Sagkriotis, S. and Pezaros, D. (2022) Scalable Data Plane Caching for Kubernetes. In: 2022 18th International Conference on Network and Service Management (CNSM), Thessaloniki, Greece, 31 October - 4 November 2022, pp. 345-351. ISBN 9783903176515 (doi: 10.23919/CNSM55787.2022.9964497).

This is the Author Accepted Manuscript.

There may be differences between this version and the published version. You are advised to consult the publisher's version if you wish to cite from it.

http://eprints.gla.ac.uk/279331/

Deposited on: 13 September 2022

Enlighten – Research publications by members of the University of Glasgow
http://eprints.gla.ac.uk

Scalable Data Plane Caching for Kubernetes

Stefanos Sagkriotis
School of Computing Science
University of Glasgow
Glasgow, UK
s.sagkriotis.1@research.gla.ac.uk

Dimitrios Pezaros
School of Computing Science
University of Glasgow
Glasgow, UK
dimitrios.pezaros@glasgow.ac.uk

***Abstract*—Computation offloading to the programmable data plane enabled the acceleration of key-value stores which offer coordination services for large-scale data centres. Previous research reduced the response latency of key-value requests by half through deploying the store in the programmable data plane. In this work, we examine Kubernetes' central store, etcd, as a candidate for deployment in data plane. We discuss performance and scalability limitations existing in the default architecture of Kubernetes and how these can be alleviated through data plane offloading. Moreover, we investigate previous design decisions of in-network caching mechanisms that led to increased traffic generation and latency. We propose a new in-network key-value store platform that maintains strong consistency and fault-tolerance while improving performance and scalability over the state-of-the-art.**

I. Introduction

Advances in the area of programmable switches, both in programming tools (P4 [1], PSA [2]) and in hardware implementations (Intel Tofino [3], Broadcom Trident [4] have allowed line-rate performance for compiled binaries executed on programmable switches. These enablers have driven innovation in the area of in-network compute. Researchers have used such technology to offload computation primitives in Programmable Data Plane (PDP), significantly improving the performance of the applications that rely upon these primitives [5]–[8].

For Key Value Stores (KVSs), computation offloading in PDP not only enables line-rate generation of replies but effectively reduces the amount of hops necessary in a query's path. The Round-Trip latency is reduced in half by generating a reply from the first network device in the path instead of traversing all the way to the server that stores KV pairs. Given the instrumental role of KVSs in providing configuration management [9], locking mechanisms [10], and web-service related operations in large-scale data centres [11], the potential performance improvement from PDP deployment is significant.

Apart from standalone deployments, KVSs operate as part of other orchestration frameworks, e.g., Kubernetes. Kubernetes utilised virtualised computations to enable service deployment across a versatile set of computing nodes. With its extensible and fault-tolerant architecture it has driven the transition to the microservices era for numerous mainstream operators. Central to its architecture is etcd [9], a fault-tolerant, consistent KVS that provides coordination services and is used as the backup store for all of Kubernetes' control-plane components.

However, it has been shown that etcd presents scalability bottlenecks [12]. Etdc relies on a consensus-based approach to ensure consistency among the participating nodes. This approach requires a growing amount of time to confirm that changes have been committed in the majority of the participating nodes, which in turn creates increased response time. In this work we examine the workloads imposed to etcd by Kubernetes from the scope of PDP deployment. Based on our analysis we proceed to suggest an extended Kubernetes architecture that offloads part of the KV traffic to PDP devices.

NetChain [13], one of the most prominent works in the area of in-network caching, by accommodating queries entirely in data plane is effectively the fastest in-network KV platform existing today. Other important works perform offloading of certain KV processes, such as conflict detection [14], [15], which offer performance improvements over legacy storage. Through comparison with NetChain other platforms seem to be inferior in terms of maximum throughput and latency making NetChain the fastest available framework as of now.

While being the fastest in-network KVS, NetChain still presents some limitations that make it less appealing for large-scale deployments in a data centre environment. Its CR mechanism in order to maintain per-item consistency among the participating nodes requires full chain traversals to fetch values from the appointed reference node. We analyse and evaluate the impact of this design decision in performance and reveal its shortcomings.

