No-hop: In-network Distributed Hash Tables

Lily Hügerich
TU Berlin
lily@inet.tu-berlin.de

Apoorv Shukla
Huawei Munich Research Center
apoorv.shukla1@huawei.com

Georgios Smaragdakis
TU Delft
g.smaragdakis@tudelft.nl

ABSTRACT

We make a case for a distributed hash table lookup in the network data plane. We argue that the lookup time performance of distributed hash tables can be further improved via an in-network data plane implementation. To this end, we introduce No-hop, an in-network distributed hash table implementation, which leverages the data plane programmability at line rate gained from P4. Our initial results of transporting distributed hash table logic from hosts' user space to the fast path of switches in the network data plane are promising. We show that No-hop improves the performance of locating the responsible host and maintains the properties of distributed hash tables while outperforming two baselines.

CCS CONCEPTS

• **Networks** → **Network protocol design**; **Data center networks**; **Bridges and switches**; **Programmable networks**; *In-network processing*; **Network algorithms**.

KEYWORDS

DHT, Scalability, P4, Data Plane Algorithms

ACM Reference Format:
Lily Hügerich, Apoorv Shukla, and Georgios Smaragdakis. 2021. No-hop: In-network Distributed Hash Tables. In *Symposium on Architectures for Networking and Communications Systems (ANCS '21), December 13–16, 2021, Layfette, IN, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3493425.3502757

1 INTRODUCTION

Distributed hashing was introduced more than 20 years ago to efficiently locate and store objects in a distributed system. A special case of distributed hashing, namely, consistent hashing, became very popular via its application to peer-to-peer systems [1–3] and content delivery networks [4, 5]. In consistent hashing, when the Distributed Hash Table (DHT) is resized, e.g., due to the churn of nodes, only a small number of nodes needs to be remapped. Moreover, each node does not need to maintain a routing table for all the nodes in the network. In popular implementations, e.g., Chord [1], each node keeps a state for only $O(log(N))$ other nodes in the system, where N is the total number of nodes. The lookup also terminates within a small number of forwarding steps. Later, Gupta et al. [2] proposed a technique, called One Hop, to store complete routing tables at each node. With this technique, they showed that one hop lookup is possible even for relatively large networks (100k nodes) with minimal bandwidth requirements for update messages.

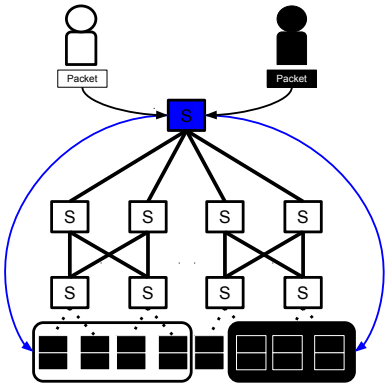

Figure 1: Illustration of two load balanced scalable clusters (in black and white) managed by one No-hop switch (blue).

Today, variations of the original Chord DHT or One Hop lookup systems are used by large-scale datacenter applications, e.g., Amazon's Dynamo [6] and Cassandra [7]. At the same time, datacenters are becoming increasingly programmable, e.g., with the use of the P4 language [8, 9] for implementing custom in-house functionalities. Such as, the work of Pegasus [10] which uses programmable TOR switches to coordinate the replication and location of popular items in server racks. In this paper, we investigate whether the dataplane programmability enabled by P4 can further improve lookup time and, thus, improve the performance of applications that run in programmable datacenters. In particular, we ask: *"Can we ensure faster key-value lookups compared to existing DHTs and One Hop lookups, by leveraging network programmability?"*. To answer this question, we implement an in-network key-value lookup system for DHTs and compare against two baselines, namely, Chord [1] and One Hop [2].

Distributed Hash Tables in a Datacenter Environment. DHTs are a distributed lookup system [1, 2]. Even though in the datacenter scenario we have a central controller, we argue that the properties of DHTs can be beneficial. Since the addition and subtraction of hosts can happen with minor disruption in Chord [1], which makes it an ideal candidate to implement scalable clusters in a datacenter. Furthermore, such clusters can be load balanced if a uniform (probability of each output value occurring is about the same [11]) hash function is chosen.

