Kadabra: Adapting Kademlia for the Decentralized Web

Yunqi Zhang and Shaileshh Bojja Venkatakrishnan

The Ohio State University
{zhang.8678, bojjavenkatakrishnan.2}@osu.edu

Abstract. Blockchains have become the catalyst for a growing movement to create a more decentralized Internet. A fundamental operation of applications in a decentralized Internet is data storage and retrieval. As today's blockchains are limited in their storage functionalities, in recent years a number of peer-to-peer data storage networks have emerged based on the Kademlia distributed hash table protocol. However, existing Kademlia implementations are not efficient enough to support fast data storage and retrieval operations necessary for (decentralized) Web applications. In this paper, we present Kadabra, a decentralized protocol for computing the routing table entries in Kademlia to accelerate lookups. Kadabra is motivated by the multi-armed bandit problem, and can automatically adapt to heterogeneity and dynamism in the network. Experimental results show Kadabra achieving between 15–50% lower lookup latencies compared to state-of-the-art baselines.

Keywords: Multi-armed bandit · Decentralized protocol · Kademlia p2p routing.

1 Introduction

Decentralized peer-to-peer applications (dapps) fueled by successes in blockchain technology are rapidly emerging as secure, transparent and open alternatives to conventional centralized applications. Today dapps have been developed for a wide gamut of application areas spanning payments, decentralized finance, social networking, healthcare, gaming etc., and have millions of users and generate billions on dollars in trade [5]. These developments are part of a growing movement to create a more "decentralized Web", in which no single administrative entity (e.g., a corporation or government) has complete control over important web functionalities (e.g., name resolution, content hosting, etc.) thereby providing greater power to application end users [46, 11].

A fundamental operation in dapps is secure, reliable data storage and retrieval. Over the past two decades, the cloud (e.g., Google, Facebook, Amazon) together with content delivery networks (CDNs; e.g., Akamai, CloudFlare) have been largely responsible for storing and serving data for Internet applications. Infrastructure in the cloud or a CDN is typically owned by a single provider, making these storage methods unsuitable for dapps. Instead dapps—especially

those built over a blockchain (e.g., ERC 721 tokens in Ethereum)—directly resort to using the blockchain for storing application data. However, mainstream blockchains are notorious for their poor scalability which limits the range of applications that can be deployed on them. In particular, realizing a decentralized Web that supports sub-second HTTP lookups at scale is infeasible with today's blockchain technology.

To fill this void, a number of recent efforts have designed decentralized peer-to-peer (p2p) data storage networks—such as IPFS [15], Swarm [45, 47, 7], Hypercore protocol [3], Safe network [6] and Storj [43]—which are seeing rapid mainstream adoption. E.g., the IPFS network has more than 3 million client requests per week with hundreds of thousands of storage nodes worldwide as part of the network [46]. In these networks, each unique piece of data is stored over a vast network of servers (nodes) with each server responsible for storing only a small portion of the overall stored data unlike blockchains. The networks are also characterized by their permissionless and open nature, wherein any individual server may join and participate in the network freely. By providing appropriate monetary incentives (e.g., persistent storage in IPFS can be incentivized using Filecoin [8, 2]) for storing and serving data, the networks encourage new servers to join which in turn increases the net storage capacities of these systems.

A key challenge in the p2p storage networks outlined above is how to efficiently locate where a desired piece of data is stored in the network. Unlike cloud storage, there is no central database that maintains information on the set of files hosted by each server at any moment. Instead, p2p storage networks rely on a distributed hash table (DHT) protocol for storage and retrieval by content addressing data. While tens of DHT constructions have been proposed in the past, in recent years the Kademlia DHT [33] has emerged as the de facto protocol and has been widely adopted by practitioners. For instance, IPFS, Swarm, Hypercore protocol, Safe network and Storj are all based on Kademlia. To push or pull a data block from the network, the hash of the data block (i.e., its content address) is used to either recursively or iteratively route a query through the DHT nodes until a node responsible for storing the data block is found.

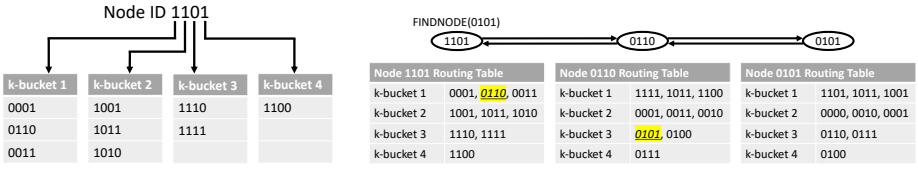
For latency-sensitive content lookup applications, such as the Web where a delay of even a few milliseconds in downloading webpage objects can lead to users abandoning the website [9], it is imperative that the latency of routing a query through Kademlia is as low as possible. Each Kademlia node maintains a routing table, which contains IP address references to other Kademlia nodes in the network. The sequence of nodes queried while performing a lookup is dictated by the choice of routing tables at the nodes. Today's Kademlia implementations choose the routing tables completely agnostic of where the nodes are located in the network. As a result, a query in Kademlia may take a route that criss-crosses continents before arriving at a target node costing significant delay. Moreover, the open and permissionless aspects makes the network inherently heterogeneous: nodes can differ considerably in their compute, memory and network capabilities which creates differences in how fast nodes respond to queries; data blocks published over the network vary in their popularity, with demand

for some data far exceeding others; the network is also highly dynamic due to peer churn and potentially evolving user demand for data (e.g., a news webpage that is popular today may not be popular tomorrow). Designing routing tables in Kademlia that are tuned to the various heterogeneities and dynamism in the network to minimize content lookup delays is therefore a highly nontrivial task.

Prior works have extensively investigated how to design location-aware routing tables in Kademlia. For example, the proximity neighbor selection (PNS) [17] advocates choosing routing table peers that are geographically close to a node (more precisely, peers having a low round-trip-time (RTT) ping delay to the node), and proximity routing (PR) [17] favors relaying a query to a matching peer with the lowest RTT in the routing table. While these location-aware variants have been shown to exhibit latency performance strictly superior to the original Kademlia protocol [33], they are not adaptive to the heterogeneities in the network. PNS is also prone to Sybil attacks which diminishes its practical utility [36]—an adversary controlling a large number of fake Kademlia nodes at a location can cause a nearby node's routing table to be completely filled with adversarial IP addresses. Real world Kademlia implementations in libp2p [4], IPFS and other file sharing networks therefore have resorted to maintaining the peer routing tables largely per the original Kademlia protocol. S/Kademlia [14] is a particularly popular implementation which uses public-key cryptography for authentication and proof-of-work puzzles to avoid Sybil attacks.

We propose Kadabra, a decentralized, adaptive algorithm for selecting routing table entries in Kademlia to minimize object lookup times (to push or get content) while being robust against Sybil attacks. Kadabra is motivated by the (combinatorial) multi-armed bandit (MAB) problem [40, 16], with each Kademlia node acting as an independent MAB player and the node's routing table configurations being the arms of the bandit problem. By balancing exploring new routing table configurations with exploiting known configurations that have resulted in fast lookup speeds in the past, a node is able to adaptively discover an efficient routing table that provides fast lookups. Importantly, the discovered routing table configuration at a node is optimized precisely to the pattern of lookups specific to the node. Our proposed algorithm is fully decentralized, relying only on local timestamp measurements for feedback at each node (time between when a query was sent and its corresponding response received) and does not require any cooperation between nodes. To protect against Sybil attacks, Kadabra relies on a novel exploration strategy that explicitly avoids including nodes that have a low RTT to a node within the node's routing table with the RTT threshold specified as a security parameter. At the same time, Kadabra's exploration strategy also avoids selecting nodes very far from a node. To accelerate discovery of an efficient routing table configuration, Kadabra decomposes the problem into parallel independent MAB instances at each node, with each instance responsible for optimizing peer entries of a single k-bucket. In summary, the contributions of this paper are:

1. We consider the problem of efficient routing table design in Kademlia and formulate it as an instance of the multi-armed bandit problem. Using data-



(a) (b)

Fig. 1: (a) Example of k-buckets at a node in a network with 4-bit node IDs. (b) Example of a recursive routing path taken to lookup key 0101. A yellow-highlighted node ID is a peer to which the query is forwarded for the next hop.

driven techniques for optimizing lookup speeds in structured p2p networks has not been proposed before, to our best knowledge.
2. We propose Kadabra, a fully decentralized and non-cooperative algorithm for learning the routing table entries to accelerate lookups. Kadabra is adaptive to both the traffic demand patterns of the users and the heterogeneities in the network.
3. We validate Kadabra through simulations under various network and traffic settings. In each case, we observe Kadabra to consistently outperform baselines by between 15–50% in latency.