To surpass the shortcomings of NetChain's design we use NetCRAQ, a novel in-network KV platform which builds upon the state-of-the-art and improves average read throughput by approx. 130% compared to NetChain. We argue that the integration of PDP devices in Kubernetes enhances end-to-end programmability and propose NetCRAQ's integration to Kubernetes by extending Kubernetes' original architecture. This new architecture addresses previous scalability and performance constraints stemming from etcd's integration and benefits pod scheduling and scaling performance.

Overall, this paper contributes by:

- Analysing the behaviour of Kubernetes with respect to etcd during the deployment and scheduling of services.
- Identifying weaknesses and performance limitations of KV platforms that operate in PDP.

- Proposing a new in-network KV platform that shows performance and scalability improvements when compared to the state-of-the-art.
- Analysing a new storage replication mechanism as the main tool to offload Kubernetes related KV pairs.
- Extending Kubernetes' architecture to integrate data plane devices and improve the performance of core Kubernetes components, i.e., etcd.

II. Background

KV replication methods are commonly separated between quorum-based and primary-backup methods. Quorum-based methods require the majority of the participating nodes to conduct an operation in order to render it successful (e.g., Paxos [16]). Primary-backup methods operate by appointing one of the participating nodes as the primary (or reference) node. Operations are considered successful when they are executed on this reference node. The rest of the nodes act as replicas of the primary one (e.g., Chain Replication (CR) [17]).

A. Chain Replication

Under CR, the participating nodes/programmable switches form a chain and each has a distinct role: head, tail, or replica. All of the participating nodes hold the same KV pairs. Write queries originate from the head and then propagate across all replica nodes until they reach the tail. The tail issues a response which acts as an acknowledgement for the write. A tail node is also responsible for responding to read queries. Only the tail is considered to be up-to-date with the latest commit for a value and acts as a reference point for the entire chain. Replicas can replace the head or the tail in case of a failure.

Defining the tail as the reference point allows per-key consistency for the entire chain to be achieved. When a write query reaches the tail, it has certainly been processed by all previous chain nodes. Therefore, all chain nodes are updated with its latest version. If the write query is lost before reaching the tail, then all subsequent reads will be replied with the previous version for this object. This ensures consistency in replies.

B. Transferring KVS in PDP

The IncBricks [18] paper was among the first to capture the potential of PDP devices because of their central location in a query's message path. The message path for a query in a legacy KVS would start from the client, go through network devices, and finally it would reach the servers that host the KVS. IncBricks substituted the need to reach coordination servers for some types of queries and suggested a shorter path for cached values which stays between the client and the network devices. However, the use of programmable network processors offered limited instructions and provided limited programmability that was dependent on the manufacturer.

NetChain [13] used the same approach but through the use of P4 [1] managed to deploy an in-network KVS in high-performance ASICs (instead of Network Processor Units (NPUs)), and therefore achieved greater performance than IncBricks. Its design realised that queries can be processed in PDP with minimal interactions with the control plane, as part of a fully deployed replication method.

The query response mechanism that was employed relies heavily on the incoming packets that are processed using the match-action pipeline [2]. A custom packet format was used, layered over UDP, which contained the following fields: `OP` - the type of operation (read, write), `KEY` - the ID of the object in question, `VALUE` - its value, `SEQ` - a monotonically increasing sequence number that mitigates out-of-order deliveries, `SC` - the number of chain nodes in the header, $\mathtt{S_k}$ - IP of the $\mathrm{k^{th}}$ participating node. Storing the IPs of the nodes in the header aims at reducing the amount of stored data per switch and allowing dynamic mapping of data to chains.

We notice that the suggested packet structure can add a significant amount of overhead bytes, especially for bigger chains where all the participating node IPs have to be added in the header. For a 4-node chain, NetChain's header is 58 bytes, and grows by 32 bits with every node addition. The linear growth of the packet size with the chain length can cause increased parsing times and adds complexity when fields need to be added or removed [19]. Another issue arises from the use of a monotonically increasing value in the `SEQ` packet field, which is 16 bits by default. This size allows just $65,536$ operations before the field overflows.

The reasoning behind the choice of CR as the main replication method for NetChain reflects the limitations and features of the deployment environment and provides important lessons. Firstly, CR has small redundancy requirements to achieve fault tolerance: to survive f node failures, it requires $f+1$ nodes. This is a significant reduction when considering that it translates to the amount of programmable switches in use. Secondly, CR presents low implementation complexity by requiring a simple commit & forward pipeline to execute a write query among the chain nodes. A quorum-based approach would require several RTTs to reach consensus on a successful write or respond to a read, which would increase implementation complexity.