In this paper, we develop No-hop, for fast lookups in datacenters that use DHT logic by offloading the lookup mechanism to the dataplane via programmable switches. In Figure 1, we illustrate one of the implementations of No-hop running on only one programmable switch which can be deployed in a datacenter containing no prior programmable switches. It shows that two DHTs create two clusters (black and white) for working with two different user groups. The blue switch runs our system, No-hop, and is responsible for both sets of key-value lookups. The DHT node ID signify the responsibility of a node for a subset of the system's resources, this can either be computational or storage resources.

Existing DHTs such as our baselines, Chord and One Hop can also bring these benefits to our environment but the lookup of IDs is first fulfilled at a host [1, 2]. This leads to network and host resources being wasted as the packet travels first to the incorrect host and then the incorrect host has to lookup the ID and resend the packet. We argue that this is unnecessary in a centralized environment.

Challenges. The main challenge was to overcome the problem of a greenfield deployment which requires all the switches to be programmable but is extremely costly. To work around it, we develop two implementations. The *first implementation* has table sizes that stay consistent as the system scales. This implementation, however, relies on an architecture with only P4-enabled programmable switches. The *second implementation* (see Figure 1) requires only one P4-enabled switch in the entry path for all nodes. This switch maintains a full table with the respective ranges of all nodes. Given these options, No-hop can serve a variety of deployment scenarios.

Our Contributions:

- We identify a key set of challenges and opportunities in transporting DHT logic from the user space of hosts to the network data plane.
- We present No-hop system which leverages data plane programmability via P4 to offload host DHT lookups in the user space to the fast path in the P4 switches.
- Our experiments demonstrate that No-hop outperforms two baselines, namely, Chord and One Hop, by up to 383% and 68%, respectively.
- We release the No-hop software [12] including, two No-hop implementations for exclusively P4-enabled switches and hybrid (partially P4-equipped) systems.

2 BACKGROUND

2.1 Chord

Chord [1] is a DHT that maps a given key onto a Chord node where nodes are arranged in a ring. Each Chord node is responsible for a set of IDs ranging between its ID and the ID of the predecessor of the node. The predecessor is the Chord node that comes before that node in the ring. The ID partitioning system allows the convenient implementation of data lookup, lending Chord well to peer-to-peer applications.

Design Goals. Chord aims to address the problems of load balancing, availability, scalability, decentralization, and flexible naming in peer-to-peer applications. Each Chord node has an ID range for which it is responsible. This partitioning of the ID range leads to natural load balancing. It also facilitates availability because if one Chord node fails, its successor becomes responsible for the failed nodes as well. Furthermore, a ring in Chord allows for the seamless addition and removal of nodes which ensures scalability and resilience [13]. Chord is decentralized, this is facilitated by Chord node-join and stabilize processes.

Finger Table. A key aspect of Chord is the finger table. The use of finger tables reduces the otherwise linear search time for a Chord node storing a key to $O(log(N))$, where N is the number of nodes in a ring.

2.2 One Hop

Similar to Chord, One Hop is a DHT. The biggest difference to Chord is that instead of finger tables which only store a subset of the values One Hop stores all DHT member values [1, 2]. This leads to One Hop usually finding the correct host while only contacting one other host, hence the name One Hop [2]. The authors of One Hop argue that even though small tables ease organizational work with membership changes, the cost of the extra routing hops versus the one in One Hop is too high [2].

Design Goals. One Hop aims to create a manageable and scalable system where only one host has to be contacted before reaching the final destination [2]. If the single hop criteria cannot be achieved, it is referred to as a failed query in One Hop. Failed queries occur when the queried node does not receive notification of a change to the table [2].

2.3 P4 Language

P4 [8, 9] is a domain-specific language that allows data plane programmability by writing custom P4 programs executed on P4 (enabled) switches. These programs determine how the packets are processed by the P4 switches even when these switches are already deployed in a network. Furthermore, a controller manages and controls underlying P4 switches.

