenarios where datastore servers are hosted on static edge infrastructure. We discuss how Portkey can be extended to support scenarios with mobile datastore servers (e.g., drones that act as datastores in disaster relief scenarios) in §7.

To cover the wide range of realistic topologies – ranging from zero-infrastructure peer-to-peer routing to wireless access points and/or cellular base stations connected via a wired backhaul [26, 32] – we assign latencies in a tiered manner. In particular, clients access their *local* AP with a random latency ranging from 5-10ms [43, 78], *regional* APs with a random latency between 20-30ms, and all *other* APs with a random latency between 40-50ms. Importantly, this subsumes the two extreme deployments noted above, as well as those in between. For example, in peer-to-peer deployments, latencies scale with the number of hops, and routinely grow to values on the order of ~100ms [18, 51]. In contrast, wireless-to-wired-backhaul settings have reported latencies of 5-10ms to the local AP [43, 78], and upwards of ~50ms contacting neighboring endpoints over the wired network [58, 75]. In §6.2, we breakdown Portkey's performance in specific deployment scenarios, and also consider different latency values for each tier, including those that minimize the benefits that Portkey delivers. Additionally, §6.4 presents results from two small-scale applications running over real (not emulated) networks and edge devices.

Application Workloads. Edge applications vary in terms of the relationship between data locality and data access patterns. To incorporate these properties, we decomposed the YCSB benchmark suite into 6 workload traces that holistically cover both regional locality and the local vs. global nature of KV ownership:

- **per-client**: comprises KVs that are owned and modified by singular clients.
- **regional**: KVs are assigned to individual regions, and clients only access (read or write) KVs for the region they are currently located in.
- **group**: splits KVs and clients into random groups (irrespective of region). Clients only access the KVs within their group.
- **global**: contains global KVs accessed and updated by all clients, e.g., full dataset scans.
- **all-RW**: combines all previous workloads, with request type evenly split across KV reads and writes.
- **all-R**: contains the same KV access patterns as "all-RW," but all accesses are reads.

Each workload has a data access trace per client, and each trace consists of 20,000 read or write (i.e., get or set) requests to a subset of 1,000 keys. All KVs are sized at 1KB. We note that these workloads are intended to cover a broad range of application data access patterns; indeed, different manifestations of each application type in §2.2 can exhibit varying workload characteristics and deployment scenarios. Consequently, there does not necessarily exist a one-to-one mapping between the applications in §2.2 and the benchmark variants described above.

3 BACKGROUND AND MOTIVATION

We provide a brief overview of existing KV datastore placement strategies (§3.1), and present measurements that show why these placement strategies are suboptimal for deployments spanning edge networks (§3.2). Note that KV stores such as Redis can be used as front-end caches for other backing datastores. However, the focus of this paper is on distributed implementations of KV stores (e.g., Redis Cluster) that serve as persistent (i.e., durable) stores by using replication and fault tolerance mechanisms to tolerate failures.

3.1 Existing Placement Strategies

Random KV Assignment. The vast majority of distributed KV stores employ hash-based sharding when partitioning keys across a cluster [20]. Figure 3 depicts two popular implementations: consistent hashing (used by Cassandra [39] and memcached [21]) and hash slot sharding (used by Redis Cluster [65] and MongoDB [49]). Both approaches result in a *random* assignment of KVs; the difference is in the granularity of assignment. In consistent hashing, each key is hosted and replicated at its nearest servers in a hash ring. Hash slot sharding groups keys into slots, with each slot having an individual assignment to replica servers. Although such

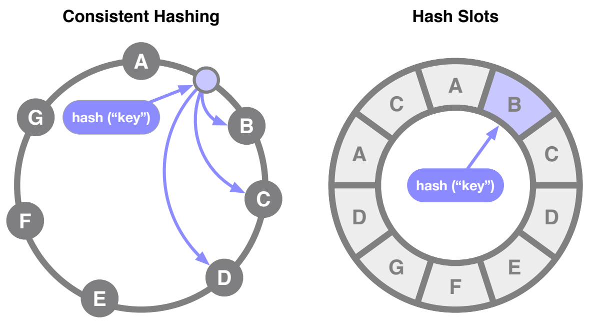

Figure 3: In consistent hashing, servers and keys are hashed onto a ring. Replicas are selected by traversing the ring. In hash slot sharding, the ring space is divided into equal slots, with each slot assigned to datastore servers.

randomized placements ignore the effects of non-uniform client-server latencies (§3.2), they do offer desirable load balancing properties.

Data Replication. Consistency and fault tolerance across a replica set are accomplished through either quorum consensus or primary-secondary replication [20]. Primary-secondary replication allows consistent data accesses to be optimized by only focusing on the primary replica placement. In a quorum, request latency is based on the slowest reply, thus requiring the colocation of a quorum near the accessing clients. In this paper, we target primary-secondary approaches and focus on optimizing placements of a key's primary replica. However, we note that Portkey's placement strategy can be directly applied to secondary or quorum replicas as well. Further, as discussed in §5, Portkey adopts the same fault tolerance and consistency guarantees as the datastore it runs atop.

3.2 The Case for Adaptive KV Placement

To illustrate the performance benefits of adaptive placement, we compared the *Random* placement strategy used by existing KV stores to an *Optimal* placement strategy that leverages future (perfect) knowledge of client locations and data accesses. Using this information, the Optimal strategy predetermines the ideal placement for each KV over a short (near-instantaneous) time window. More specifically, each KV is hosted at the datastore server that minimizes average or tail (95th percentile) request latency across all client accesses in the current window. §4.2 formalizes the optimization problem that underlies the Optimal strategy.

Figure 4 shows the results for both strategies across all of our workloads (§2.2). As shown, the Random strategy results in average request latencies that are 1.7-5.4× worse than the Optimal across the workloads; suboptimalities are 1.4-11.9× for tail latencies. The main issue with Random placement is that it ignores the (time-varying) locality of datastore clients and their KV accesses, and thus potentially places KVs far away from their accessing clients. Of course, the impact of

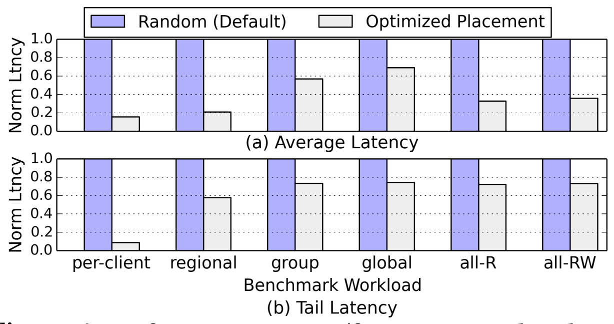

Figure 4: Performance impact (for average and 95th percentile tail latency) of an optimal KV placement policy that explicitly considers client mobility, versus the standard randomized placement policy. Results are normalized to performance with the randomized policy.

this omission is more pronounced in certain workloads than others. For example, when considering average request latency, the inefficiency is most pronounced (4.8-6.4×) for the *per-client* and *regional* workloads where client KV accesses are inherently localized (to the client's location or its encapsulating region). In contrast, the discrepancies between Random and Optimal placements are lower (1.5-1.8×) for the *group* and *global* workloads where KV accesses are not explicitly centered around spatial locality. However, even in these cases, the Optimal strategy outperforms the Random one, primarily by hosting KVs at the servers nearest the majority of (potentially dispersed) accessing clients.

Takeaway. These results suggest that incorporating knowledge of client mobility and data access patterns into KV placement decisions can substantially improve overall datastore performance (i.e., client-perceived request latencies). Of course, the Optimal placement strategy presents a loose upper bound and is unrealistic in practice given its oracle knowledge of future data accesses and client locations. In §4, we describe how Portkey realizes many of these benefits in a practical manner, by using continuous (but efficient) workload and network profiling, and an online migration strategy to achieve real-time, dynamic KV placement.