While the choice of CR as an in-network replication method displayed superior performance over legacy KVSs, we observe some performance-limiting factors. Based on the principle that only the tail can reply to read queries, the amount of generated packets is substantial: for n participating nodes, $2n$ packets are required for read queries and $n+1$ for write queries. NetChain, by employing CR, adopts this design which, in the context of a data centre environment, has the following limitations: 1) generating messages for the tail results in high packet gain for the platform. It requires network resources for extensive parsing and forwarding; 2) the chain's reply rate is limited to the throughput that can be provided by the tail node, being the only one responsible to reply. This heavily harms scalability; 3) directing all queries to a certain node can also be a root cause for hot-spots within the topology; 4) the response latency increases linearly with the chain length because of the

increasing number of hops.

C. CRAQ

Chain Replication with Apportioned Queries (CRAQ) [20] operates in a similar manner to CR: the nodes formulate a chain and each node can be a head, tail, or replica. The key differences with CR are: CRAQ's ability to distribute load across all chain nodes – effectively enhancing scalability, and its ability to operate under relaxed consistency guarantees to benefit performance.

In CRAQ, each KV pair can be either clean, in which case the there are no pending commits for its value, or dirty, which means that there is a most recent commit which is yet to be acknowledged by the tail. Therefore, multiple versions of a value might correspond to a key. CRAQ places this information inside each participating node. Upon receiving a read query, each node can either: respond to it, if the version is clean, or redirect the query to the tail in order to fetch the latest version. Writes operate similarly to CR: a node has to propagate a write down the chain until it reaches the tail and then be acknowledged as the latest clean version. Once this happens, the rest of the chain nodes are notified and can delete previous versions of this object.

D. Kubernetes requests on etcd

To find out the type of requests directed to etcd, we operate a Kubernetes cluster with four nodes while monitoring various metrics of the generated requests. Stateless services are deployed in the form of containers which in Kubernetes are further enclosed within pods. We then proceed to scale these pods equally among the participating nodes. The pods are originally 2 and then scale up to 25. After the deployment has finished successfully we proceed to delete all of them.

The results are the average metrics as obtained over multiple runs of the same experiment. The majority of the requests were read queries (known as ranges in etcd) of single KV pair – approx. 15.3k requests. The write requests were just 2.9k, which is 16% of the total number of queries. Most of the read requests, 55%, were directed to just 25 KV pairs which concerned health requests, leases, and scheduler values. There were 3.1k consensus proposals, all of which were successfully conducted.

To establish the performance comparison baseline, we set up an etcd benchmark with the characteristics of the suggested deployment by Kubernetes: 3 nodes, 128bit values, 1 client with multiple connections. We used the default benchmark tool for etcd to establish these parameters. The results showed an average write duration of 0.21s and an average read query duration of 0.7ms. These numbers are orders of magnitude higher that what can be offered by PDP platforms like NetChain.

The obtained metrics reveal a read-mostly workload that is skewed towards a small subset of KV pairs. The amount of consensus proposals appears to be significant considering that each consensus involves multiple RTTs to be conducted. Moreover, the performance of etcd is significantly slower than the existing in-network implementations. These results compile a good use case for in-network KV offloading and the integration of programmable devices in Kubernetes' design.

III. Proposed Design

Our design can be broken down in two main domains: the PDP components that utilise P4 to enable in-network replication (NetCRAQ); and the Kubernetes framework that was extended to support offloading KV pairs to PDP based on real-time metrics. An overview of the design is shown in Figure 1, components in red depict our additions, green components represent etcd, and blue/gray are used for Kubernetes.

A. Kubernetes control plane extension

Kubernetes' control plane is comprised of 5 components: the API server which is the front end for the control plane; the controller manager that monitors jobs, endpoints, and distributes tokens; etcd as aforementioned; the Control Network Interface (CNI) that establishes network routing among the participating nodes; and the scheduler which allocates pods to workers by comparing the requested resources with the available worker resources [21]. In our proposed design, two Kubernetes components have been extended to support the integration of PDP: etcd and CNI.

A monitoring component has been added to etcd in order to identify most commonly accessed KV pairs. It uses the integrated Prometheus endpoint to read metrics [22]. The most frequent KV pairs are selected as candidates for deployment in PDP. The number of pairs is decided based on the available memory of the network device and the total size of the candidate KV pairs.