3 No-hop: SYSTEM DESIGN

To explain the idea of No-hop, we show in Figure 2 the potential path taken by a packet through the same network but compared between Chord (grey), One Hop (black), and No-hop (blue). All incoming DHT packets from all systems first are routed to one random host in the system, this random host is marked with a white triangle. The No-hop packet

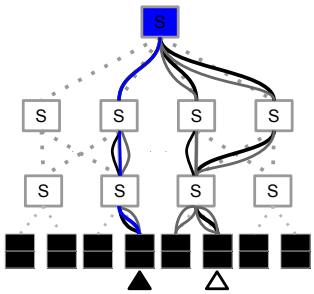

Figure 2: Example process of a packet moving to its final destination (black triangle) after being originally sent to a random member node (white triangle). The grey path represents the one taken by a packet in Chord, black a packet taken by One Hop and blue a packet taken by No-hop.

never reaches this random host because the No-hop switches direct the packet immediately to the correct host, resulting in zero host hops to the final location (marked with a black triangle). This is contrasted to our two baselines, Chord and One Hop. Where Chord takes two hops to reach the final location and One Hop one. This diagram is just one example of many of the potential paths incoming packets can take.

To achieve the key-value lookup in-network offloading of a DHT we implemented two versions. The first, No-hop Forward relies on a system with exclusively P4-enabled switches. The second, No-hop Rewrite only needs one P4-enabled switch in every packet's entry path. As illustrated in Figure 2, a No-hop Forward implementation would run on all switches while a No-hop Rewrite implementation would only need to run on the blue switch. Our two implementations can be compared to our two baselines. No-hop Forward like Chord has smaller tables. No-hop Rewrite uses, like One Hop, a complete routing table, which allows a single switch to handle the process but leads to a larger routing table. Both No-hop Forward and No-hop Rewrite have the same performance and route along the same paths but are different in the hardware needs and the table construction.

No-hop Rewrite. It only needs a subset of the switches to run the No-hop Rewrite P4 code, the rest can be classic IP packet routing switches. In No-hop Rewrite when a packet arrives at the switch running the P4 code, the switch looks up the correct value and rewrites the IP header of the packet to send the packet to the correct host. Snippet 1 illustrates the rewrite action in the No-hop Rewrite P4 program.

Snippet 1: Rewrite action of No-hop Rewrite

```
action dht_rewrite(bit<32> dht_address){
    hdr.ipv4.dstAddr=dht_address;
    }
```

The packet is then routed as an IP packet. If a packet does not pass a switch running the No-hop Rewrite P4 code on its way or if the responsible switch table has not been updated, the receiving host can lookup to send the packet to the correct host. The corresponding controller is responsible for filling the No-hop Rewrite tables which are the same for all programmable switches running the P4 code.

No-hop Forward. Unlike No-hop Rewrite, No-hop Forward needs all involved switches to be P4 enabled. No-hop Forward switches can without using IP send the packet to the correct host. Once a packet enters a No-hop Forward switch, the switch checks its table to see which port the packet should be sent out. Snippet 2 shows the P4 action for a No-hop Forward forwarding.

Snippet 2: No-hop Forward forwarding

```
action dht_forward(bit<9> port){
    standard_metadata.egress_spec=port;
    }
```

The corresponding controller fills the No-hop Forward tables which are all dependent on the switches' placement. The table for any switch A has, per group, match ranges for every neighbor of A. The ranges for every neighbor are the combined ID ranges of the reachable group hosts from that neighbor without traversing A. If two neighbors have the same ranges, the neighbors split the ranges. Generate_Range_Table(Switch A) shows the pseudo code for generating the ranges for the table of switch A in a single group scenario. A range match in the table returns the outgoing port to the corresponding neighbor.

```
Generate_Range_Table(Switch A):
    %RT[X]=Range table entry for neighbor X
    for Switch N ∈ Neighbors(A):
        for Host H ∈ Hosts:
            if A ∉ Shortest_Path(H,N):
                RT[N].add_ranges(H.ranges)
    for Switch N ∈ Neighbors(A):
        for Switch B ∈ Neighbors(A):
            if (B ≠ N) & (RT[B] = RT[N]):
                split(RT[B],RT[N])
    return RT
```

Group ID. No-hop has a Group ID to implement multiple DHTs in one system coordinated by one set of switches, such as in Figure 1, or the use case of scalable clusters. The Group ID is the ID of the DHT to which the packet should be routed. In Snippet 3 the lookup table for both No-hop Rewrite and No-hop Forward can be seen. The group ID has to fit exactly while the packet ID has to fit in the range of the responsible node.