2 Background

2.1 Kademlia

Overview. Kademlia is arguably the most popular protocol for realizing a structured p2p system on the Internet today. In a Kademlia network, each node is assigned a unique binary node ID from a high-dimensional space (e.g., 20 byte node IDs are common). When the network size is large, it is difficult for a node to know the node ID of every single node in the network. A node may have knowledge of node IDs of only a small number (such as logarithmic in network size) of other nodes. The most basic operation supported by Kademlia is *key-based routing* (KBR) wherein given a key from the node ID space as input to a node, the protocol determines a routing path from the node to a target node whose ID is the 'closest' to the input key. Closeness between a key and a node ID in Kademlia is measured by taking the bitwise-XOR between the two binary strings, and converting the resultant string as a base-10 integer. The basic KBR primitive can be used to realize higher-order functions such as a distributed hash table (DHT). In a DHT, a (key, value) store is distributed across nodes in the network. A (key, value) pair is stored at a node whose node ID is the closest to the key according to the XOR distance metric. To protect against node failures, in practice a copy of the (key, value) is also stored at a small number (e.g., 20) of sibling nodes that are nodes whose IDs are closest to the initial storing node. To store a (key, value) in the network a node invokes a STORE(key, value) remote

procedure call (RPC), and to fetch a value corresponding to a key the node calls a FindValue(key) RPC [33, 14]. KBR is implemented as a FindNode(key) RPC, which returns the Kademlia node having the closest ID to key.

Routing. Each Kademlia node maintains a routing table containing node ID, IP address and port information of other peers using which Store, FindValue or FindNode queries are routed to their appropriate target nodes. For node IDs that are n bits long, the routing table at each node comprises of n k-buckets, where each k-bucket contains information about k peers. The IDs of peers in the i-th k-bucket of a node's routing table share the first $i-1$ bits with the node's ID, while differing in the i-th bit (Fig. 1a). For a network with m nodes, it can be shown that on average only the first $\log(m)$ k-buckets can be filled with peer entries while the remaining k-buckets are empty due to lack of peers satisfying the prefix constraints.

Queries in Kademlia are routed either recursively or iteratively across nodes. In a recursive lookup, a query is relayed sequentially from one node to the next until a target node is found. The response from the target node is then relayed back on the reverse path to the query initiator. In an iterative lookup, a query initiating node itself sequentially contacts nodes until a target node is found, and receives a response directly from the target node. We focus primarily on recursive routing in this work (Fig. 1b).

When a query for key x is received at a node v, the node searches for a peer v' within its routing table with an ID that is closest to x. If the distance between the IDs x and v' is less than the distance between x and v, then v forwards the query to node v'. Later when v receives a response to the query from v', v relays the response back to the node from whom it received the query. If the distance between x and v is less than than distance between x and v', the node v issues an appropriate response for the query to the node from whom v received the query. To avoid lookup failures, a query initiator issues its query along α (e.g., $\alpha = 3$) independent paths. This basic lookup process described above is fundamental to implementing the Store, FindValue and FindNode functions. We point the reader to prior papers [33, 14] for more details on the lookup process.

2.2 Lookup Latency and Node Geography

A Kademlia node may include any peer it has knowledge of within its k-buckets, provided the peer satisfies the required ID prefix conditions for the k-bucket. Nodes get to know of new peers over the course of receiving queries and responses from other nodes in the network. As node IDs are assigned to nodes typically in a way that is completely independent of where the nodes are located in the world, in today's Kademlia it is likely that the peers within a k-bucket belong to diverse geographical regions around the world without any useful structure. E.g., a recent study [46] measuring performance on the IPFS network reports a 90-th percentile content storing latency of 112s with 88% of it attributed to DHT routing latency. For retrieving content, the reported 90-th percentile delay is 4.3s which is more than 4× the latency of an equivalent HTTPS lookup. Similar observations have been made on other Kademlia systems in the past as well [20].

There has been an extensive amount of work on reducing lookup latencies in DHTs by taking the physical location of nodes on the underlay [26, 27, 30, 37, 39, 48]. For instance, Kaune et al. [31] propose an algorithm that takes the ISPs of nodes into consideration, and also uses network coordinates for reducing latencies. Jimenez et al. [28] tune the number of parallel lookup queries sent or bucket size to achieve speedup. Chen et al. [19] minimize the mismatch between Kademlia's logical network and the underlying physical topology through a landmark binning algorithm and RTT detection. Gummadi et al. [23] do a systematic comparison of proximity routing and proximity neighbor selection on different DHT protocols. The algorithms proposed in these and other prior works are hand-crafted designs, which are not tuned to the various heterogeneities in the network. Moreover, security in these proposed methods has not been discussed as a first-order concern. Indeed, today's DHT implementations have not adopted these proposals into their systems.

2.3 Security in Kademlia

A Kademlia node is susceptible to various attacks, especially in permissionless settings. We consider the following attacks in this work.

Eclipse and Sybil attacks. In an Eclipse attack, an attacker blocks one or more victim nodes from connecting to other nodes in the network by filling the victim nodes' routing table with malicious nodes. In a Sybil attack, the attacker creates many fake nodes with false identities to spam the network, which may eventually undermine the reputation of the network. Today's Kademlia implementations circumvent these attacks using ideas largely inspired from S/Kademlia [14]. In S/Kademlia, the network uses a supervised signature issued by a trustworthy certificate authority or a proof-of-work puzzle signature to restrict users' ability to freely generate new nodes.

Adversarial routing. In Kademlia, a malicious node within an honest node's routing table may route messages from the honest node to a group of malicious nodes. This attack is called adversarial routing, and it may cause delays and/or make the queries unable to find their target keys. To alleviate adversarial routing, S/Kademlia makes nodes use multiple disjoint paths to lookup contents at a cost of increased network overhead.

Churn attack. Attackers can also enter and exit the network constantly to induce churns to destabilize the network. Kademlia networks handle these kind of attacks by favoring long-lived nodes [33, 32].

3 System Model

We consider a Kademlia network over a set of nodes V with each node $v \in V$ having a unique IP address and a node ID from ID space $\{0,1\}^n$. Each node maintains n k-buckets in its routing table, with each k-bucket containing the IP address and node ID of up to k other peers satisfying the ID prefix condition.

We consider a set S of (key, value) pairs that have been stored in the network; each (key, value) pair $(x, y) \in S$ is stored in k peers whose IDs are closest to x in XOR distance. We let S_x denote the set of keys in S. Time is slotted into rounds, where in each round a randomly chosen node performs a lookup for a certain key. If a node $v \in V$ is chosen during a round, it issues a lookup query for key $x \in S_x$ where x is chosen according to a demand distribution p_v, i.e., $p_v(x)$ is the probability key x is queried. We focus primarily on recursive routing in this paper. When a node v initiates a query for key x, it sends out the query to α closest (to x, in XOR distance) peers in its routing table. For any two nodes u, w, $l(u, w) \geq 0$ is the latency of sending or forwarding a query from u to w. When a node w receives a query for key x and it has stored the value for x, the node returns the value back to the node u from whom it received the query. Otherwise, the query is immediately forwarded to another node that is closest to x in w's routing table. When a node w sends or forwards a value y to a node u, it first takes time $\delta_w \geq 0$ to upload the value over the Internet followed by time $l(w, u)$ for the packets to propagate to u. We do not model the time take to download the value, as download bandwidth is typically higher than upload bandwidth. Thus, for a routing path v, u, w with v being the query initiator and w storing the desired value, the overall time taken for v to receive the value is $l(v, u) + l(u, w) + \delta_w + l(w, u) + \delta_u + l(u, v)$. The above outlines our lookup model for the DHT application. For KBR, we follow the same model except only a single query (i.e., $\alpha = 1$) is sent by the initiating node. We assume each node has an access to the IP addresses and node IDs of a small number of random nodes in the network.

Problem statement. For each of the KBR and DHT applications, our objective is to design a decentralized algorithm for computing each node's routing table such that the average time (averaged over the distribution of queries sent from the node) taken to perform a lookup is minimized at the node. We consider non-cooperative algorithms where a node computes its routing table without relying on help from other nodes.