Monitoring is also in place for the values already existing in PDP, which have counters for access frequency. These statistics are obtained through P4Runtime which is executed as part of the CNI [23]. Through the NetCRAQ placement scheduler, also located within the CNI, these metrics are compared and a decision on which values will be transferred to PDP is made. The most frequently read values are placed in data plane. The pairs are transferred through packets that are generated in the CNI.

The P4Runtime CNI component is also used for less time-critical operations related to data plane. Forwarding rules are generated and installed through the control plane upon initialisation or failure. Equally, the roles of the switches are initialised through the control plane. They are installed across all switches to make sure role-based forwarding does not involve retrieving data from the controller, which would introduce delays in the packet processing pipeline. The CNI is entirely responsible for reacting to failures. In the case of a failure, it initially activates the failover mechanism, which redirects traffic from the failed node to a working switch. This is done to minimise traffic loss while the node is down. Once a recovery node is booted, the CNI installs the latest KV pairs in its registers and then replaces the failed node with the recovery one.

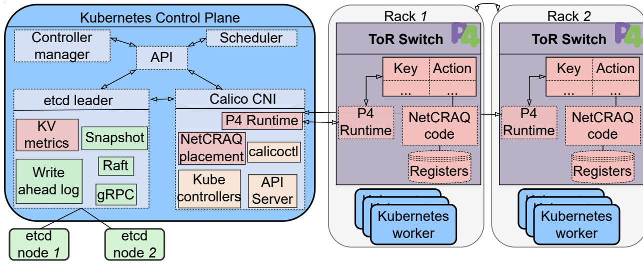

Fig. 1. Overview of proposed design.

B. NetCRAQ overview

We leverage the line-rate performance of PDP devices to offload all the query processing tasks in the data plane. This ensures that fast responses are generated even when retrieving values from other switches cannot be avoided. The time necessary to retrieve a local key-value pair is minimised by placing the necessary data structures within each switch's registers. The registers are located within the ASIC and ensure line-rate accesses [2]. Moreover, we keep all coordination messages within the data plane to ensure minimum delay and consistency. The NetCRAQ components are running in Top of Rack switches directly above Kubernetes worker nodes.

IV. NetCRAQ Data Plane Design

NetCRAQ's data plane design is responsible for two main operations: storing/retrieving the KV pairs in the relevant data structures and processing and forwarding queries and coordination messages. To adhere to the constraints of PDP devices and program under a solid set of rules that will make our implementation transferable to programmable switch chips, we choose the P4 programming language [1] and the Portable Switch Architecture (PSA) [2]. P4 is specifically designed to be compatible with programmable switches and ensures that the compiled binary can be executed with near line-rate performance. This choice also allows for a direct comparison against NetChain, which is developed with the same tools.

A. KV data structures

A key difference between CR and CRAQ is the way that new writes are processed. In CRAQ, for each object k, there are potentially multiple versions, n. In CR, appending multiple values for an object is not required and instead the only "clean" version exists in the tail. In the context of programmable switches, to satisfy CRAQ's requirement, n register cells need to be available to commit writes. For this reason, the switch is initialised with $k \times n$ sequential register cells reserved, forming an array. We call this the `objects_store` array.

The state of each object (clean/dirty) has to be retrieved to determine the future control logic operations. We implicitly define the state as clean iff the latest committed value exists in the first cell of the object's space within the array. This implicit definition is based on the principle that every previous value is deleted upon a successful write of an object, i.e., when this is acknowledged by the tail node. We store the location of the latest committed value for each object in the `read_index` array. Similarly, the location of the next available cell to commit a write is stored in the `write_index` array. The latter is also used to prevent out-of-bound writes.

B. Packet format

In Section II-B, NetChain's packet structure is described. We reiterate its variability according to the chain length and the large amount of overhead bytes that can be added. Since extensive packet parsing cannot be avoided when messages have to traverse the entire chain, even by employing CRAQ, truncating the packet size and reducing packet modifications on each switch should enable faster forwarding between the participating nodes. NetCRAQ's packet format follows a simpler approach by having just three fields layered over UDP:

- `KV_OP`: defines the type of the operation: read request/reply, write request, acknowledgement. (2 bit)
- `KEY_ID`: contains the key id. (32 bit)
- `VALUE`: the field containing the value for the specific key. (128 bit)

The number of participating nodes and the IPs thereof are omitted from the packet and instead placed within the switch, thus reducing the parsing time for all KV operations and coordination messages. The control plane is responsible for updating the roles according to changes, instead of relying on the incoming packets for information that concerns the network infrastructure. This way, when changes occur within the network, recalculation of forwarding rules and switch roles can happen in the control plane and switches can be updated accordingly.