Snippet 3: Lookup of Group ID and DHT node

```
table No_hop_lookup {
       key={
           hdr.dht.group_ID     : exact;
           hdr.dht.ID           : range;
        }
```

Node Join. It corresponds to the joining of another host for adding more hosts while the system is running. To join, a host needs a physical connection to a system switch. Then, the host sends a join message to the switch. The switch will then forward the packet to the controller which updates the switch tables accordingly. Afterwards, the controller sends a packet to the newly joined host with its ID.

Node Failure. Failures can happen in two ways. The first is an intentional failure. A host will leave the network and sends a failure message with its own ID to its TOR switch. Next, the switch forwards the message to the controller to update switch tables accordingly.
The second case is in the case of an unintentional failure. A host failed and did not send a prior failure message. The failure will be recognized by an adjacent host in the stabilization process. The notifying host will then send the failure message of the ID of the failed stabilization to its TOR switch. Finally, the TOR switch forwards it to the controller that updates the switch tables accordingly.

Stabilization. Every host sends a stabilization message periodically, i.e., after every stabilization time interval. The stabilization message is a simple lookup message (*S*) sent to the ID: (Own_ID+1) mod Max_ID. If a response is not received within the time limit, a failure message with the ID: (Own_ID+1) mod Max_ID is sent. Upon receiving a stabilize message, the host sends a lookup message with the ID: (Received_ID-1) mod Max_ID and *Ack* in the message body.

4 PROTOTYPE

The No-hop prototype includes the No-hop client, the P4 code that processes and forwards `No-hop Rewrite` and `No-hop Forward` packets, and the corresponding controllers which are written in python and follow the P4 runtime control plane specification [14].

No-hop Header. Table 1 illustrates the No-hop header. For our prototype, we use an ID_SIZE of 6, resulting in 64 different 6 bit IDs. The protocol includes messages of type $0-3$ resulting in a 2 bit `message_type` field. The message type 0 is for `first_contact` meaning the packet is entering the network and has no ID. No-hop assigns a packet with message type 0 an ID and forwards it to the corresponding host. To send a message the message type 1 should be used. The message types 2 and 3 are `failure` and `join`. Following, in the header is the packet ID field, with a length of ID_SIZE.

bit<2>	bit<ID_SIZE>
Message Type	**Packet ID**
0: `first_contact`	Empty if message type==0
1: `look_up`	$0 <$ Packet ID $\leq 2^{ID\_SIZE}$
2: `failure`	ID of failed node
3: `join`	

bit<Group_ID_SIZE>
Group ID
Defines which DHT the packet should be routed to.

Table 1: No-hop Header: ID_SIZE=6 & Group_ID_SIZE=8.

Lastly, the group defines to which DHT the ID is in reference. For more on group IDs refer to Section 3 (Group ID). Note, in our implementation No-hop is a layer four protocol.

Table Sizes. Both implementations of No-hop have table entries that contain a Group ID and one ID range. This results in one Group ID and two IDs (min ID and max ID) per entry. The implementation tables sizes differentiate because of the number of entries and the result of a match.

$$Group\_ID\_size = \lceil \log_2(\mathrm{G}) \rceil \tag{1}$$

$$ID\_size = \lceil \log_2(\mathrm{H}) \rceil \tag{2}$$

$$TS_f = \underbrace{(G \cdot (D+1))}_{\text{\# of entries}} \cdot \underbrace{(Group\_ID\_SIZE + (2 \cdot ID\_SIZE) + 9)}_{\text{Size of entries}} \tag{3}$$

$$TS_r = \underbrace{(G \cdot (H+1))}_{\text{\# of entries}} \cdot \underbrace{(Group\_ID\_SIZE + (2 \cdot ID\_SIZE) + 32)}_{\text{Size of entries}} \tag{4}$$

Where:

G = amount of groups
H = max hosts per group
D = # of neighbors with access to group hosts.
TS_f = No-hop `Forward` table Size (Bits)
TS_r = No-hop `Rewrite` table Size (Bits)

In No-hop `Forward`, all switches run the corresponding P4 code so each switch only needs to know the forwarding rules to its neighbors. In case of a match, the No-hop `Forward` table returns a nine bit port identifier. The resulting table size can be seen in Equation 3.