4 Kadabra

4.1 Overview

Kadabra is a fully decentralized and adaptive algorithm that learns a node's routing table to minimize lookup times, purely based on the node's past interactions with the network. Kadabra is inspired by ideas from non-stationary and streaming multi-armed bandit problems applied to a combinatorial bandit setting [12, 18, 35]. A Kadabra node balances efficient routing table configurations it has seen in the past (exploitation) against new, unseen configurations (exploration) with potentially even better latency efficiency. For each query that is initiated or routed through a *Kadabra* node, the node stores data pertaining to which peer(s) the query is routed to and how long it takes for a response to arrive. This data is used to periodically make a decision on whether to retain peers currently in the routing table, or switch to a potentially better set

of peers. Treating the routing table as the decision variable of a combinatorial MAB problem leads to a large space and consequently inefficient learning. We therefore decompose the problem into n independent subproblems, where the i-th subproblem learns only the entries of the i-th k-bucket. This decomposition is without loss of generality as each query is routed through peers in at most one k-bucket. In the following we therefore explain how a Kadabra node can learn the entries of its i-th k-bucket.

In Kadabra, a decision on a k-bucket (i.e., whether to change one or more entries of the bucket) is made each time after b queries are routed via peers in the bucket (e.g., $b = 100$ in our experiments). We call the time between successive decisions on a k-bucket as an epoch. Before each decision, a performance score is computed for each peer in the bucket based on the data collected over the epoch for the bucket. Intuitively, the performance score for a peer captures how frequently queries are routed through the peer *and* how fast responses are received for those queries. By comparing the performance scores of peers in the bucket during the current epoch against the scores of peers in the previous epoch, Kadabra discovers the more efficient bucket configuration which is then used as the k-bucket for the subsequent epoch.[1] To discover new (unseen) bucket configurations, Kadabra also explores random bucket configurations according to a user-defined schedule. In our implementation, one entry on the k-bucket is chosen randomly every other epoch. The overall template of Kadabra is presented in Algorithm 1.

4.2 Scoring Function

During an epoch with k-bucket Γ_{curr}, let $q_1, q_2, \ldots, q_r$ be the set of queries that have been sent or relayed through one or more peers in the k-bucket. For each query $q_i, 1 \leq i \leq r$, let $d_i(u) \geq 0$ be the time taken to receive a response upon sending or forwarding the query through peer u for $u \in \Gamma_{\text{curr}}$. If q_i is not sent or forwarded through a peer $u \in \Gamma_{\text{curr}}$ we let $d_i(u) = \Delta$ where $\Delta \geq 0$ is a user-defined penalty parameter.[2] A large value for Δ causes Kadabra to favor peers that are frequently used in the k-bucket, while a small value favors peers from which responses are received quickly. In our experiments, we choose Δ to be a value that is slightly larger than the moving average of latencies of lookups going through the bucket. The function $\textsc{ScoringFunction}(u, \mathcal{D})$ to compute the score for a peer u is then defined as $\text{score}(u) = \textsc{ScoringFunction}(u, \mathcal{D}) = -\sum_{i=1}^{r} d_i(u)\ \forall u \in \Gamma_{\text{curr}}$. The overall score for the k-bucket is then given as $\textsc{ScoreBucket}(\Gamma_{\text{curr}}, \mathcal{D}) = -\sum_{u \in \Gamma_{\text{curr}}} \text{score}(u)/|\Gamma_{\text{curr}}|$. For a k-bucket that is empty, we define its score to be $-\Delta$.

[1] To increase stability under churn, we may choose to replace only unresponsive peers—as in the original Kademlia protocol—in each epoch.

[2] Notice that $d_i(u)$ is well-defined for both recursive and iterative routing.

Algorithm 1: Algorithm outline for updating entries of i-th k-bucket of node v in each epoch.

input : data $\mathcal{D}$ on queries sent during current epoch; peers Γ_{curr} and Γ_{prev} in k-bucket of current and previous epochs respectively; total score PrevScoreBucket of previous k-bucket; flag F indicating whether to explore in next epoch; list $\mathcal{L}$ of peers eligible to be included within k-bucket; security parameter ρ;
output: updated set of peers Γ_{next} for next epoch;
`/* Score each peer in` Γ_{curr} `using a scoring algorithm based on measurements collected during epoch */`
score$(u) \leftarrow$ ScorePeer$(u, \mathcal{D})$, for each peer $u \in \Gamma_{\text{curr}}$
if *flag F is true* **then**
  `/* Replace worst peer with a random peer during next epoch */`
  $u^* \leftarrow \text{argmin}_{u \in \Gamma}$ score(u)
  $\Gamma_{\text{next}} \leftarrow \Gamma_{\text{curr}} \backslash \{u^*\} \cup$ SelectRandomPeer$(\mathcal{L}, \rho)$
else
  `/* Choose best peer set between current and previous epoch as decision for next epoch */`
  if ScoreBucket$(\Gamma_{curr}, \mathcal{D}) >$ PrevScoreBucket **then**
    $\Gamma_{\text{next}} \leftarrow \Gamma_{\text{curr}}$
  else
    $\Gamma_{\text{next}} \leftarrow \Gamma_{\text{prev}}$
  end
end

4.3 Random Exploration

To discover new k-bucket configurations with potentially better performance than past configurations, a Kadabra node includes randomly selected peers within its bucket through the SelectRandomPeer() function as outlined in Algorithm 1. The Kadabra node maintains a list $\mathcal{L}$ of peers eligible to be included within its k-bucket, which satisfy the required node ID prefix conditions. In addition to the peer IP addresses, we assume the node also knows the RTT to each peer in the list. For a random exploratory epoch, the node replaces the peer having the worst score from the previous epoch with a randomly selected peer from the list. The number of peers in the bucket that are replaced with a random peers can be configured to be more than one more generally.

A key contribution in Kadabra is how peers are sampled from the list of known peers to be included in the k-bucket. Depending on the number of nodes in the network, and the index of the k-bucket, the number of eligible peers can vary with some peers close to the node while some farther away (in RTT sense). A naïve approach of sampling a node uniformly at random from the list, can eventually lead to a bucket configuration in which all peers are located close to the node. This is due to the algorithm 'discovering' the proximity neighbor selection (PNS) protocol which has been demonstrated to have efficient latency performance compared to other heuristics [23, 14]. However, as with PNS, the

routing table learned with a uniformly random sampling strategy is prone to a Sybil attack as it relatively inexpensive to launch a vast number of Sybil nodes concentrated at a single location close to a victim node(s) [38, 42]. While the PNS peer selection strategy does not have an efficient performance in all scenarios (e.g., if the node upload latencies are large; see §5), in cases where it does, Kadabra would be susceptible to attack. What we desire, therefore, is to learn a routing table configuration in which not all peers are located close to the node. Such a routing table configuration may not be performance efficient (e.g., PNS may have a better latency performance in certain scenarios), but is more secure compared to PNS.

We capture this intuition by introducing a security parameter $\rho \geq 0$, that is user-defined, to restrict the choice of peers that are sampled during exploration. For a chosen ρ value, a Kadabra node computes a subset $\mathcal{L}_{>\rho} \subseteq \mathcal{L}$ of peers to whom the RTT is greater than ρ from the node. The SelectRandomPeer$(\mathcal{L}, \rho)$ then samples a peer uniformly at random from $\mathcal{L}_{>\rho}$. A high value for ρ selects peers that are at a distance from the node, providing security against Sybil attacks at a cost of potentially reduced latency performance (and vice-versa).

5 Evaluation

5.1 Experiment Setup

We evaluate Kadabra using a custom discrete-event simulator built on Python following the model presented in §3.[3]

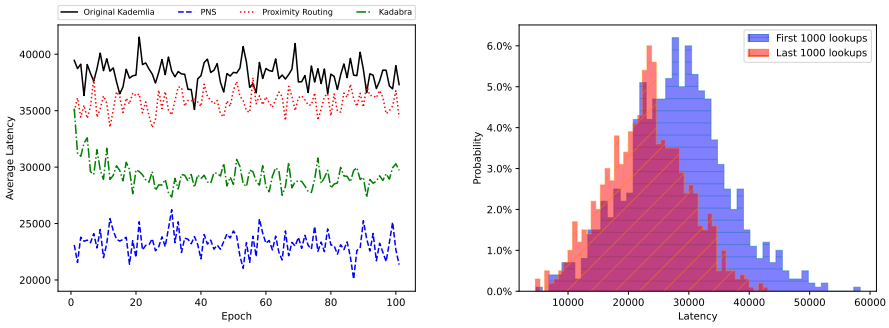
Baselines. Since the main focus of Kadabra is on how to configure the routing table, we compare our algorithm against the following baselines with differing (i) routing table (bucket) population mechanisms, and (ii) peer selection methods during query forwarding:

(1) Vanilla Kademlia [33, 14]. The original Kademlia protocol in which a node populates its buckets by randomly adding peers from node ID ranges corresponding to the buckets. When forwarding a query, the node chooses the peer whose node ID is closest (in XOR distance) to the query's target node ID from the appropriate bucket.