C. Control logic

The control logic, executed by all participating switches, entails all the necessary operations for interacting with the KV pairs and managing the network traffic. All operations of the KVS are atomic to protect the values from simultaneous accesses. NetCRAQ's control logic relies heavily on the match-action concept, while values obtained from parsing the NetCRAQ header are matched against a pre-defined table that dictates the action that is executed when a match occurs. These match-action pairs are computed at the control plane.

Metadata fields, used for branching decisions, are also filled by the control plane in advance using the same mechanism. These metadata fields contain values that need to be regularly retrieved to manage incoming traffic. For example, the role of each switch or the IP of the switch appointed to be the tail of the chain. This constitutes a key design difference with NetChain, since the aforementioned information is already stored and maintained in the switches instead of being passed through incoming packets (in the case of NetChain). Having this information stored instead of parsed, should enable faster overall parsing [19].

The control of packets that contain the NetCRAQ header is primarily dictated by the `KV_OP` field. The allowed operations for this field are: `READ`, `READ_REPLY`, `WRITE`, and

Algorithm 1: Control Logic

```
1  objects_store = register[k * n];
2  read_index = register[n];
3  write_index = register[n];
4  if kv_op == READ then
5  |   get_read_index(KEY_ID);
6  |   get_my_role();
7  |   if meta.read_index == 0 then
8  |   |   clean_read(KEY_ID);
9  |   |   generate_reply();
10 |   else if meta.my_role == TAIL then
11 |   |   dirty_read(KEY_ID);
12 |   |   generate_reply();
13 |   else
14 |   |   forward_to_tail();
15 else if kv_op == WRITE then
16 |   get_write_index(KEY_ID);
17 |   get_my_role();
18 |   if meta.write_index == 0 then
19 |   |   clean_write(KEY_ID);
20 |   |   forward_to_tail();
21 |   else
22 |   |   if meta.write_index >= NUMBER_OF_VERSIONS then
23 |   |   |   drop();
24 |   |   else
25 |   |   |   dirty_write(KEY_ID);
26 |   |   |   forward_to_tail();
27 |   if meta.my_role == TAIL then
28 |   |   clean_write(KEY_ID);
29 |   |   generate_acknowledgement();
30 |   |   multicast();
31 else if kv_op == ACKNOWLEDGEMENT then
32 |   clean_write(KEY_ID);
```

ACKNOWLEDGE. Deletes happen in the form of a WRITE operation, since the memory is statically managed and cannot be freed upon removal of KV pairs.

Algorithm 1 shows the complete control logic. If the identified operation is a READ, the next decision is based on the position of the value within the register. If a value exists in the first position of the object's register space, we know that the version is clean, otherwise it is dirty. The next stage includes checking the role of the node. Only a tail node can reply to a read with a dirty version, while the rest can only reply with clean versions of an object. Writes in the tail node may be committed but not yet acknowledged by all nodes, therefore replying with the latest value is not voiding consistency.

For WRITE operations, we use a similar technique to identify clean writes: the index for the next available cell to commit a write should be at the first cell of the object's register space. Otherwise, the write should be considered dirty and, once committed, should be forwarded to the tail. If the write attempts to exceed the object's predefined register space, it is considered to be out of bounds and the packet is dropped. All write requests that arrive at the tail are considered to be the latest clean ones and acknowledgements are generated for the rest of the chain. To quickly and efficiently update the rest of the chain, we use the multicast functionality of P4, which automatically generates the correct amount of necessary packets, based on the size of the chain and transmits them at once. This removes the need to sequentially retrieve the IPs and generate packets for the rest of the nodes, resulting in faster updates for the chain.

Lastly, receiving an ACKNOWLEDGEMENT message in any node implies that the contained value of the message is the latest clean version for the object. Therefore, all previous versions are deleted and the relevant indices for the object are reset.

V. Handling Failures

Failure mitigation happens in two phases: 1) immediate redirection of traffic to a failover node to reduce the traffic loss; and 2) complete recovery with a replacement node and re-installation of forwarding rules and KV pairs.