Since No-hop `Rewrite` is designed to require at minimum one switch to maintain the table, the switch tables must contain the location of all hosts in all groups. This results in as many entries as there are group hosts. Additionally, the reliance of No-hop `Rewrite` on IP packet forwarding results in matches returning an IP address (32 bit IPv4). The resulting table size can be seen in Equation 4.

Fault Tolerance. If a packet arrives at the wrong host because the host's ID changed midst routing, this would lead to a faulted routing. Depending on the use case of No-hop this fault could be handled differently. If No-hop is used as a classic DHT and the IDs are to certain data resources, the host

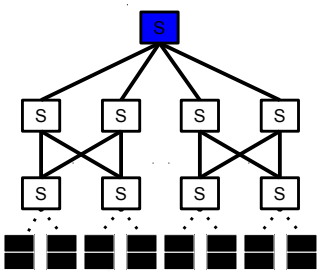

Figure 3: 4-ary Fat-Tree Test Topology.

would reprocess the packet and send it to the correct host resulting in a one hop lookup in case of a fault. If No-hop is used to divide, track and load balance stateless computation resources, this packet could still be processed by the host formerly responsible for the ID and the failure would be treated as a delayed ID change.

5 EVALUATION

The No-hop prototype runs in a mininet network [15] and uses the Simple Switch GRPC, a version of the BMv2 Simple Switch [16]. Our tests are conducted in a 64 bit Ubuntu virtual machine with 2048 MB base memory. The hosts run our server-client implementation which processes and sends No-hop, One Hop, and Chord packets. The host program is implemented using Scapy [17].

Test Topology. To evaluate our No-hop prototype, we test No-hop on a tree topology with 9 switches and 8 hosts. Figure 3 shows the test topology. In No-hop Forward all switches run the corresponding P4 code while in No-hop Rewrite only the blue switch runs the corresponding P4 code.

Baselines. To compare No-hop, we implement two baselines, Chord and One Hop. In our baselines, IPv4 packets are used. The controller computes the shortest path for all the baseline packets using the breadth first search algorithm as all links in our test topologies have equal weight.

Test Packets. We evaluate the benefits of No-hop by comparing with the two baselines on two performance metrics. These metrics are: (i) number of switch traversals and (ii) time. Switch traversals can be seen as an indicator of the network resources used by the incoming packet. In both evaluations, we send 16 packets of every possible 64 (2^{ID_SIZE}) packet IDs, resulting in 1024 packets for each system.

Switch Traversal. Figure 4 shows three heatmaps representing switch traversals. Switch traversals are the number of times the packet passed through a switch. The closer the heatmap is to white, the fewer switch traversals occurred. As can be seen, all packets in No-hop have exactly 3 switch traversals, this is because the switches in No-hop immediately route the packet to the correct host, and all hosts are

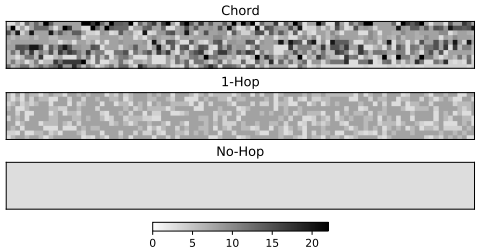

Figure 4: Switch traversals per lookup for first 1000 test packets of every implementation. No-hop has exclusively 3 hops while Chord and One Hop have on average ≈ 9 and ≈ 6.3 switch traversals, respectively.