(2) Proximity routing (PR) [23, 14]. In PR buckets are populated exactly as in vanilla Kademlia. However, when routing a query the query is sent to the peer in the appropriate k-bucket that is closest to the node in RTT.

(3) Proximity neighbor selection (PNS) [23, 14]. In PNS the node picks peers which are closest to itself (in terms of RTT) from among eligible peers to populate each k-bucket. When forwarding a query, the node chooses the peer whose node ID is closest to the target node ID from the appropriate bucket.

[3] We do not use the erstwhile popular OverSim [13] and PeerSim [34] simulators, as they are outdated and no longer maintained by their authors. Kadabra simulator is available at https://github.com/yunqizhang99/KadabraSim/.



(a) (b)

Fig. 2: Nodes in a square under uniform demand: (a) Average latency of queries in each epoch for queries routed through the 1st k-bucket of an arbitrary node. (b) Histogram of lookup latencies in Kadabra during the first and last 1000 rounds in a 10 million query run.

Network settings. We consider two network scenarios: nodes distributed over a two-dimensional Euclidean space, and nodes distributed over a real-world geography.

(1) Nodes in a square. In this setting, 2048 nodes are assigned random locations within a 10000 $\times$ 10000 square. The latency $l(u,v)$ between any two nodes u,v is given by $l(u,v) = ||u-v||_2 + w(u,v)$, where $||u-v||_2$ is the Euclidean distance between u and v on the square and $w(u,v)$ is random perturbation from an uniform distribution between 100 and 5000. Each node has a node latency (δ in §3) sampled uniformly between 100 and 2000.

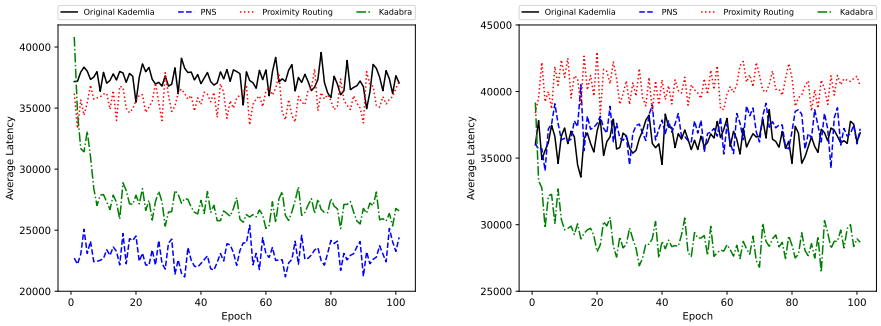
(2) Nodes in the real world. We again consider 2048 nodes located in various cities around the world, as reported by Ethereum node tracker [1]. The latency between nodes in any pair of cities is obtained from a global ping latency measurement dataset [10].[4] Each node has a node latency sampled from an exponential distribution of mean 1000ms.

Application and traffic patterns. We first consider the KBR application under the following three traffic patterns:

(1) KBR under uniform demand. In this setting, a node during a round (see §3) issues a lookup to another node chosen uniformly at random from among the available nodes.

(ii) KBR under demand hotspots. In this setting, there are 20% of keys (nodes) that form the target destination for 80% of lookups. The hotspot nodes are randomly chosen.

[4] For cities not included in the ping dataset, we measure the latency to the geographically closest city available in the dataset.



(a) (b)

Fig. 3: Nodes in a square: (a) Average latency during each epoch for queries routed through the 1st k-bucket of an arbitrary node with demand hotspots. (b) Performance of a node within a region of high node latency nodes.

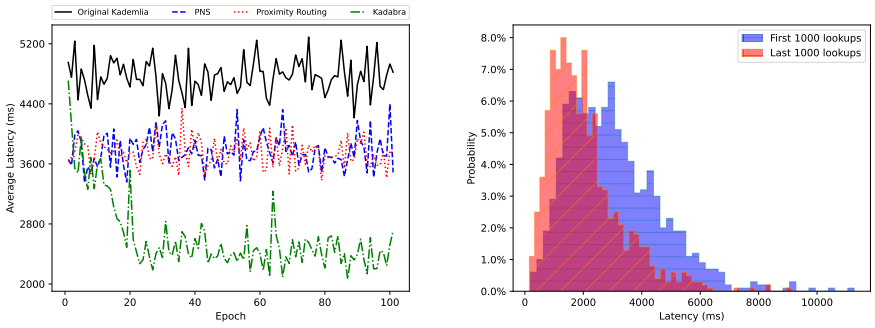
(3) KBR under skewed network bandwidth. To model regions around the world with poor Internet speeds, we consider a subset of geographically close nodes whose node upload latencies (see §3) are twice as large as the average node latency in the network.

In Appendices A–F, we have presented additional results for the above settings, and have also considered the DHT application and iterative routing.

5.2 Results

Nodes in a square. Fig. 2a plots the average latency between forwarding a query through the 1st k-bucket and receiving a response during each epoch for an arbitrarily chosen node within the square. We observe that starting with a randomly chosen routing table configuration (at epoch 0), Kadabra continuously improves its performance eventually achieving latencies that are 15% better. Compared to the latencies in the original Kademlia protocol, Kadabra's latencies are lesser by more than 20%. For this specific network setting, PNS shows the best performance (at the cost of poor security). We have used ρ values of [400, 350, 300, 250, 200, 150, 100, 50, 0] for the different k-buckets (1st to last) in Kadabra, which results in slightly higher latencies compared to PNS.

To show that all nodes in the network benefit from Kadabra, we conduct an experiment lasting for 10 million rounds (1 query per round from a random source to a random destination), with the sequence of first 1000 queries being identical to the sequence of the last 1000 queries. Fig. 2b plots a histogram of the query latencies for the first and last 1000 queries in Kadabra. We observe the 90-th percentile latency of Kadabra during the last 1000 queries is lesser than that in the beginning by more than 24%.



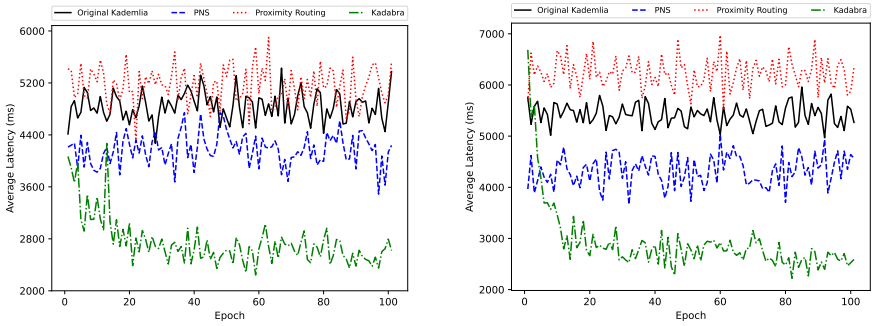
(a) (b)

Fig. 4: Nodes in the real world: (a) Performance of queries routed through the 1st k-bucket of a node in Frankfurt. (b) Histograms of query latencies before and after learning in Kadabra with 10 million lookups.

Fig. 3a shows the average query latency over epochs for queries routed through the 1st k-bucket of an arbitrary node under hotspot demand. With certain keys being more popular than others, Kadabra adapts the node routing tables biasing them for fast lookups of the popular keys—a capability that is distinctly lacking in the baselines. As a result, we observe Kadabra outperforming the original Kademlia and PR by more than 25%.

To show that Kadabra adapts to variations in the Internet capacities of nodes, we consider an experiment where nodes within an area (2000×2000 region in the center of the square) alone have a higher node latency (5000 time units) than the default node latency values. This setting models, for instance, low-income countries with below-average Internet speeds. For a node within the high node latency region, PNS ends up favoring nearby peers also within that region which ultimately severely degrades the overall performance of PNS (Fig. 3b). Kadabra, on the other hand, is cognizant of the high node latencies in the region, and discovers k-bucket entries that provide more than 25% improvement in latency performance compared to PNS.

Nodes in the real world. Unlike the square setting where nodes are uniformly spread out, in the real world setting nodes are concentrated around certain regions in the world (e.g., Europe or North America). Moreover the node latencies are also chosen to reflect retrieval of large files [50, 46]. Fig. 4a shows the latencies for queries routed through the 1st k-bucket of an arbitrary node (in this case, the node is located in Frankfurt). Kadabra has 50% lower latencies compared to the original Kademlia protocol and 35% lower latencies compared to PNS and PR. This is because the baseline algorithms are not aware of the different node latencies of the peers, whereas Kadabra is able to focus its search on peers having low node latencies. As in the square case, Fig. 4b shows the benefit of Kadabra extends to the entire network.