4 DESIGN

This section details the system design that Portkey uses to practically realize adaptive KV placement with near-optimal performance. Doing so requires addressing two key questions. First, how can the essential information needed to determine an optimal placement (i.e., data accesses, time-varying client-server latencies) be efficiently collected across resource-constrained edge devices and networks? Second, despite the associated computational complexity, how can the collected information be used to make rapid but effective placement decisions that keep pace with time-varying networks and client mobility?

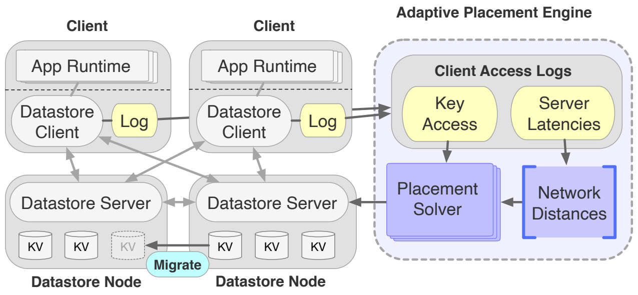

Figure 5: Overview of Portkey. Data accesses and latency information are collected by clients and periodically uploaded to the Adaptive Placement Engine, which determines near-optimal placements and issues migration instructions to datastore servers.

The intuition underlying Portkey's design is to lean into the client mobility and dynamism intrinsic to edge settings. More specifically, without an oracle, Portkey must rely on recent client access patterns and locations/latencies to predict future accesses and latencies. Of course, accurate predictions become more challenging to obtain over long time horizons. Further, in our target settings, recent accesses and current predictions are likely to become outdated quickly as clients move around and client-server latencies adjust accordingly. Thus, Portkey prioritizes fast (and frequent) approximate decisions over slow optimal placements. Accordingly, Portkey opts for an iterative, fast-correcting approach to KV placement for real-time adaptation to edge system dynamics.

Figure 5 illustrates Portkey's workflow. Portkey is incorporated as a modular extension atop existing datastore systems. The client datastore library is augmented to track each KV access and judiciously monitor end-to-end client-server latencies (§4.1); §5 discusses why we opt for client-side profiling. This data is uploaded to the Adaptive Placement Engine at an application-defined *window size*.[1] Upon reception from all clients, the engine first computes a global network distance matrix. Along with the aggregate sets of client-key accesses, the Placement Solver performs fast global approximation of optimal KV placements and issues migration instructions to the appropriate datastore servers (§4.2).

4.1 Efficient Data Collection

Efficient profiling of information that influences optimal KV placements can be broadly decomposed into two categories: minimizing network (and accordingly, device energy) overheads with judicious client-server latency probing (§4.1.1), and succinctly storing latency information and KV access statistics to minimize device memory overheads (§4.1.2).

[1]The frequency of upload affects both the agility with which Portkey adapts to system dynamics, and the network overheads imposed by shipping profiling information. We use a default window of 10 seconds.

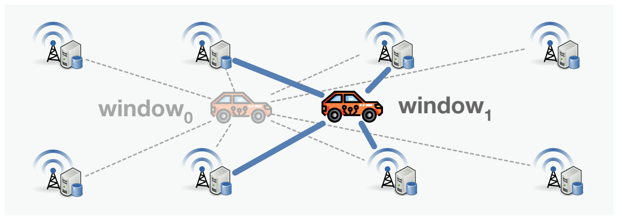

Figure 6: Portkey's locality-aware reprofiling. In subsequent profiling windows, clients only contact their nearest datastore servers, and use the observed latency values to determine if placement-altering motion has occurred; if so, clients then collect latencies to the remaining servers.

4.1.1 Judicious Network Probing. Identifying the optimal KV hosts requires an understanding of the latency delays that each client would incur in contacting each potential datastore server. This latency should encapsulate both the network delays in contacting a server, and the processing delays that the server imposes in serving a requested key. A naive collection strategy would be for each client to simply, in each time window, issue a probe request to each server to log the necessary information. However, this additional (per client-server pair) traffic would add undue stress to servers and edge networks, as well as client devices that must expend energy to support network transfers.

Instead, Portkey employs a lightweight, end-to-end probing technique that is inspired by Network Tomography [83]; importantly, we eschew approaches that rely on support from intermediary network nodes as we target general edge applications with varying administrative policies. At a high level, tomography seeks to infer network internals using only end-to-end measurements that take one of two forms: *passive* tomography leverages data from traffic generated naturally by users, while *active* tomography inserts probes into the network to glean measurements. The goal of Portkey's approach is to leverage its operation over short time-scales (i.e., short windows) to minimize the number of active probes required to obtain accurate and holistic client-server latency information, despite client mobility.

Approach. To develop a comprehensive latency profile to all servers in a given time window, Portkey's client datastore library passively profiles every application-generated request to log the accessed server, as well as the incurred round-trip latency (including server processing delay). Of course, a client's natural data accesses may result in incomplete latency information by failing to contact certain datastore servers; this is especially true with adaptive placement, as distant servers should be minimally contacted. To fill in the missing information, Portkey actively injects requests targeting only the excluded servers, thereby limiting overheads.

Given the short windows over which Portkey makes placement decisions, regenerating an entirely new set of latency values in each window is impractical. This is true even with Portkey's judicious injection of active probes, as clients are unlikely to contact many servers in a short period of time. To handle this, Portkey uses *locality-aware reprofiling*, depicted in Figure 6, to recollect client-server network information only if there is sufficient evidence that latency values have changed enough to *potentially* alter KV decisions. In each time window, a client collects latency information (preferably passively, but if not, actively) only to the k-nearest datastore servers; k is a configurable parameter that is set to 5 by default in our experiments. If the relative ordering of latency values amongst those servers changes, *or* if any latency values change by more than a configurable threshold (20% by default), then a client is deemed to have moved enough to warrant full reprofiling, and active probes are injected to any remaining datastore servers that are not passively contacted.

The guiding intuition is that a client's mobility is inherently localized over short durations, and latencies to the nearest servers are the best indicators of how much motion has occurred. Importantly, with adaptive placement, the nearest servers to a client are the most likely ones to host the KVs that the client accesses. Thus, latency information to the k-nearest datastore servers will often be passively profiled, resulting in low overheads.

4.1.2 Efficient Storage. During normal operation in a window, a client may passively collect multiple latency values per server. Given the potential memory constraints on edge clients, this information must be stored efficiently. To do this, Portkey provides a succinct latency sketching framework that enables applications to specify which part of the latency distribution they would like to consider. When tracking average request latency, each client stores the number of accesses and total aggregate latency for a server, which can be used to derive average request latency while requiring only an 8-byte memory footprint per server. If the application instead wishes to optimize for a metric that requires the full latency distribution (e.g., tail latency), Portkey employs DDSketch [44] to track approximate quantiles with a minimal memory footprint.

Tracking Data Accesses. In addition to latency information, Portkey clients must also track KV accesses. In particular, for each data request, Portkey must record which KV was accessed, as well as the corresponding payload size. Payload sizes must be collected because they dictate which KVs can fit on a given server, and they provide a mechanism with which to compare the relative importance of different key placements; indeed, Portkey's Placement Solver (§4.2.2) relies on payload sizes to scale each KV placement to a marginal per-byte cost benefit.

The process of logging client data accesses is fairly straightforward: Portkey transparently proxies each request/response

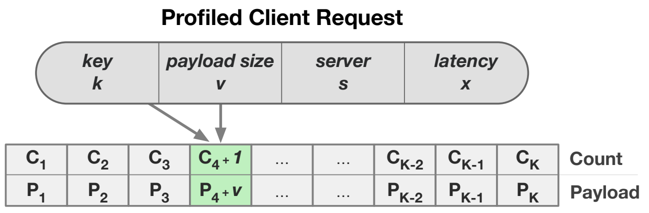

Figure 7: Portkey's sketches for efficiently tracking client data accesses. Access counts and aggregate payload size capture individual client workload patterns.

in the client datastore library. Instead, the primary consideration here is efficient storage, particularly since each client can access a given KV multiple times in a given window. Portkey relies on a key sketch (Figure 7) to bound the memory overhead at each client. Key access counts and aggregate payload size are stored to infer an average payload size for a KV. Given the sparse nature of client-key accesses, Portkey only stores non-zero key counts, consuming 8 bytes each.