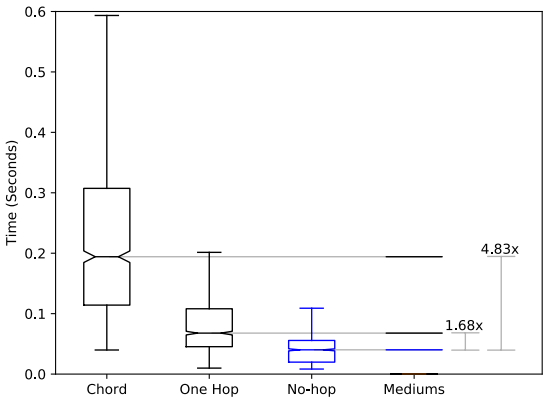

Figure 5: Box plot Graph without outliers for packet lookup time to responsible host, 1024 packets per data set. Notches depict the 95% confidence intervals. Whiskers area maximum length of 1.5× the interquartile range. No-hop shows a 4.83× and 1.68× speedup to Chord and One Hop respectively.

reachable with three switch traversals. As can be seen some packets in One Hop and Chord also only take 3 hops, this is the case when the correct host is chosen randomly by the client. One Hop follows No-hop in switch traversal performance. The highest amount of switch traversals were observed in Chord with some packets experiencing greater than 20 switch traversals.

Performance Evaluation. To test lookup time, we send the same set of packets with a 10 milliseconds interval between each. Figure 5 illustrates the results of this test. The results correspond to the findings of the switch traversals per lookup in Figure 4. The median lookup time of packets in No-hop is up to 1.68× faster than One Hop and 4.83× faster than Chord. We observe performance improvement from Chord to One Hop because all nodes in One Hop maintain full routing tables and thus, can fulfill the lookup with one hop. This comes at the expense of larger tables [2]. No-hop Forward, however, benefits from an efficient table size while still improving on the performance of One Hop. To see more on the difference in table sizes between No-hop Forward and No-hop Rewrite, refer to Section 4 (Table Sizes).

6 RELATED WORK

Distributed Hash Tables. Distributed hash tables (DHT) are often used in distributed file systems [18], especially peer-to-peer systems. Chord [1] was first introduced in 2001 and is still in use. While Chord has a ring topology, other DHTs [2, 3, 13, 19–23] can be implemented in a variety of network topologies, including trees (Tapestry [23]), XOR (Kademlia [3]), butterfly (Viceroy [21]), and hybrid (Pastry [22]) topologies. Some DHTs focus on minimizing the number of hops and thereby latency [2, 6, 24–26]. Examples include our second baseline One Hop [2]. This is often achieved by each node maintaining full routing tables to minimize the hops [2, 6, 24]. Others use a replication framework additionally to a classic DHT to enable low latencies [25] or a combination of subdivision and replication within subdivisions [26]. Still, to the best of our knowledge No-hop has been the only DHT system to consistently need no hops in the lookup process.

Distributed File Systems. Distributed file systems (dFT) are a key component of cloud computing [27, 28]. Amongst the dFTs that are based on DHT principles is the Amazon file system Dynamo [6, 7, 29–33]. Dynamo and others can be seen as a one hop DHT since every node has access to the locations of all others [6, 7, 24, 29, 30]. These systems' performance can be improved by incorporating the offloading of the lookup process to a programmable switch to reduce further the hops required to locate an ID.

In-network key-value Lookup. Recent work studies in-network caches that also benefit from an offloaded key-value lookup system [10, 34–37]. However, the focuses of these works are different than this of No-hop. Zhu et al. [37] focus on maintaining consistency at line rate and Liu et al. [35] enable in-network computing. The work presented in [34, 36] focuses on in-network caching. Lastly, the work of Jialin Li et al. [10] uses programmable switches to load balance and keep track of replicated items within server racks. This work requires that the clients know the home server location of the requested item so it does not form a lookup. No-hop however is a lookup for multi peer systems in the network data plane that also has properties which lead to the natural load balancing across all network resources including network links. Although these works also benefit from load balancing, their focus is not on offloading for a DHT.

In-network Applications. The advancement of a custom data plane facilitated by data plane programming via P4 has led to new data plane applications. P4xos [38, 39] proposes moving the Paxos consensus algorithm [40] to the data plane. One of the P4xos contributions is moving the logic that allows for fault tolerance to the data plane. Other P4-based in-network applications include load balancers [41], data center switches [42], network-based, e.g., LPM (longest prefix match) switches [43], multimedia traffic routing based on RTP (Real-Time Transport Protocol) timestamps [44], automated troubleshooting have been addressed in the data plane in [41, 45–52], and network monitoring [53–55].