(a) (b)

Fig. 5: Nodes in the real world: (a) Performance when there are demand hotspots. (b) Performance when 4% of nodes near New York City have above average node latencies.

Fig. 5a shows performance when there are demand hotspots. Compared to uniform demand, both PNS and PR worsen in performance increasing the gap to Kadabra. A similar trend is observed in Fig. 5b when we consider a region of nodes (near New York City in our experiments) and set their node latency to double the default average value. While even Kadabra shows a slight degradation, it is still more than 40% more efficient compared to PNS.

In addition, we evaluate the security of *Kadabra* by setting 20% of the nodes as adversarial, which deliberately delay queries passing through them by 3× their default node latencies. While all algorithms degrade in this scenario, Fig. 6a shows that when the adversarial nodes are located at random cities Kadabra discovers routes which avoid the adversarial nodes resulting in overall quicker lookups. Even when the adversarial nodes are concentrated in one region close to a victim node, Fig. 6b shows how a victim running Kadabra can effectively bypass the adversarial nodes while PNS takes a huge performance loss at more that 2× the latency of Kadabra.

6 Related Work

A great many number of prior works have studied how to speedup DHTs by being aware of the peer locations in the underlying physical Internet [28, 31]. However, all of these works propose hand-crafted heuristics which do not adapt to network heterogeneity. Using parallel lookups and increasing the number of content replicas are some of the early methods. R/Kademlia enhances Kademlia routing with recursive overlay routing instead of iterative routing from vanilla Kademlia [25]. Some algorithms utilize caching to accelerate lookups by identifying hotspots [22] and lowering the load on congested nodes [21]. Heck et al. [24]

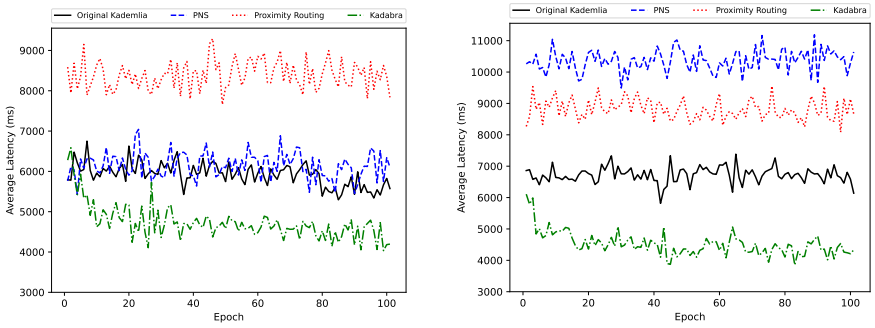


Fig. 6: Nodes in the real world: Kadabra outperforms baselines even when 20% of the nodes are adversarial in the network. (a) Adversarial nodes are randomly located. (b) Adversarial nodes are concentrated in one region close to the victim node. Performance is measured at the victim node.

evaluate the network resilience of Kademlia. Jain et al. [27] compare performance of various DHTs against measurement-based overlays. In Kanemitsu et al. [29], the authors propose KadRTT which uses RTT-based target selection and ID arrangement to accelerate lookups. Ratnasamy et al. [39] use landmark nodes and binning to optimize latencies in overlay networks. Steiner et al. [41] proposes an integrated content lookup protocol to reduce content retrieval times in Kad, a popular file-sharing application built using Kademlia. Stutzback et al. [44] advocate for parallel lookups and study optimal system parameters in Kad. Zhu et al. [49] presents a storage algorithm for Kademlia against load imbalance. To the best of our knowledge, Kadabra is the first effort to accelerate DHTs through a data-driven approach.

7 Conclusion

We have presented Kadabra, a decentralized data-driven approach to learning the routing tables in Kademlia for accelerated lookups. Unlike existing heuristics, Kadabra is cognizant to heterogeneity in network conditions resulting in routing tables that are tuned to the network and demand patterns. Our proposed protocol is also secure against Sybil, Eclipse and adversarial routing attacks. While these attacks are important, a thorough analysis of Kadabra's robustness against other known attacks [32] is a direction for future work. In our experiments, we observe Kadabra typically converges in a few epochs. Testing Kadabra's convergence and performance in a real world network (e.g., IPFS and Swarm) and obtaining a theoretical understanding on the convergence are also important directions for future work.

References

1. Ethereum Node Tracker, https://etherscan.io/nodetracker
2. Filecoin, https://filecoin.io/
3. Hypercore Protocol, https://hypercore-protocol.org/
4. Libp2p Kademlia, https://github.com/libp2p/go-libp2p-kad-dht
5. Report: DApp daily users surge to 2.4M in Q1 2022 despite headwinds, https://cointelegraph.com/news/report-dapp-daily-users-surge-to-2-4m-in-q1-2022-despite-headwinds
6. Safe Network, https://safenetwork.tech/
7. Swarm, https://www.ethswarm.org/
8. The IPFS-Filecoin Interface, https://github.com/filecoin-project/specs/issues/143
9. Website Load Time Statistics: Why Speed Matters in 2022, https://www.websitebuilderexpert.com/building-websites/website-load-time-statistics/
10. WonderNetwork Pings, https://wondernetwork.com/
11. Alabdulwahhab, F.A.: Web 3.0: The Decentralized Web Blockchain networks and Protocol Innovation. In: 2018 1st International Conference on Computer Applications & Information Security (ICCAIS). pp. 1–4. IEEE (2018)
12. Assadi, S., Wang, C.: Exploration with Limited Memory: Streaming Algorithms for Coin Tossing, Noisy Comparisons, and Multi-armed Bandits. In: Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing. pp. 1237–1250 (2020)
13. Baumgart, I., Heep, B., Krause, S.: OverSim: A Flexible Overlay Network Simulation Framework. In: 2007 IEEE global internet symposium. pp. 79–84. IEEE (2007)
14. Baumgart, I., Mies, S.: S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing. In: 2007 International conference on parallel and distributed systems. pp. 1–8. IEEE (2007)
15. Benet, J.: IPFS - Content Addressed, Versioned, P2P File System. arXiv preprint arXiv:1407.3561 (2014)
16. Bubeck, S., Cesa-Bianchi, N.: Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. Foundations and Trends® in Machine Learning **5**(1), 1–122 (2012)
17. Castro, M., Druschel, P., Hu, Y.C., Rowstron, A.: Exploiting Network Proximity in Distributed Hash Tables. In: International Workshop on Future Directions in Distributed Computing (FuDiCo). pp. 52–55. Citeseer (2002)
18. Cavenaghi, E., Sottocornola, G., Stella, F., Zanker, M.: Non Stationary Multi-Armed Bandit: Empirical Evaluation of a New Concept Drift-Aware Algorithm. Entropy **23**(3), 380 (2021)
19. Chen, Z., Huang, M., Tan, Q.: The Design of Kademlia System Base on Network Topology Matching. In: 2010 Second International Workshop on Education Technology and Computer Science. vol. 2, pp. 146–149. IEEE (2010)
20. Crosby, S.A., Wallach, D.S.: An Analysis of BitTorrent's Two Kademlia-Based DHTs (2007)
21. Einziger, G., Friedman, R., Kantor, Y.: Shades: Expediting Kademlia's Lookup Process. Computer Networks **99**, 37–50 (2016)
22. Guangmin, L.: An Improved Kademlia Routing Algorithm for P2P Network. In: 2009 International Conference on New Trends in Information and Service Science. pp. 63–66. IEEE (2009)