4.2 Fast Placement Decisions

Given the information collected in §4.1, the Portkey Placement Solver is responsible for making adaptive placement decisions. Solving for optimal data placement is known to be NP-hard, as it reduces to the Partition problem [8, 35]. Worse, an optimal solution is likely to change with client mobility, which can occur at very short time-scales in edge networks. We begin by formalizing the placement optimization problem (§4.2.1) and then describe Portkey's greedy approximation to enable fast placements that closely resemble optimal decisions (§4.2.2). We center the discussion on optimizing average request latency, and conclude with a description of modifications to support tail latency optimization (§4.2.3).

4.2.1 Problem Formalization. Given a datastore spanning N nodes, the solver's objective is to identify the best host servers for the K keys contained in the system. The best host for a particular key k is chosen by minimizing the overall cost $C(k)$ across all candidates. The specific cost metric depends on the desired performance objective, e.g., minimizing average or tail request latency; we focus on average latency for now. Accordingly, the host is chosen by minimizing

$$C(k) = \min_n \; C_n(k) \quad \forall\, n \in N$$

where $C_n(k)$ is the cost (i.e., latency) of hosting a key k at node n. This latency cost can be computed as an average over every client's distance from the candidate node weighted by the frequency of client access to the given key,

$$C_n(k) = \sum_{i=1}^{N} f_i(k) \cdot d_{in}$$

where d_{in} refers to the distance between nodes i and n (e.g., the average request latency in serving a client i from host n), and $f_i(k)$ represents the relative frequency that client i accesses key k. Client access patterns for each key can be modeled as a frequency access vector

$$\overrightarrow{f(k)} = \begin{bmatrix} f_1(k) & f_2(k) & \dots & f_N(k) \end{bmatrix}$$

which is dictated by each client's access count for the specified key in its data access sketch. Network distances can be represented via the following distance matrix:

$$\mathbf{D} = \begin{bmatrix} d_{11} & d_{12} & d_{13} & \dots & d_{1N} \\ d_{21} & d_{22} & d_{23} & \dots & d_{2N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ d_{N1} & d_{N2} & d_{N3} & \dots & d_{NN} \end{bmatrix}$$

Such a distance matrix can be extracted from the statistics collected across each client's network sketch. Given this formulation, the key's cost vector can be computed as a vector-matrix multiplication:

$$\overrightarrow{C(k)} = \overrightarrow{f(k)} \cdot \mathbf{D}$$

This process can then be independently repeated for each key. If each key's frequency access vector were to be instead represented as a row in a key access matrix, multiplying the key access matrix with the distance matrix derives a cost matrix containing cost vectors across all keys with a single matrix-matrix operation. The computational complexity in computing this cost matrix scales linearly with the number of keys, and in polynomial time with the number of nodes; that is, $O(kn^2)$. The $k \times n$ cost matrix must then be scanned to identify the best candidates for placement. An optimal assignment of KVs over this cost matrix is NP-hard when considering the limited storage capacity at each server [8].

4.2.2 Portkey's Adaptive Placement. Portkey's solver is predicated on the notion that an optimal assignment in any given moment is likely to become stale over time, especially with moving clients. As such, Portkey employs an approximate solver that can quickly and iteratively respond to system dynamics using the latest client-provided information. This is accomplished via a three-fold approach. First, KV costs are treated as independent. While this ignores the impact of colocating KVs onto the same server, subsequent request profiling and solver iterations will account for this effect, readjusting if needed. Second, to address host storage limitations, a greedy assignment prioritizes the KVs with the largest marginal impact on performance. This sacrifices optimal placement in exchange for a rapid adjustment of the most important KVs accessed at any moment. Finally, to ensure efficiency with data-intensive systems, a fallback heuristic is selectively applied to the least contentious KVs, e.g., a KV accessed by a single client can forego the solver and quickly be placed at its nearest host.

While this approach deviates from the optimization formulation in §4.2.1, it allows for fast and self-correcting placement decisions that closely mimic optimal decisions (§6). We next detail the three principles of Portkey's solver in turn.

Independent KV Cost Analysis. An optimal placement assignment should consider the impact of colocating KVs on the same server. This results in an exponential decision space; reassigning a KV affects server load and latency, thereby requiring a cost matrix update for all remaining KVs. The Portkey placement engine opts to treat costs as independent, ignoring the impact of KV assignment. The guiding intuition is two-fold. First, we predict that in many cases the larger contributor to round-trip request latency is network delays as opposed to server processing time. More importantly, given the iterative nature of Portkey's placement decisions, subsequent profiling (which incorporates both network and server delays, as per §4.1) will quickly reveal this latent effect, allowing for prompt self-correction.

Greedy Assignment. To greedily place KVs, Portkey first computes a utility score per KV, indicating the marginal benefit of prioritizing its reassignment. Specifically, the utility score is calculated as the difference in cost (i.e., latency) between the current assignment and the optimal assignment, scaled by the KV payload size. This provides a per-byte marginal cost benefit. These utility scores are then sorted from largest to smallest impact, and KVs are greedily assigned in order of importance. Once a server reaches its storage capacity, keys are assigned to the next best host. When including the cost matrix computation, this assignment process results in overall solver complexity of $O(kn^2 + k\log k)$.

Skipping the Formal Solver. For very large systems (with many KVs) and small window sizes, even a polynomial-time solution may be insufficient. To this end, Portkey selectively employs a locality-aware heuristic that attempts to skip the formal solver. To do so, the placement engine maintains a notion of the max number of keys that can have the full cost matrix computed and analyzed in sufficient time, based on the (1) number of datastore servers, (2) window size, and (3) reusability of computational profiling from prior iterations. If the number of keys accessed in a given window exceeds this max value, a first pass over the key access matrix sorts the keys by the number of accessing clients. Keys accessed by the largest number of clients are solved with cost vector analysis, and the remaining ones use a *dominant-node* heuristic where keys are assigned to the server nearest the most frequently-accessing client. The idea is that the formal solver is most helpful in balancing placement across a large number of (potentially dispersed) accessing clients; keys accessed by a single client are optimally placed nearest that client.

4.2.3 *Optimizing for Tail Latency.*

Portkey's solver is not inherently tied to optimizing for average latency across clients and accesses. Supporting other optimization metrics simply involves altering the generation of the cost matrix used by the adaptive placement solver. For example, to optimize for tail latency, each client uploads its tail latency (instead of average latency) to every datastore server. Then, to compute the cost for hosting a given key on a given candidate server, we generate a distribution of accesses to that key where a client's latency to that server appears a number of times equal to the number of times that client accessed the key. The latency component of the cost is then set to be the tail (e.g., 95th or 99th percentile) of that distribution, rather than a weighted average across all clients' accesses to that key. The remainder of the Portkey solver then operates in the same way as above.

5 IMPLEMENTATION

We implemented Portkey as a modular extension to the Redis Cluster distributed KV store. Aside from being one of the most popular data management systems [76], Redis natively supports deployment over lightweight embedded devices, such as Raspberry Pis and Android smartphones [67], thereby offering compatibility with our target edge applications. As Redis Cluster uses hash slot sharding (§3.1), we implement adaptive placement by dynamically adjusting the slot assignment map and migrating the associated keys. This coarsens the granularity of adaptive placement from individual KVs to hash slots; however, we discuss placements in terms of keys for ease of disposition.