7 DISCUSSION

Choice of Hash Function. The choice of a hash function is crucial for all types of DHT implementations. Cryptographic properties and aspects such as collision resistance [11, 56] should be considered to identify a suitable hash function.

Testing Limitations. In this paper, we have not exhaustively investigated the use of No-hop in diverse network architectures. Our current implementation of No-hop relies on prior knowledge of the network topology. Additionally, we have not yet tested No-hop on a hardware implementation.

8 CONCLUSION AND FUTURE WORK

We have presented No-hop that leverages the P4 language for a data plane implementation of DHTs. No-hop moves the lookup to the network data plane and, thus, processes packets in the fast path of the network data plane via switches instead of the user space of the hosts. Our experiments show that No-hop leverages the benefits of DHTs in a centralized datacenter environment, while outperforming DHTs in terms of lookup time.

eBPF Implementation and Smart NIC Offloading. In the future, we plan to implement No-hop in eBPF and XDP. This would allow a single end user to benefit from No-hop without prior knowledge of the network [57–59]. Afterwards, we would look into offloading via Smart NICs [60, 61]. Offloading eBPF programs to Smart NICs has already proven to be beneficial [62]. This will allow us to compare No-hop at different levels of the lookup process.

Evaluation of Churn Impact In our future work we also plan to evaluate the effect of churn on our system. No-hop like our baselines has the capabilities to handle the rapid joining and leaving of hosts while routing packets, however we have neither evaluated the effect of this in relationship to the lookup time in No-hop nor the baselines.

Hardware Implementation. As a part of our future agenda, we plan to apply No-hop on commercial-grade P4 programmable switches and networks to report on our experience. Accordingly, we believe No-hop Rewrite will be more feasible to implement on hardware as it only requires one programmable switch. No-hop Forward, however, will require more hardware resources. Via a hardware implementation of No-hop, we plan to investigate the benefits of No-hop in real hardware deployment.

ACKNOWLEDGEMENT

This work and its dissemination efforts are partially supported by the European Research Council (ERC) Starting Grant ResolutioNet (ERC-StG-679158) and BMBF BIFOLD (01IS18025A and 01IS18037A).

REFERENCES

[1] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[2] A. Gupta, B. Liskov, and R. Rodrigues. One Hop Lookups for Peer-to-peer Overlays. In *HotOS*, 2003.

[3] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, 2002.

[4] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, 1997.

[5] B. M. Maggs and R. K. Sitaraman. Algorithmic Nuggets in Content Delivery. *ACM CCR*, 45(3), 2015.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[7] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2), 2010.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. In *ACM CCR*, 2014.

[9] P4 Language Consortium. $P4_{16}$ language specs, version 1.1.0, 2018.

[10] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *USENIX OSDI*, 2020.

[11] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption*, 2004.

[12] L. Hügerich. No-hop Software. https://github.com/lilyhuegerich/No-hop, 2020.

[13] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *ACM SIGCOMM*, 2003.

[14] P4Runtime. https://p4.org/p4-runtime/.

[15] Mininet. http://mininet.org/.

[16] P4 Language Consortium. Simple Switch GRPC. https://github.com/p4lang/behavioral-model/tree/master/targets/simple_switch_grpc.

[17] Scapy. https://scapy.net/.

[18] C. Wu, V. Sreekanti, and J. M. Hellerstein. Autoscaling Tiered Cloud Storage in Anna. 12(6), 2019.

[19] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. *SPAA*, 1999.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, 2001.

[21] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *SOSP*, 2002.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[23] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE JSAC*, 2004.

[24] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *ISPDC*, 2013.

[25] V. Ramasubramanian and E. G. Sirer. Beehive: O (1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlay. In *NSDI*, 2004.

[26] I. Gupta, K. Birman, P. Linga, A. Demers, and R. Van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *IPTPS*, 2003.

[27] J. Balasangameshwara and H. L. Chandrakala. Performance-Driven Load Balancing for Distributed File Systems in Clouds. *International Journal of Computer Applications*, 975.

[28] S. Ghemawat, H. Gobioff, and S-T. Leung. The Google file system. In *SOSP*, 2003.