23. Gummadi, K., Gummadi, R., Gribble, S., Ratnasamy, S., Shenker, S., Stoica, I.: The Impact of DHT Routing Geometry on Resilience and Proximity. In: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. pp. 381–394 (2003)
24. Heck, H., Kieselmann, O., Wacker, A.: Evaluating Connection Resilience for the Overlay Network Kademlia. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). pp. 2581–2584. IEEE (2017)
25. Heep, B.: R/Kademlia: Recursive and Topology-aware Overlay Routing. In: 2010 Australasian Telecommunication Networks and Applications Conference. pp. 102–107. IEEE (2010)
26. Hildrum, K., Kubiatowicz, J.D., Rao, S., Zhao, B.Y.: Distributed Object Location in a Dynamic Network. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures. pp. 41–52 (2002)
27. Jain, S., Mahajan, R., Wetherall, D.: A Study of the Performance Potential of DHT-based Overlays. In: 4th USENIX Symposium on Internet Technologies and Systems (USITS 03) (2003)
28. Jimenez, R., Osmani, F., Knutsson, B.: Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay. In: 2011 IEEE International Conference on Peer-to-Peer Computing. pp. 82–91. IEEE (2011)
29. Kanemitsu, H., Nakazato, H.: KadRTT: Routing with network proximity and uniform ID arrangement in Kademlia. In: 2021 IFIP Networking Conference (IFIP Networking). pp. 1–6. IEEE (2021)
30. Karger, D.R., Ruhl, M.: Finding Nearest Neighbors in Growth-restricted Metrics. In: Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing. pp. 741–750 (2002)
31. Kaune, S., Lauinger, T., Kovacevic, A., Pussep, K.: Embracing the Peer Next Door: Proximity in Kademlia. In: 2008 Eighth International Conference on Peer-to-Peer Computing. pp. 343–350. IEEE (2008)
32. Koutrouli, E., Tsalgatidou, A.: Taxonomy of attacks and defense mechanisms in P2P reputation systems—Lessons for reputation system designers. Computer Science Review **6**(2-3), 47–70 (2012)
33. Maymounkov, P., Mazieres, D.: Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In: International Workshop on Peer-to-Peer Systems. pp. 53–65. Springer (2002)
34. Montresor, A., Jelasity, M.: PeerSim: A Scalable P2P Simulator. In: 2009 IEEE Ninth International Conference on Peer-to-Peer Computing. pp. 99–100. IEEE (2009)
35. Nobari, S.: DBA: Dynamic Multi-Armed Bandit Algorithm. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 9869–9870 (2019)
36. Pecori, R.: S-Kademlia: A trust and reputation method to mitigate a Sybil attack in Kademlia. Computer Networks **94**, 205–218 (2016)
37. Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In: Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures. pp. 311–320 (1997)
38. Putman, C., Nieuwenhuis, L.J., et al.: Business Model of a Botnet. In: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 441–445. IEEE (2018)
39. Ratnasamy, S., Handley, M., Karp, R., Shenker, S.: Topologically-Aware Overlay Construction and Server Selection. In: Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. vol. 3, pp. 1190–1199. IEEE (2002)

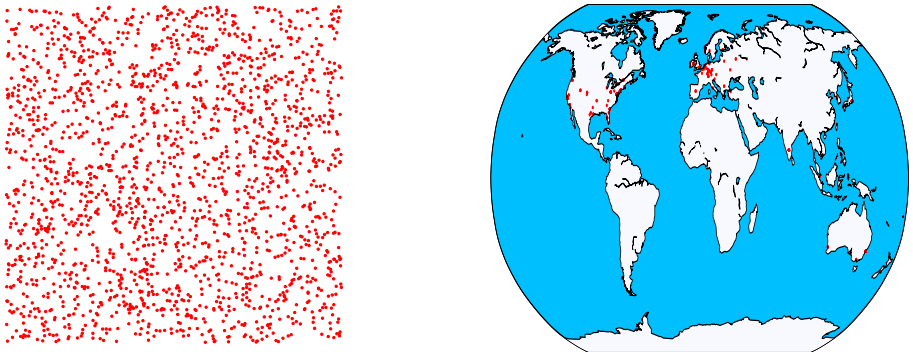
40. Slivkins, A.: Introduction to Multi-Armed Bandits. Foundations and Trends® in Machine Learning **12**(1-2), 1–286 (2019)
41. Steiner, M., Carra, D., Biersack, E.W.: Faster Content Access in KAD. In: 2008 Eighth International Conference on Peer-to-Peer Computing. pp. 195–204. IEEE (2008)
42. Steiner, M., En-Najjary, T., Biersack, E.W.: Exploiting KAD: Possible Uses and Misuses. ACM SIGCOMM Computer Communication Review **37**(5), 65–70 (2007)
43. Storj Labs Inc.: Storj: A Decentralized Cloud Storage Network Framework (2018)
44. Stutzbach, D., Rejaie, R.: Improving Lookup Performance over a Widely-Deployed DHT. In: Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications. pp. 1–12. IEEE (2006)
45. The Swarm Team: SWARM-Storage and Communication Infrastructure for a Self-Sovereign Digital Society (2021)
46. Trautwein, D., Raman, A., Tyson, G., Castro, I., Scott, W., Schubotz, M., Gipp, B., Psaras, Y.: Design and Evaluation of IPFS: A Storage Layer for the Decentralized Web. In: Proceedings of the ACM SIGCOMM 2022 Conference. pp. 739–752 (2022)
47. Trón, V.: The Book of Swarm: Storage and Communication Infrastructure for Self-sovereign Digital Society Back-end Stack for the Decentralised Web. V1. 0 pre-Release **7** (2020)
48. Zhao, B.Y., Joseph, A., Kubiatowicz, J.: Locality Aware Mechanisms for Large-scale Networks. In: Proceedings of the FuDiCo. vol. 2 (2002)
49. Zhu, L., Zheng, K.: An Improved Kademlia Algorithm Based on Qos. In: Proceedings of 2014 International Conference on Cloud Computing and Internet of Things. pp. 128–130. IEEE (2014)
50. Zichichi, M., Ferretti, S., D'Angelo, G.: On the Efficiency of Decentralized File Storage for Personal Information Management Systems. In: 2020 IEEE Symposium on Computers and Communications (ISCC). pp. 1–6. IEEE (2020)

A Nodes in a square - KBR

In this section, we present additional results for the KBR application when nodes are distributed randomly on a square.

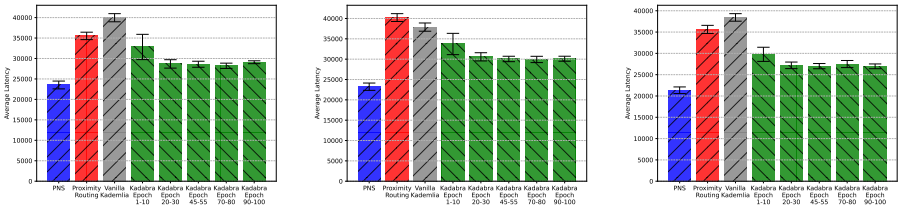
In §5.2, for uniform traffic demand we have presented how the average latency varies with epochs for an arbitrarily chosen node. To show that the presented behavior is general, and not occurring only at a few nodes, in Fig. 8 we show performance of Kadabra and baselines at five randomly chosen nodes. In all cases, we observe a similar qualitative behavior. Fig. 9 presents an example of the paths taken for a lookup from the same source to the same destination on different heuristics. Kadabra is able to achieve significantly lower path latency by choosing a relatively straight path with low node latencies. In the figure, node D's node latency is 1000. For vanilla Kademlia, node M1's node latency is 1400. For PR, node M1's node latency is 800 and M2's node latency is 1400. For PNS, node M1's node latency is 2000 and M2's node latency is 1400. For $Kadabra$, node M1's node latency is 100 and M2's node latency is 100.

Similarly, in Fig. 10 we plot the performance under hotspot demand measured from five randomly chosen nodes. Here too, we observe the qualitative behavior is the same across the five nodes. Fig. 11 shows the paths taken by

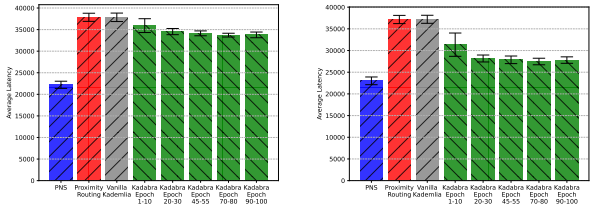


(a) (b)

Fig. 7: (a) 2048 nodes randomly distributed within a square. (b) 2048 nodes at cities around the world. A dot may represent multiple nodes due to high geographic concentration.

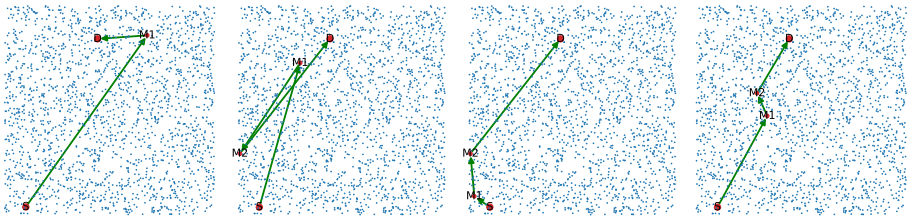


(a) Node A (b) Node B (c) Node C



(d) Node D (e) Node E

Fig. 8: Nodes in a square: We randomly sample five nodes in the square and compare the performance of *Kadabra* and the baseline algorithms at each node under uniform demand.