Our implementation of Portkey includes (1) altering the Redis client library to support the collection of datastore usage statistics (we consider the redis-clustr npm package [29]), and (2) developing a standalone program encompassing the placement solver engine. In total, our implementation required ≈1,000 new LOC. A key benefit of client-side modification is that it allows for adaptive placement on unmodified servers, thereby supporting future versions without forking the code base. Further, clients can optionally prioritize which requests are latency-sensitive with selective logging, and in doing so forego modification to the request protocol.

Datastore requirements. We note that Portkey's design is compatible with any datastore that supports the ability to reassign and/or migrate KVs across servers; for instance, our implementation leveraged Redis Cluster's hash-slot migration API. Porting Portkey to another system requires (1) adapting the datastore client library to collect client access statistics, and (2) retargeting the migration mechanism.

Consistency. The consistency of Portkey matches that of Redis Cluster. Redis Cluster cannot guarantee strong consistency, as the primary replica will acknowledge writes before ensuring propagation to a quorum of secondary replicas [65]. As a result, client-perceived write latency is solely governed by primary replica placement; it is this primary replica placement that our current implementation of Portkey aims to optimize. Applications that instead require strong consistency (e.g., via quorum consensus or primary-backup replication) would experience access latencies based on the slowest quorum replica contacted during each KV request. To optimize such scenarios, Portkey's placement strategy must be altered to adapt the placement of multiple replicas per key, i.e., all replicas that are accessed in a blocking manner for the key.

Fault tolerance and recovery. Portkey does not affect the Redis Cluster mechanisms for replication, fault tolerance, or recovery. In the case of a primary node failure, a failover mechanism promotes a secondary replica to replace the primary, adjusting the cluster as needed [65]. This failover is maintained even if a node fails during key migration. While our implementation does not currently incorporate policies for handling correlated and/or clustered failures, Portkey's solver could be easily modified to place KV replicas in different failure zones, e.g., in accordance with with the fault tolerance model of enterprise Redis Cluster versions [68].

Ensuring consistency during migration. Portkey migration follows the recommended Redis Cluster protocol typically used to redistribute keys for nodes entering or exiting the system during deployment [66]. The main benefits of this approach are that (1) no data loss can occur in the case of migration or node failure, and (2) concurrent updates are allowed during migration. This ensures that client workloads remain uninterrupted during migration, enabling transparent adaptive placement with the only noticeable effect being access latency improvement. However, a notable drawback in this reassignment protocol is a restriction that limits bulk migration due to the Redis Cluster epoch mechanism used to propagate assignment map updates. In order to maximize the immediate benefit of adaptive placement, Portkey's Placement Solver sorts key assignments based on marginal cost reduction before issuing migration commands.

6 EVALUATION

To evaluate Portkey, we primarily use the autonomous vehicular workloads and experimental setup described in §2.2. §6.4 additionally describes results from two real, small-scale application deployments. Throughout the evaluation, we consider two versions of Portkey that optimize for either average request latency or 95th percentile tail latency.

6.1 Request Latency Speedups

We compared Portkey with four alternative placement strategies. **Random** refers to the randomized placement strategy used by default Redis and most other existing KV stores

(§3.1). **Accessing Node** is a heuristic that selects the nearest server to a random client that accesses a given KV. **Dominant Node** is an alternative heuristic that places KVs closest to their most frequently accessing client. Both heuristics provide simple yet realistic locality-aware placements based on past accesses. Finally, **Optimal** presents the unachievable lower bound described in §3.2 that uses perfect knowledge of future client-server latencies and data accesses.

As shown in Figure 8, Portkey delivers 21-82% and 16-45% lower average request latencies than the Random and locality-aware heuristics; Portkey's tail latency improvements are 26-77% and 4-75%. Perhaps more importantly, despite lacking oracle-like knowledge of future latencies and KV accesses, the resulting performance with Portkey's placements are always within 15% for average latency, and almost always within 4% for tail latencies. The one exception for the latter is the per-client workload. The reason is that suboptimalities with Portkey stem from delays in learning workload/latency information and shifting KVs to their ideal servers (the optimal strategy knows ideal placements a priori). This is more pronounced in the per-client workload since the latency discrepancy between the ideal (i.e., local) server and all other servers is large, crossing a tier in our testbed. In contrast, for the regional workload, although there is still a single optimal server, the latency discrepancy with several other servers (in the same region) is low.

Of course, performance with each approach varies based on workload characteristics. For example, when considering average request latency, in the *per-client* workload, all locality-aware placements perform substantially (70-81%) better than the random assignment strategy. On the other hand, the performance discrepancies between the locality-aware heuristics and Portkey was most noticeable in the *group* and *global* workloads. In those cases, the Portkey solver was able to more intelligently balance placements across the large number of accessing clients, resulting in a 2.1-2.6x relative performance improvement.

6.2 Varying Edge Settings

In addition to the workload characteristics considered in §6.1, Portkey's performance is affected by datastore server density and client-server network latencies. Here, we present results showing Portkey's performance as these properties vary.

Impact of Datastore Server Density. Figure 9 presents the performance impact of increasing the fraction of edge APs that serve as candidate datastore hosts. For instance, 20% corresponds to 5 of the 25 APs supporting a Redis instance. As shown, Portkey's speedups grow as the density of datastore servers increases. For instance, peak speedups grow from 2.3× to 5.2× when the server density jumped from 40% to 100%. The reason is that a higher server density

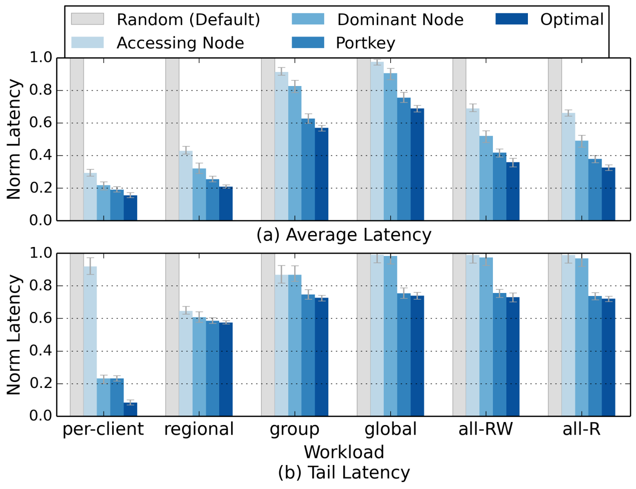

Figure 8: Portkey's average and tail (95th percentile) latency speedups over existing random placement strategies and locality-aware heuristics. Results are normalized to the random approach. The median and entire range for five runs of each workload and placement strategy are plotted.

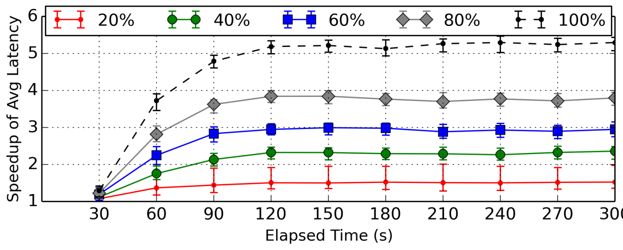

Figure 9: Performance impact of varying the percentage of our testbed's 25 edge APs that can serve as datastore servers. Results use the *per-client* workload, and points represent medians with error bars covering the spread across five runs. APs were randomly selected before each run. Portkey's speedups grow as datastore server density grows.

enables regional locality to be exploited: KVs can more often be placed such that requests are commonly served within the local region. Accordingly, in a deployment where limited resources are available, spreading out the datastore instances will maximize the regional coverage and available locality.