[29] Apache Cassandra. Overview. https://cassandra.apache.org/doc/latest/architecture/overview.html.

[30] L. Monnerat and C. L. Amorim. An effective single-hop distributed hash table with high lookup performance and low traffic overhead. *Concurrency and Computation: Practice and Experience*, 27(7), 2015.

[31] Riak. Riak core. https://github.com/basho/riak_core.

[32] B. Agapiev and I. Agapiev. DHT-based distributed file system for simultaneous use by millions of frequently disconnected, world-wide users, 2010. US Patent 7,716,179.

[33] Project Voldemort. Voldemort is a distributed key-value storage system. http://www.project-voldemort.com/voldemort/.

[34] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *SOSP*, 2017.

[35] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. Incbricks: Toward in-network computation with an in-network cache. In *ASPLOS*, 2017.

[36] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *FAST*, 2019.

[37] H. Zhu, Z. Bai, J. Li, E. Michael, D. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *VLDB*, 13(3), 2019.

[38] H. Dang, P. Bressana, H. Wang, K. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. P4xos: Consensus as a network service. In *IEEE/ACM Transactions on Networking*, volume 28. IEEE, 2020.

[39] NetPaxos. https://github.com/usi-systems/p4xos-public.

[40] L. Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, 2019.

[41] P4 Language Consortium. Load Balance. https://github.com/p4lang/tutorials/tree/master/exercises/load_balance.

[42] switch.p4. https://github.com/p4lang/switch.

[43] P4 Language Consortium. basic. https://github.com/p4lang/tutorials/tree/master/exercises/basic.

[44] FOX Networks Engineering & Operations Advanced Technology. Switching media streams based on RTP timestamps in P4. https://github.com/FOXNEOAdvancedTechnology/ts_switching_P4, 2016.

[45] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[46] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, M. Townsley, and T. Clausen. Stateless load-aware load balancing in p4. In *ICNP*, 2018.

[47] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *ACM SOSR*, 2016.

[48] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid. Runtime Verification of P4 Switches with Reinforcement Learning. In *ACM NetAI*, 2019.

[49] A. Shukla, S. Fathalli, T. Zinner, A. Hecker, and S. Schmid. P4CONSIST: Towards Consistent P4 SDNs. In *IEEE Journal on Special Areas in Communication (JSAC)- NetSoft*, 2020.

[50] A. Shukla, K. Hudemann, Z. Vági, L. Hügerich, G. Smaragdakis, A. Hecker, S. Schmid, and A. Feldmann. Fix with P6: Verifying Programmable Switches at Runtime. In *INFOCOM*, 2021.

[51] A. Shukla, S. Schmid, A. Feldmann, A. Ludwig, S. Dudycz, and A. Schuetze. Towards transiently secure updates in asynchronous sdns. In *SIGCOMM*, 2016.

[52] A. Shukla, S. J. Saidi, S. Stefan, M. Canini, T. Zinner, and A. Feldmann. Towards Consistent SDNs: A Case for Network State Fuzzing. In *IEEE Transactions on Network and Service Management*, 2019.

[53] Int specification. https://github.com/p4lang/p4-applications/blob/master/docs.

[54] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, 2017.

[55] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster. P4v: Practical Verification for Programmable Data Planes. In *ACM SIGCOMM*, 2018.

[56] E. Andreeva, B. Mennink, and B. Preneel. Open problems in hash function security. In *Designs, Codes and Cryptography*. Springer, 2015.

[57] M. Fleming. A thorough introduction to eBPF. *Linux Weekly News*, 2017.

[58] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[59] N. Van Tu, J-H. Yoo, and J. W-K. Hong. Accelerating Virtual Network Functions With Fast-Slow Path Architecture Using eXpress Data Path. *IEEE Transactions on Network and Service Management*, 17(3), 2020.

[60] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *NSDI*, 2018.

[61] K. Sabhanatarajan, A. Gordon-Ross, M. Oden, M. Navada, and A. George. Smart-NICs: Power proxying for reduced power consumption in network edge devices. In *IEEE ISVLSI*, 2008.

[62] J. Kicinski and N. Viljoen. eBPF Hardware Offload to SmartNICs: cls_bpf and XDP. In *NetDev*, 2016.