(a) Vanilla: 24225 (b) PR: 27124 (c) PNS: 21976 (d) *Kadabra*: 15674

Fig. 9: Nodes in a square: Paths and latencies (below the plots) of an example lookup. Kadabra is trained for 50 epoch under uniform demand. Mi refers to a node that is on the path but is not the source or destination node.

the different heuristics on the same sample query. Kadabra has been trained for 50 epochs. We observe Kadabra's path latency is $> 25\%$ more efficient than the original Kademlia protocol. In the figure, node D's node latency is 800. For vanilla Kademlia, node M1's node latency is 400 and node M2's node latency is 1100. For PR, node M1's node latency is 100 and node M2's node latency is 1800. For PNS, node M1's node latency is 1700 and node M2's node latency is 1200. For *Kadabra*, node M1's node latency is 400 and node M2's node latency is 400.

B Nodes in a square - DHT

Next, we consider the DHT application in which a (key, value) pair is stored on 3 nodes. When a node initiates a query for the key, it sends out queries on $\alpha = 2$ independent paths. The overall latency lookup latency is the time between sending out the queries and the earliest time when a response arrives on any of the paths. As in the KBR application, we consider three traffic settings:

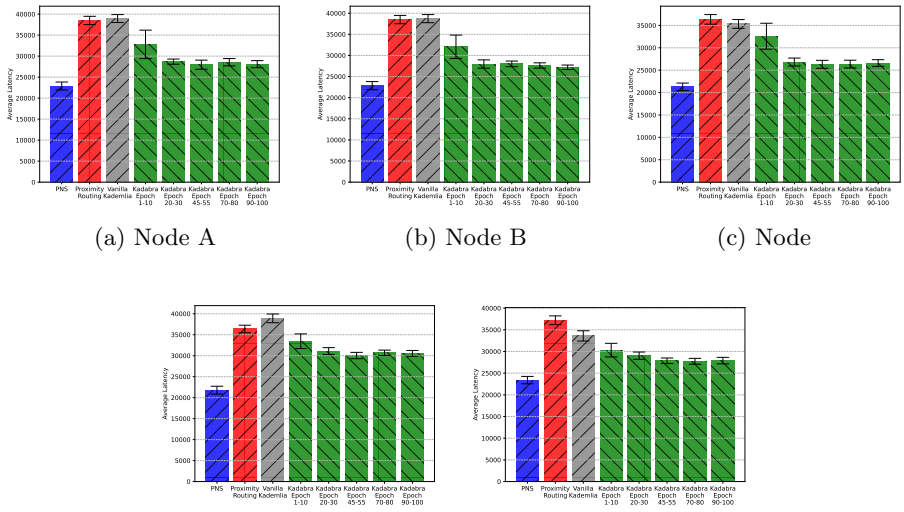
(1) DHT under uniform demand.

(2) DHT under demand hotspots.

(3) DHT under skewed network bandwidth.

These settings are similar to the KBR case, and hence we do not elaborate them. Fig. 12a and 12b show the performance, at an arbitrarily chosen node, of queries sent through the 1st k bucket under uniform demand and hotspot demand respectively. Similar to KBR, with hotspot nodes, the improvement we gain from *Kadabra* is larger than the scenario without hotspot nodes.

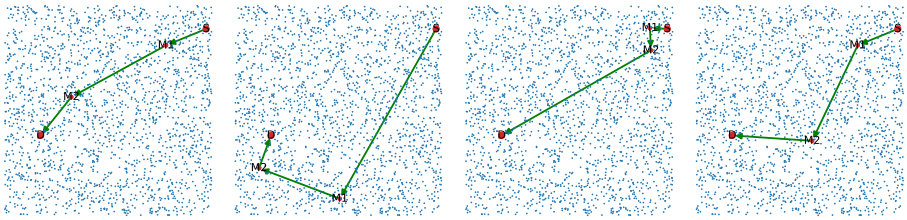
Fig. 13 shows the case where nodes within a small 2000×2000 region at the center of the square have high node latencies than default values. Unlike KBR, in the DHT case PNS and Kadabra are much closer.



(a) Node A (b) Node B (c) Node

(d) Node D (e) Node E

Fig. 10: Nodes in a square: We randomly sample five nodes in the square and compare the performance of *Kadabra* and the baseline algorithms at each node under demand hotspots.



(a) Vanilla: 25746 (b) PR: 21773 (c) PNS: 17186 (d) *Kadabra*: 18714

Fig. 11: Nodes in a square: Paths and latencies (below the plots) of a sample lookup under hotspot demand. The source node (S) and the destination node (D) are the same pair of nodes for all protocols. Mi refers to a node that is on the path but is not the source or destination node.

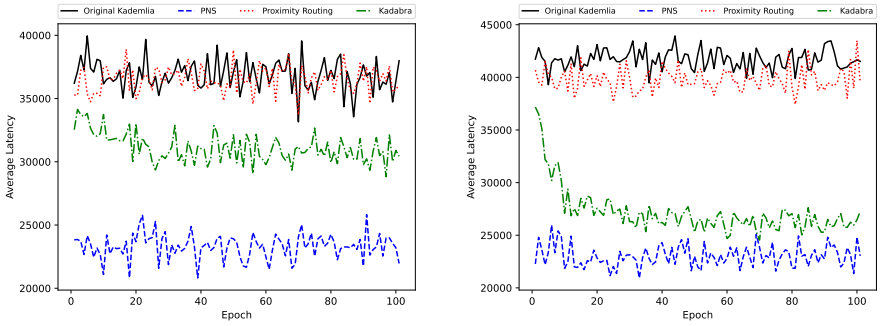


Fig. 12: Nodes in a square: (a) Performance under uniform demand. (b) Performance under demand hotspots.

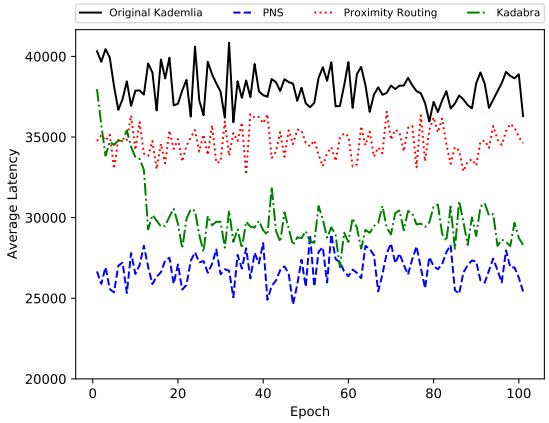
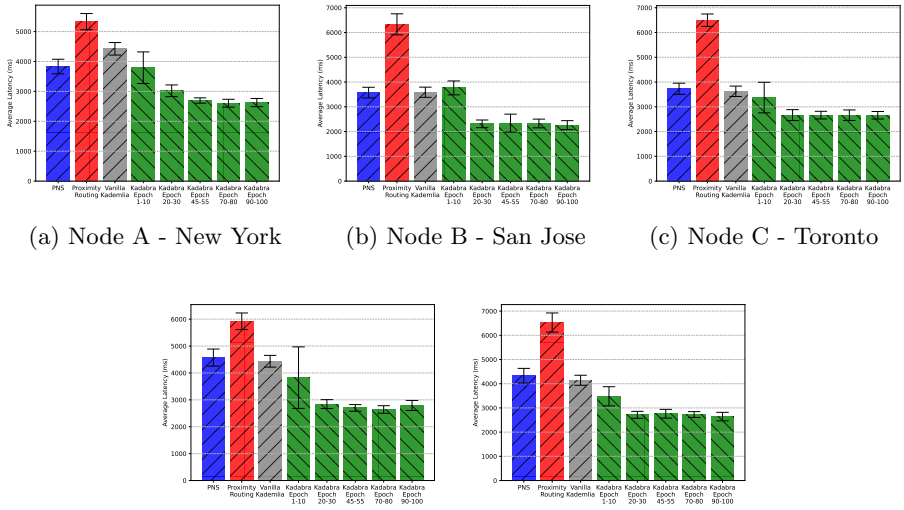


Fig. 13: Nodes in a square with DHT application: Performance when a region of nodes have higher node latencies than average. Measurements are taken from within the high node latency area.



(a) Node A - New York (b) Node B - San Jose (c) Node C - Toronto

(d) Node D - Frankfurt (e) Node E - London

Fig. 14: Nodes in the real world: Performance under uniform demand at five randomly sampled nodes around the world.