Varying Network Latency. As described in §2.2, our testbed follows a three-tier approach to client-server latencies. More specifically, *local*, *regional*, and *other* servers are randomly assigned between 5-10ms, 20-30ms, and 40-50ms, respectively. To understand how Portkey performs under different network settings, and to analyze Portkey's performance in specific deployment scenarios, we considered four variants for latency assignment: **base** follows the strategy from §2.2, **slow** increases the minimum latency in each tier by 5× while keeping the width the same, **p2p** increases the maximum

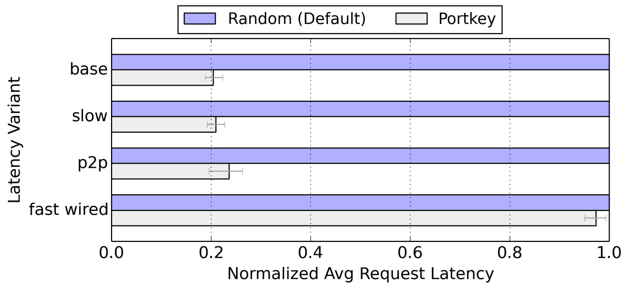

Figure 10: Impact of edge network latencies on Portkey's performance; §6.2 defines the four scenarios. Results use the *per-client* workload and are normalized to random placement. Portkey's performance is largely unaffected by latency values, other than when all client-server latencies are equivalent (eliminating the importance of placements).

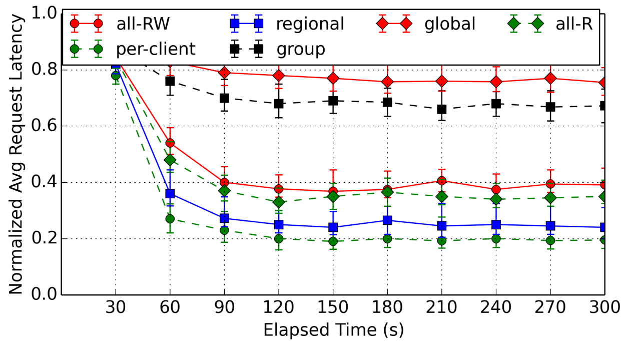

Figure 11: Tail latency improvement over time for Portkey. Results are a snapshot of windowed performance over the first five minutes and are normalized to randomized placement. Placements converge after approximately two minutes, when client workload patterns and network perspective have been sufficiently inferred. Further adjustments are mostly in response to client mobility.

value in each range by 5×, in line with the expected wireless latencies in peer-to-peer routing [1], and **fast wired** reduces all client-server latencies across all tiers to the same value, as might be observed in a well-provisioned cellular network connected to a fast wired backhaul. As shown in Figure 10, altered latency values do not significantly affect the speedups that Portkey delivers. Instead, the key determinant to Portkey's wins is the existence of latency discrepancies between servers, which in turn lead to speedups when KVs migrate close to their accessors. Consequently, Portkey offers little advantage in the *fast wired* scenario; placement becomes unimportant as client-server latencies are uniform.

6.3 Profiling Portkey

Convergence. Figure 11 presents Portkey's performance over time when optimizing for tail latency. Results are windowed and averaged over thirty second intervals. These temporal results indicate the pattern in Portkey's approach to extracting client workload information. As shown, the first iterations of the adaptive placement engine result in the largest number of migrations. Over time, the datastore is able to converge to an asymptotic performance baseline once sufficient data about client workload patterns and network state has been inferred. In other words, after an initial warm up period of approximately 1-2 minutes, performance remained relatively stable and varied primarily in response to continual client mobility.

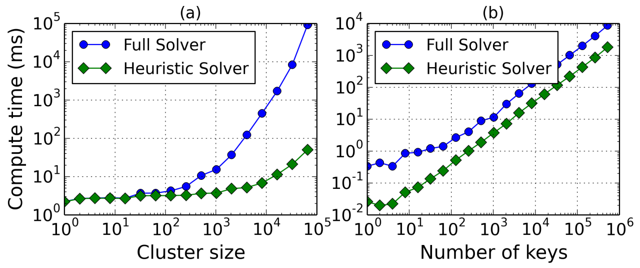

Figure 12: Scalability of Portkey's placement solver when varying (a) cluster size and (b) key set size.

Placement Solver Scalability. An important aspect of the Portkey placement solver is its ability to scale up to large workloads and deliver fast decisions. Figure 12 presents the scalability of a solver instance computing placement of (a) 1024 keys over a varying number of servers, and for (b) a varying number of keys over a 1024-server cluster. Profiling was done on a 2019 Macbook Pro. The full placement solver as described in Section 4.2.2 scales linearly with the number of keys, and quadratic to the number of nodes. The heuristic-based approaches scale linearly across both dimensions. Portkey's adaptive solver leverages this notion to selectively fallback to heuristic placement for large cluster and key sizes. For example, with a window size of 10 secs, a single solver operating on a 1,024 node cluster will perform full cost matrix analysis for up to 65,536 keys (consuming 1 sec) and use heuristic placement for additional keys.

Memory Overhead. The memory overhead for the data access log at each client is determined by the number of KV accesses and servers within the system. The key sketch and network sketch each grow linearly with the cardinality of client-key accesses and servers in the system, respectively. Each key access requires 8 bytes to store the client access count and aggregate payload, while each server consumes 8 bytes to store its access count and aggregate latency. Thus, a datastore spanning thousands of nodes and tens of thousands of keys consume less than 1MB at each client.

Network Overheads. We profiled the aggregate requests and associated payloads issued by the cluster with and without Portkey for our workloads. Bandwidth overheads ranged

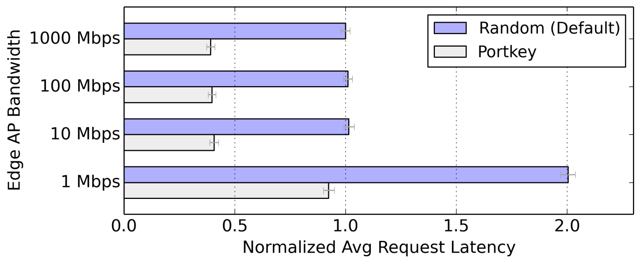

Figure 13: Impact of restricting edge bandwidth for the *all-RW* workload. Portkey's relative advantages persist across the considered bandwidths. Bars list medians (normalized to the 1000 Mbps random placement values) with error bars spanning the range of values across 5 runs.

from 0.75% to 3.11%, depending on the magnitude of KV migration. To ensure that the added bandwidth requirements do not result in a cost of migration that outweighs the benefits, we ran an experiment that increasingly restricted the amount of AP bandwidth. As shown in Figure 13, the most notable impact of restricting bandwidth is the drop from 10 to 1 Mbps, where both Portkey and the default Redis Cluster experience significant slowdown. However, Portkey retains its relative speedup, which consistently falls within 2.1-2.5× across all considered bandwidths.

6.4 Small-Scale Deployments

We deployed two small-scale applications to validate Portkey over real networks and edge devices. The primary objective in these experiments is to highlight how Portkey's benefits in these conditions are similar to those observed in our testbed.

Smart Building Interface. The first application consisted of a Redis Cluster deployment over ten Raspberry Pis (RPi) spread throughout a building and connected over a wireless network, with a single RPi running the Portkey placement engine. Each RPi updates the datastore with (1) frequent updates to the latest reading from attached sensors (e.g., camera image, ambient noise level, wireless network signal strength), and (2) infrequent updates to a key corresponding to device status. An accompanying smartphone application provided user access to view device status and the latest published values. The difference in read-write dominance of each key resulted in varying placements; in particular, the frequently updated sensor value keys migrate to each RPi's local Redis instance, enabling faster system writes and a 5x average latency improvement over randomized placement.