When a node remains unresponsive for a certain amount of time – 50ms in our case, the client can automatically direct requests to a different chain node. This time can be adjusted based on what is considered as a prolonged lack of response according to the average response rate of the network. Once the failure is noticed by the control plane, the forwarding rules are updated by removing the node from the forwarding tables and the multicast group. During this phase, the total capacity of the chain is expected to decrease to the maximum capacity that can be offered by the working nodes.

In the second phase, a new node re-enters the chain. To maintain consistency, we follow CRAQ's approach to identify which node will be used to copy KV pairs from. The control plane, depending on the position of the failed node, decides the node that will be used to copy the KV pairs to the new node. Due to space limitations, we refer the reader to the original CRAQ paper for the complete list of scenarios [20].

The recovery node remains offline while the control plane copies the KV pairs from an online node. During this phase, the control plane also disables any writes across the chain in order to preserve consistency. When the copy is complete, the node is added in the forwarding tables and the multicast group of the chain. The total capacity of the chain is completely restored to the initial one.

VI. Evaluation

Taking into account the workloads that Kubernetes generates towards etcd, we evaluate our proposed data plane method. We compare against NetChain, the current state-of-the-art in-network replication method. Our testbed runs a bare-metal installation of Ubuntu 18.04 (kernel: 4.15.0-140-generic) on Intel Core i7-4790 CPU and 16GB of DDR3 RAM. P4 behaviour is emulated using the reference BMv2 switch [24] - compiled using performance flags. The topology is generated and managed using Mininet [25] and P4-utils [26]. The control plane communicates with the Mininet switches using Thrift [27] and the P4-utils API.

A. Throughput

We evaluate the throughput of both platforms based on the maximum attainable rate at which they can provide responses to queries. The measurements are in Queries Per Second (QPS). We direct millions of packets to each switch while increasing the packet rate. The maximum attainable response

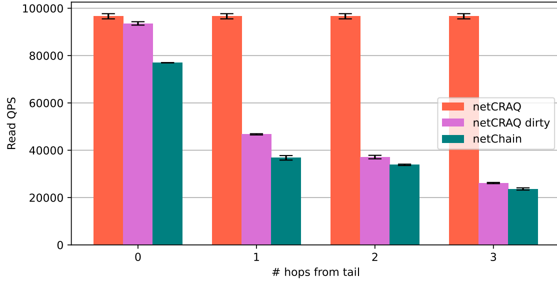

Fig. 2. Max read QPS vs distance from tail.

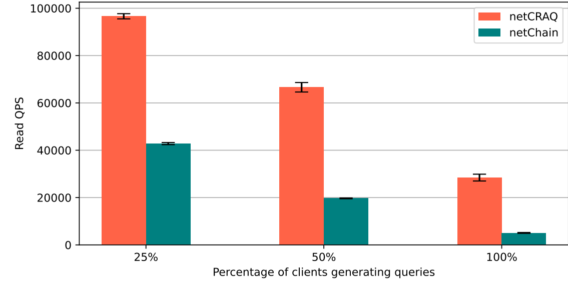

Fig. 3. Sustained read throughput vs percentage of congestion.

rate is considered the rate at which the response rate starts to decrease and the response latency rises.

In figure 2, we test the impact of NetCRAQ's ability to provide responses to read queries of a clean version. NetChain has to direct queries to the tail node in order to respond, regardless of the object's state. This difference is expected to generate traffic that will congest the link of the tail node, causing decreased response throughput and increased latency. This effect is displayed by using a 4-node chain and individually generating queries to each participating node. We monitor the throughput each node is able to achieve given the distance it has from the tail.

Figure 2 shows that NetCRAQ's throughput is not impacted by distance when the queried object is clean. The reduction in required hops and computations create a big performance difference in favour of NetCRAQ: $4.08\times$ higher throughput for queries directed to the head of the chain. In case of dirty objects, throughput is still higher than NetChain with the difference being attributed to the smaller packet size used by NetCRAQ, 72 overhead bytes for NetChain (`SEQ` field set to 128 bits to allow continuous traffic) vs 22 bytes for NetCRAQ. This difference results in smaller parsing times, which when the number of hops increases is less apparent.

When dirty queries are generated directly at the tail, the amount of processing required to generate a reply is the only factor impacting performance since hops are minimum. In that case NetCRAQ shows 22% higher throughput than NetChain, proving higher overall computation efficiency. When queries are directed to the head, the amount of nodes between the source of the queries and the tail is introducing limitations in throughput. Here NetCRAQ is 10.5% faster. Overall, in terms of throughput, regardless of the state of the object and the distance from tail, NetCRAQ shows superior performance.