C Nodes in the real world - KBR

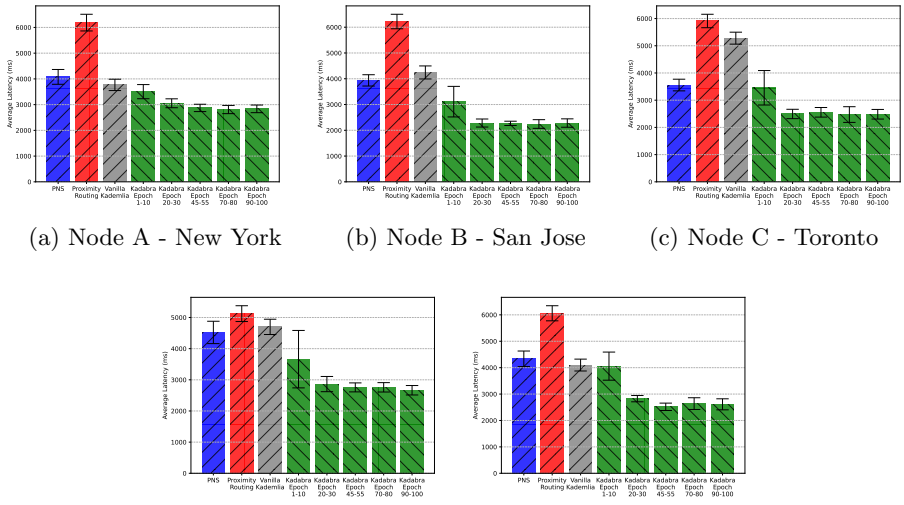
In this section, we provide supplementary results for the cases considered under KBR application when nodes are distributed over a real world geography.

Fig. 14 and 15 show performance at five randomly chosen nodes, for uniform demand and hotspot demand settings. The behavior observed in these plots are consistent with our discussion in §5.2.

Fig. 16a and 16b show the paths and their corresponding latencies of a sample query, under uniform traffic and hotspot traffic respectively. Some cities are repeated on the paths are there may be multiple nodes within the same city. The latency between two nodes in the same city is set to 1 ms in our experiments. Compared to baselines, Kadabra chooses efficient paths with less number of hops and low node and link latencies.

D Nodes in the real world - DHT

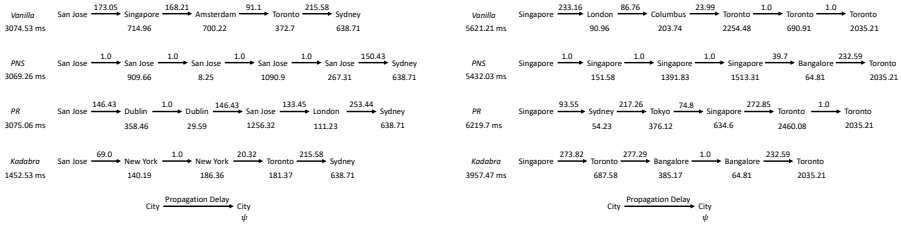
This section considers the DHT application for the setting of nodes in the real world. Fig. 17a and 17b show performance under uniform traffic and hotspot traffic demands respectively. In each case, Kadabra is exhibits lower latencies than the baselines. Kadabra is more than 30% faster compared to PNS in the two cases.



(a) Node A - New York (b) Node B - San Jose (c) Node C - Toronto

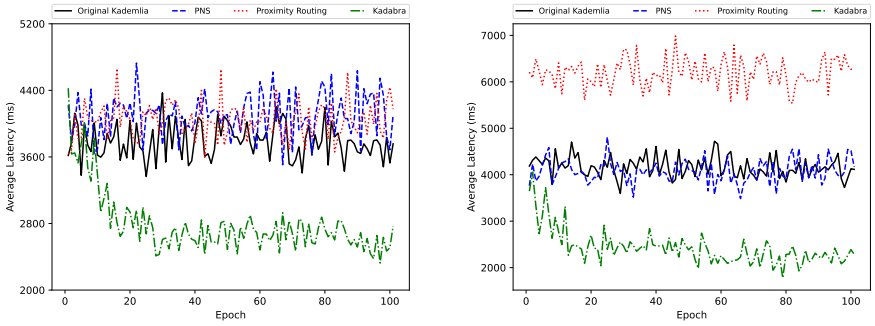
(d) Node D - Frankfurt (e) Node E - London

Fig. 15: Nodes in the real world: Performance under demand hotspots at five randomly sampled nodes around the world.



(a) (b)

Fig. 16: Nodes in the real world: Paths and latencies of a sample query under (a) uniform demand, and (b) hotspot demand.



(a) (b)

Fig. 17: Nodes in the real world with DHT application: (a) Performance under uniform demand. (b) Performance under hotspot demand.

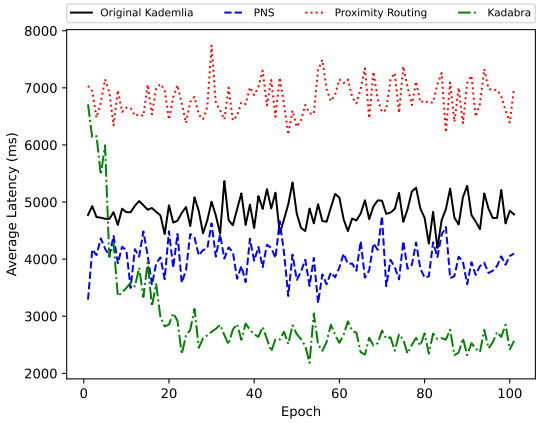


Fig. 18: Nodes in the real world with DHT application: 4% of nodes have a high node latency compared to network-wide average. Measurements are taken from one of the nodes within the high node latency area.

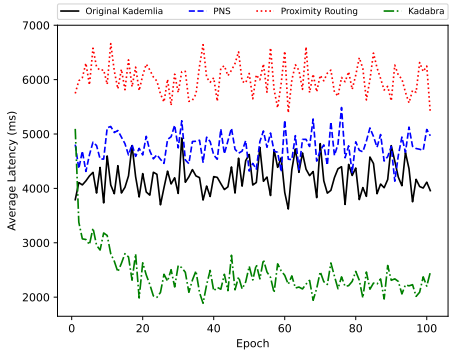


Fig. 19: We performed iterative routing on nodes in the real world with KBR application under uniform demand. The result suggests that *Kadabra* works effectively under iterative routing.

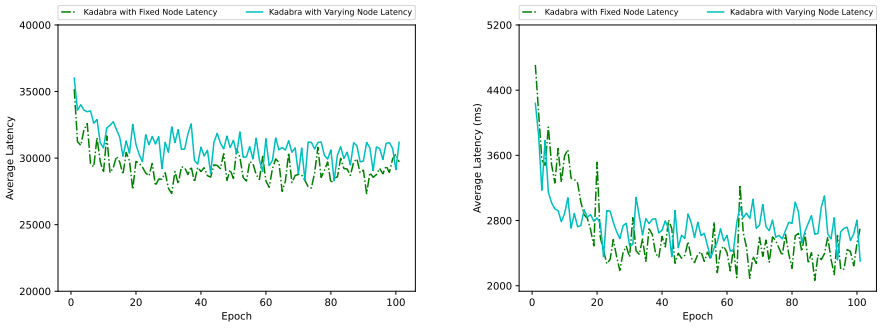
Fig. 18 considers a setting where a region of nodes around New York have node latency that is twice the default average value. Here too, we observe Kadabra outperforms the other baselines.

E Iterative routing

There are two options when it comes to routing in Kademlia: recursive routing and iterative routing. In recursive routing, the source node contacts the first hop node, and the first hop node then contacts the second hop node. In iterative routing, the first hop node returns the second hop node information to the source node, and the source node contacts the second hop node by itself. Although most current Kademlia implementations use recursive routing, vanilla Kademlia uses iterative routing. In our evaluations we have mainly discussed recursive routing. In this section, we consider iterative routing under uniform traffic demand when nodes are located on the real world. The result is shown in Fig. 19, where Kadabra outperforms the baselines.

F Network instability

Node latency at the same node may vary for different queries. In our evaluations we have used fixed node latencies (δ) for Kadabra nodes. To capture potential network instabilities, in this section, we evaluate Kadabra with varying node latencies. For each query, we add a random noise (random number from $[-0.05 \times \delta_u, +0.05 \times \delta_u]$) to the fixed node latency δ_u at a node u. We consider the KBR application under uniform demand for both nodes in a square and nodes in the real world. The result in Fig. 20a and 20b indicates that the impact of the varying node latency on Kadabra is insignificant. Specifically, for nodes in a square with



(a) (b)

Fig. 20: Average latency during each epoch for queries routed through the 1st k-bucket of an arbitrary node. KBR application with nodes that have noisy node latencies: (a) Nodes in a square. (b) Nodes in the real world.

varying node latencies, the performance of Kadabra (average latency for the last 10 epochs) decreases by at most 5%. For nodes in the real world with varying node latencies, the performance of Kadabra decreases by at most 8%.

Thus, we conclude that Kadabra is a robust algorithm that can adapt to a wide variety of situations.