Crowd-Sourced Data Collection. A wide-area Redis Cluster was deployed over nine RPis spread across three campuses, with one RPi designated to host the Portkey engine. User smartphones ran an app that passively updated each user's current GPS coordinates within the datastore to provide crowd-sourced live traffic information. This live activity

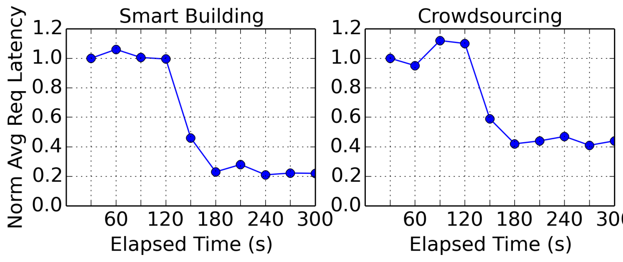

Figure 14: Portkey's speedups in two real deployments. Portkey was enabled after 120 secs of random placements.

map could be optionally accessed and viewed on each smartphone. User mobility, including changing physical location or network connectivity (Wi-Fi vs cellular), resulted in the continual migration of user data to their nearest host. With Portkey, a user's location data migrated to their nearest RPi within their currently occupied campus, yielding a 2.5x improvement in request latency over randomized placement.

Takeaway. Portkey's request latency speedups for both applications are summarized in Figure 14, and closely match those from our emulated testbed. Importantly, without any developer-specific input providing insight into the particular network deployment or workload, Portkey was able to optimize for the most frequent datastore accesses. The designated RPis executing these extensions consumed additional memory footprints of less than 20MB, with less than 2% increase in overall CPU utilization.

7 DISCUSSION

Caching and secondary replicas for eventually consistent reads. In support of reads that do not require strong consistency, secondary replicas or content caches can improve access latency and/or reduce the load on the primary replica. Such approaches are orthogonal to this work, which focuses on primary replica placement to optimize consistent data accesses to the underlying datastore.

Security risks with client-side logging. Trusting statistics reported by client libraries inherently poses security risks. A compromised client may negatively influence placement by misreporting or falsely injecting unnecessary requests. To this end, Portkey supports an optional configuration that limits the overall impact of a compromised client: key access vectors (§4.1.2) can be scaled to a unit vector, resulting in each client equally contributing to placement decisions (§4.2.1). This mitigates the effect of nefarious clients at the expense of a potentially superior placement decision if clients honestly report genuine data accesses.

Scalability limitations of bulk migration. Data migration costs depend on a variety of factors, including the volume of KVs, payload sizes, network latencies, and network bandwidth constraints. In certain scenarios, particularly with large datasets that are configured with short window sizes, Portkey's KV migrations could trail client mobility, thereby forgoing Portkey's benefits, i.e., migrations may be made after a client has moved enough to warrant a superior (different) placement. To address such scalability limitations, we note that migration within a window need not be viewed (and carried out) as all-or-nothing. Instead, Portkey can provide benefits by carrying out only a subset of the KV migrations suggested by its solver that can be completed within the current window, i.e., before new placement decisions are made. To do this, Portkey can estimate the cost of migration of the solver's output based on the required server-to-server coordination latency and data transmission costs, and use that information to enforce a cutoff on the sorted list of KV migrations output by the solver (§4.2.2). Further, while our implementation opted for a design that does not modify the client API, Portkey could expose a client-facing priority mechanism through which developers can specify the latency sensitivity of different keys; this information could then be used to assign utility values to each (potential) KV migration and determine the aforementioned cutoff.

Spanning geographic regions. Deployments spanning geographic regions pose unique constraints when considering an idealized datastore architecture. Quorum architectures enable load balancing across an individual key at the expense of an increased number of client-side system requests. Alternatively, a primary-secondary architecture reduces the required number of requests for consistent access at the risk of overburdening the primary. This work aims to mitigate the downsides of a primary-secondary architecture by migrating the primary replica such that consistent requests can be made fast. Meanwhile, load balancing is achieved by spreading key placements across alternative datastore hosts.

Supporting mobile servers. Recall that Portkey's design and evaluation testbed target scenarios with static datastore servers. However, as discussed in §2.1, certain application deployments may be faced with both mobile clients and datastore servers. The primary component of Portkey that must be altered to account for such scenarios is its locality-aware reprofiling mechanism (§4.1.1). More specifically, in its current form, there is a potential for missed reprofiling triggers if clients and nearby servers all move in tandem (i.e., a mobile "neighborhood" of nodes). Potential remedies include (1) reducing false negatives by including additional random server probes during locality-aware reprofiling, or (2) forcefully triggering full reprofiling at certain intervals.

Tuning the window size. The window size provides a mechanism enabling the datastore administrator or application developer to tune the tradeoff between latency speedups and network overheads based on preferences and perceived notions of dynamism. In particular, larger window sizes imply less responsiveness to changing dynamics, but also lower network overheads from shipping client profiling information. Thus, relatively static deployments require less agility, thereby maintaining high performance with minimal overhead by using large windows. In contrast, rapidly changing networks should configure Portkey for increased agility (i.e., small windows) to more quickly respond to changes.

8 RELATED WORK

Improving Data Locality. Historical work in peer-to-peer distributed hash tables not only laid the foundation for modern distributed datastore design [69, 77], but introduced principal mechanisms for data locality optimization, most notably geographic hash tables that explicitly incorporate location into data placement [60, 61]. In the context of cellular networks, call handoff techniques take an analogous approach of maximizing QoS by dynamically adjusting allocated cellular tower bandwidth based on user locality and range [22, 80], while control plane optimization can use a relaxed session consistency to improve cellular access latency [5].

In the cloud computing domain, previous work seeking to improve the locality of clients accessing a distributed datastore can be broadly categorized into approaches that either (1) intelligently select across a set of existing replicas, or (2) explicitly migrate data across datacenters. Determining the best subset of static replicas for a client to issue datastore requests to (e.g., [79]) can be further enhanced by splitting object data to increase the effective spread, and therefore locality, across datacenters [82]. Volley [3] initially proposed an approach of migrating data between datacenters by cross-referencing a reverse-IP look-up of user location; their solver triggers application-specific migration mechanisms at a course granularity across weeks and months. Future approaches built upon this notion by increasing the frequency of reconfiguration to intra-day [7], grouping small KV shards based on developer insight [6], improving cost efficiency with storage tiers [55], and maximizing availability [13]. More generally, custom hashing methods including locality-sensitive hashing [42] and Social Hash [72] leverage application-specific information to collocate jointly accessed data with clients.

In focusing on the cloud computing setting, placements across individual hosts within a cluster have been left relatively ignored. In contrast, while inspired by previous work in locality-aware placement, much of Portkey's design is centered on accurately collecting and storing latency profiling information despite tight edge constraints. Additionally, unlike work that enforce an association of data with location, Portkey infers locality via observed access latency, thereby supporting KVs not inherently tied to a static location.

Edge Datastores. Recent works have explored datastore platforms that are customized to the inherent variability and distribution of edge networks. Systems that enforce an association of location with data and devices provide a means for locality-aware geographic placement strategies [30, 31]. HDFS-like storage systems have been proposed with large immutable files spread across participating storage devices [70]. For applications that can tolerate eventual or session-based consistency, write-back caches and convergent data structures can be used to scale out a KV datastore at the edge for low-latency writes, high-throughput reads, and efficient inter-replica communication [46, 50]. Finally, application-specific KV systems, such as those for computer vision, have also been proposed [63].

Portkey serves to complement this body of work with latency-aware data placement decisions. Without enforcing an association of location with data, or prior knowledge of application or network characteristics, Portkey seeks a generalized solution to improve the locality of clients accessing a datastore. The objective is to customize a given deployment to specific and variable client accesses while offering the equivalent consistency guarantees of the underlying system, all without necessitating developer input. To the best of our knowledge, none of the previously proposed systems are openly available for use; our Redis Cluster extensions serve to provide a tangible implementation.

9 CONCLUSION

This paper presents Portkey, a new distributed KV store that explicitly targets the intrinsic mobility and time-varying client-server latency profiles experienced in edge applications. Unlike prior datastores that opt for randomized data placement policies, Portkey dynamically adapts data placements according to periodically-profiled latencies and data access patterns. Key to Portkey is its treatment of mobility as a first-class primitive, and its prioritization of rapid (but approximate) placement decisions over slow optimal ones. These insights enable efficient profiling strategies that adhere to edge device and network constraints, as well as greedy placement heuristics that are self-correcting over short timescales. Results with an autonomous vehicle dataset, as well as two small-scale application deployments, show that Portkey reduces average and tail request latencies by 21-82% and 26-77% compared to existing placement strategies.

ACKNOWLEDGEMENTS

We thank our shepherd, Ken Birman, and the anonymous SoCC reviewers for their constructive comments. This work was sponsored in part by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF- 17-2-0196, NSF grants CNS-1901510 and CNS-2105773, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, NSF, DARPA, SRC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] Irshad Ahmed Abbasi and Adnan Shahid Khan. 2018. A review of vehicle to vehicle communication protocols for VANETs in the urban environment. *future internet* 10, 2 (2018), 14.

[2] Ramia Babiker Mohammed Abdelrahman, Amin Babiker A Mustafa, and Ashraf A Osman. 2015. A Comparison between IEEE 802.11 a, b, g, n and ac Standards. *IOSR Journal of Computer Engineering (IOSR-JEC)* 17 (2015), 26–29.

[3] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. 2010. Volley: Automated Data Placement for Geo-distributed Cloud Services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=1855711.1855713

[4] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. 2020. CarMap: Fast 3D Feature Map Updates for Automobiles. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 1063–1081.

[5] Mukhtiar Ahmad, Syed Usman Jafri, Azam Ikram, Wasiq Noor Ahmad Qasmi, Muhammad Ali Nawazish, Zartash Afzal Uzmi, and Zafar Ayyub Qazi. 2020. A low latency and consistent cellular control plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication.* 648–661.

[6] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the shards: managing datastore locality at scale with Akkio. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 445–460.

[7] Masoud Saeida Ardekani and Douglas B Terry. 2014. A self-configurable geo-replicated cloud storage system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 367–381.

[8] Ivan Baev, Rajmohan Rajaraman, and Chaitanya Swamy. 2008. Approximation algorithms for data placement problems. *SIAM J. Comput.* 38, 4 (2008), 1411–1429.

[9] Subir Biswas, Raymond Tatchikou, and Francois Dion. 2006. Vehicle-to-vehicle wireless communication protocols for enhancing highway traffic safety. *IEEE communications magazine* 44, 1 (2006), 74–82.

[10] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, H Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046* (2019).

[11] Lorenzo Bracciale, Marco Bonola, Pierpaolo Loreti, Giuseppe Bianchi, Raul Amici, and Antonello Rabuffi. 2014. CRAWDAD dataset roma/taxi (v. 2014-07-17). Downloaded from https://crawdad.org/roma/taxi/20140717. https://doi.org/10.15783/C7QC7M

[12] Tristan Braud, Farshid Hassani Bijarbooneh, Dimitris Chatzopoulos, and Pan Hui. 2017. Future networking challenges: The case of mobile augmented reality. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1796–1807.

[13] Marc Brooker, Tao Chen, and Fan Ping. 2020. Millions of Tiny Databases. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 463–478.

[14] John Carmack. [n.d.]. Latency mitigation strategies. https://danluu.com/latency-mitigation/. *Twenty Milliseconds* ([n. d.]). Accessed: 2021-09-12.

[15] Eunhee Chang, Hyun Taek Kim, and Byounghyun Yoo. 2020. Virtual reality sickness: a review of causes and measurements. *International Journal of Human–Computer Interaction* 36, 17 (2020), 1658–1682.

[16] Aakanksha Chowdhery, Marco Levorato, Igor Burago, and Sabur Baidya. 2018. Urban iot edge analytics. In *Fog computing in the internet of things*. Springer, 101–120.

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[18] Sergio Correia, Azzedine Boukerche, and Rodolfo I Meneguette. 2017. An architecture for hierarchical software-defined vehicular networks. *IEEE Communications Magazine* 55, 7 (2017), 80–86.

[19] Steve Crowe. 2019. How drones & robots helped extinguish Notre Dame fire. https://www.therobotreport.com/how-drones-robots-helped-extinguish-notre-dame-fire/. Accessed: 2021-05-17.

[20] Miguel Diogo, Bruno Cabral, and Jorge Bernardino. 2019. Consistency Models of NoSQL Databases. *Future Internet* 11, 2 (2019), 43.

[21] Dormando. 2020. memcached - a distributed memory object caching system. https://memcached.org/. Accessed: 2021-05-17.

[22] Nasif Ekiz, Tara Salih, Sibel Kucukoner, and Kemal Fidanboylu. 2005. An overview of handoff techniques in cellular networks. *International journal of information technology* 2, 3 (2005), 132–136.

[23] Mohammed S Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. 2018. Toward low-latency and ultra-reliable virtual reality. *IEEE Network* 32, 2 (2018), 78–84.

[24] Milan Erdelj, Michał Król, and Enrico Natalizio. 2017. Wireless sensor networks and multi-UAV systems for natural disaster management. *Computer Networks* 124 (2017), 72–86.

[25] Milan Erdelj and Enrico Natalizio. 2016. UAV-assisted disaster management: Applications and open issues. In *2016 international conference on computing, networking and communications (ICNC)*. IEEE, 1–5.

[26] Marco Fiore and Jérôme Härri. 2008. The networking shape of vehicular mobility. In *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*. 261–272.

[27] Ramon R Fontes, Samira Afzal, Samuel HB Brito, Mateus AS Santos, and Christian Esteve Rothenberg. 2015. Mininet-WiFi: Emulating software-defined wireless networks. In *2015 11th International Conference on Network and Service Management (CNSM)*. IEEE, 384–389.

[28] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-based quantile sketches for efficient high cardinality aggregation queries. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1647–1660.

[29] GoSquared. 2020. gosquared/redis-clustr: Redis Cluster client for Node.js. https://github.com/gosquared/redis-clustr. Accessed: 2021-05-17.

[30] Harshit Gupta and Umakishore Ramachandran. 2018. Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. ACM, 148–159.

[31] Harshit Gupta, Zhuangdi Xu, and Umakishore Ramachandran. 2018. Datafog: Towards a holistic data management platform for the iot age at the network edge. In *{USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18)*.

[32] Xiaolin Jiang, Hossein Shokri-Ghadikolaei, Gabor Fodor, Eytan Modiano, Zhibo Pang, Michele Zorzi, and Carlo Fischione. 2018. Low-latency networking: Where latency lurks and how to tame it. *Proc. IEEE* 107, 2 (2018), 280–306.

[33] Rafał S Jurecki and Tomasz L Stańczyk. 2014. Driver reaction time to lateral entering pedestrian in a simulated crash traffic situation. *Transportation research part F: traffic psychology and behaviour* 27 (2014), 22–36.

[34] Gorkem Kar, Shubham Jain, Marco Gruteser, Fan Bai, and Ramesh Govindan. 2017. Real-time traffic estimation at vehicular edge nodes. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. 1–13.

[35] Srinivas Kashyap and Samir Khuller. 2003. Algorithms for non-uniform size data placement on parallel disks. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 265–276.

[36] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).

[37] Qingkai Kong, Qin Lv, and Richard M Allen. 2019. Earthquake early warning and beyond: Systems challenges in smartphone-based seismic network. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. 57–62.

[38] Dhruv Kumar, Aravind Alagiri Ramkumar, Rohit Sindhu, and Abhishek Chandra. 2019. Decaf: Iterative collaborative processing over the edge. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*.

[39] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[40] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[41] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. 2018. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 751–766.

[42] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 950–961.

[43] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. 2018. Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 286–299.

[44] Charles Masson, Jee E Rim, and Homin K Lee. 2019. DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2195–2205.

[45] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. 2017. Fogstore: Toward a distributed data store for fog computing. In *2017 IEEE Fog World Congress (FWC)*. IEEE, 1–6.

[46] Christopher Meiklejohn, Heather Miller, and Zeeshan Lakhani. 2018. Towards a solution to the red wedding problem. In *{USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18)*.

[47] mininet. 2020. mininet/mininet: Emulator for rapid prototyping of software-defined networks. https://github.com/mininet/mininet. Accessed: 2021-05-17.

[48] Sarah E Minson, Benjamin A Brooks, Craig L Glennie, Jessica R Murray, John O Langbein, Susan E Owen, Thomas H Heaton, Robert A Iannucci, and Darren L Hauser. 2015. Crowdsourced earthquake early warning. *Science advances* 1, 3 (2015), e1500036.

[49] MongoDB. 2020. Sharding – MongoDB Manual. https://docs.mongodb.com/manual/sharding/. Accessed: 2021-05-17.

[50] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. 2018. Toward session consistency for the edge. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*.

[51] Diala Naboulsi and Marco Fiore. 2013. On the instantaneous topology of a large-scale urban vehicular network: the cologne case. In *Proceedings of the fourteenth ACM international symposium on Mobile ad hoc networking and computing*. 167–176.

[52] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*. 1221–1236.

[53] Joseph Noor. 2021. Portkey: Adaptive Key-Value Placement over Dynamic Edge Networks. https://github.com/nesl/Portkey. Accessed: 2021-09-20.

[54] Piotr Nowakowski, Krzysztof Szwarc, and Urszula Boryczka. 2018. Vehicle route planning in e-waste mobile collection on demand supported by artificial intelligence algorithms. *Transportation Research Part D: Transport and Environment* 63 (2018), 1–22.

[55] Kwangsung Oh, Abhishek Chandra, and Jon Weissman. 2017. TripS: Automated multi-tiered data placement in a geo-distributed cloud environment. In *Proceedings of the 10th ACM International Systems and Storage Conference*. ACM, 12.

[56] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. 2011. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 9.

[57] Ben Paynter. 2019. Could a network of sensors give first responders more time to control wildfires? https://www.fastcompany.com/90424260/could-a-network-of-sensors-give-first-responders-more-time-to-control-wildfires. Accessed: 2021-05-18.

[58] Petar Popovski, Jimmy J Nielsen, Cedomir Stefanovic, Elisabeth De Carvalho, Erik Strom, Kasper F Trillingsgaard, Alexandru-Sabin Bana, Dong Min Kim, Radoslaw Kotaba, Jihong Park, et al. 2018. Wireless access for ultra-reliable low-latency communication: Principles and building blocks. *Ieee Network* 32, 2 (2018), 16–23.

[59] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. 2018. Avr: Augmented vehicular reality. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 81–95.

[60] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. 2001. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. 161–172.

[61] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker. 2002. GHT: a geographic hash table for data-centric storage. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. 78–87.

[62] Ashish Rauniyar, Paal Engelstad, Boning Feng, et al. 2016. Crowdsourcing-based disaster management using fog computing in internet of things paradigm. In *2016 IEEE 2nd international conference on collaboration and internet computing (CIC)*. IEEE, 490–494.

[63] Arun Ravindran and Anjus George. 2018. An Edge Datastore Architecture For Latency-Critical Distributed Machine Vision Applications. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*.

[64] RedisLabs. 2020. Pub/Sub - Redis. https://redis.io/topics/pubsub. Accessed: 2020-05-18.

[65] RedisLabs. 2020. Redis Cluster Specification. https://redis.io/topics/cluster-spec. Accessed: 2021-05-17.

[66] RedisLabs. 2020. Redis Commands - CLUSTER SETSLOT. https://redis.io/commands/cluster-setslot. Accessed: 2021-05-17.

[67] RedisLabs. 2020. Redis on ARM. https://redis.io/topics/ARM. Accessed: 2021-05-17.

[68] RedisLabs. 2021. Rack-zone awareness in Redis Enterprise Software. https://docs.redis.com/latest/rs/concepts/high-availability/rack-zone-awareness/. Accessed: 2021-09-11.

[69] Antony Rowstron and Peter Druschel. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 329–350.

[70] Mathew Ryden, Kwangsung Oh, Abhishek Chandra, and Jon Weissman. 2014. Nebula: Distributed edge cloud for data intensive computing. In *2014 IEEE International Conference on Cloud Engineering*. IEEE, 57–66.

[71] Mauro Salazar, Matthew Tsao, Izabel Aguiar, Maximilian Schiffer, and Marco Pavone. 2019. A congestion-aware routing scheme for autonomous mobility-on-demand systems. In *2019 18th European Control Conference (ECC)*. IEEE, 3040–3046.

[72] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. 2016. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *13th* {*USENIX*} *Symposium on Networked Systems Design and Implementation (*{*NSDI*} *16)*. 455–468.

[73] Guni Sharon, Stephen D Boyles, and Peter Stone. 2017. Intersection management protocol for mixed autonomous and human-operated vehicles. *Transportation Research Part C: Emerging Technologies (Under submission TRC-D-17-00857)* (2017).

[74] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[75] Minho Shin, Arunesh Mishra, and William A Arbaugh. 2004. Improving the latency of 802.11 hand-offs using neighbor graphs. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. 70–83.

[76] SolidIT. 2020. DB-Engines Ranking - popularity ranking of database management systems. https://db-engines.com/en/ranking. Accessed: 2021-05-17.

[77] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking* 11, 1 (2003), 17–32.

[78] Kaixin Sui, Mengyu Zhou, Dapeng Liu, Minghua Ma, Dan Pei, Youjian Zhao, Zimu Li, and Thomas Moscibroda. 2016. Characterizing and improving wifi latency in large-scale operational networks. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 347–360.

[79] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th* {*USENIX*} *Symposium on Networked Systems Design and Implementation (*{*NSDI*} *15)*. 513–527.

[80] Nishith D Tripathi, Jeffrey H Reed, and Hugh F VanLandinoham. 1998. Handoff in cellular systems. *IEEE personal communications* 5, 6 (1998), 26–37.

[81] Animesh Trivedi, Lin Wang, Henri Bal, and Alexandru Iosup. 2020. Sharing and Caring of Data at the Edge. In *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*.

[82] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V Madhyastha. 2020. Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 157–180.

[83] Yehuda Vardi. 1996. Network tomography: Estimating source-destination traffic intensities from link data. *Journal of the American statistical association* 91, 433 (1996), 365–377.

[84] Panagiotis Vouzis. 2014. What is Network Tomography? https://netbeez.net/blog/network-tomography/. Accessed: 2021-05-17.

[85] Xiangpeng Wan, Hakim Ghazzai, and Yehia Massoud. 2020. A Generic Data-Driven Recommendation System for Large-Scale Regular and Ride-Hailing Taxi Services. *Electronics* 9, 4 (2020), 648.

[86] Shuo Wang, Xing Zhang, Yan Zhang, Lin Wang, Juwo Yang, and Wenbo Wang. 2017. A survey on mobile edge networks: Convergence of computing, caching and communications. *Ieee Access* 5 (2017), 6757–6779.

[87] Wuyang Zhang, Jiachen Chen, Yanyong Zhang, and Dipankar Raychaudhuri. 2017. Towards efficient edge cloud augmentation for virtual reality mmogs. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. 1–14.

[88] Xingzhou Zhang, Mu Qiao, Liangkai Liu, Yunfei Xu, and Weisong Shi. 2019. Collaborative cloud-edge computation for personalized driving behavior modeling. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 209–221.

[89] Guangxu Zhu, Yong Wang, and Kaibin Huang. 2019. Broadband analog aggregation for low-latency federated edge learning. *IEEE Transactions on Wireless Communications* 19, 1 (2019), 491–506.