We examine how the two platforms operate in an environment with limited resources in order to determine the impact of traffic gain and the overall computation efficiency. A platform with minimum redundant operations will be able to utilise the available resources to generate replies instead of performing chain traversals and packet parsing. This is indicative of the wasted processing cycles and link strain that would occur in a data centre environment. To evaluate these properties, we create congestion in an increasing number of switches across the chain and assess the impact this has in throughput. To make the comparison fair, all different combinations of clients are averaged, regardless of their distance from the tail. Figure 3 shows the outcome of this experiment. Once more, NetCRAQ achieves better utilisation of the testbed resources and sustains higher throughput under intense workload scenarios: $2.25\times$ higher throughput for 25% congestion in the chain, $2.8\times$ higher throughput for 50% congestion, and $4.73\times$ higher throughput for 100% congestion.

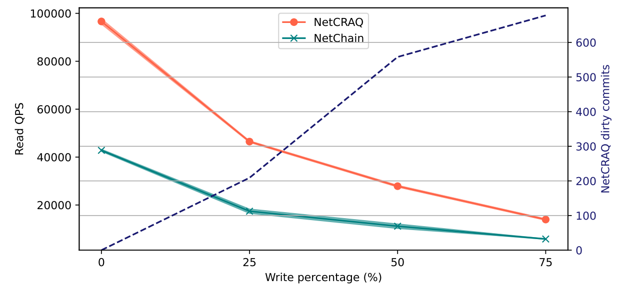

Fig. 4. Performance under mixed read/write workloads.

B. Mixed workloads

We evaluate the platforms under realistic workloads containing a mix of reads and writes. The behaviour of both platforms under such workloads is shown in Figure 4. Starting from a read-only workload, we gradually increase the percentage of writes with a step of 25%. The performance of the platforms is judged by their attainable response rate. NetCRAQ achieves more than double the read throughput for all write percentages. With 75% of the queries being writes, both platforms show a decrease of around 85% of their read-only workload performance. Nonetheless, the read efficiency of NetCRAQ enables higher throughput. Adequate register cells need to be budgeted to maintain all dirty versions before they can be acknowledged by the tail. This is depicted by the increasing amount of dirty commits as write percentage rises, observed in the right y axis of Figure 4.

C. Scalability

NetCRAQ is also able to operate over longer chains with smaller throughput and latency losses over NetChain. We

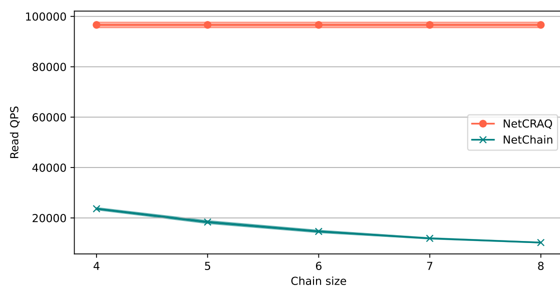

Fig. 5. Read throughput vs chain length.

validate this in Figure 5. Here, the comparison is between read queries directed to the head of the chain. We vary the chain length from 4 to 8 nodes. The throughput difference can be up to 9.46× in favour of NetCRAQ, for the case of 8 chain nodes. Each node is able to respond directly and therefore can accommodate queries at its maximum capacity without forwarding traffic to other nodes, a key aspect that increased the average throughput of the platform.

VII. Conclusion

We investigate the workload characteristics of Kubernetes in etcd. By analysing the obtained metrics we assert that in-network deployment of KV pairs can benefit the performance of Kubernetes' scaling and deployment time. We propose a new design to deploy KV pairs in-network by extending Kubernetes' architecture. We study and evaluate the performance limitations of previous state-of-the-art in-network coordination platforms. We use a new, faster, in-network coordination platform – NetCRAQ. Without harming strong consistency, we effectively increase mean throughput, reduce mean latency (up to multiple orders of magnitude), and improve scalability. By integrating in-network compute in Kubernetes we extend the set of supported devices and suggest better end-to-end programmability.

References

[1] 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," *SIGCOMM Comput. Commun. Rev.*, vol. 44, p. 87–95, July 2014.

[2] "Portable switch architecture." https://p4lang.github.io/p4-spec/docs/PSA-v1.1.0.html. Accessed: 2022-01-29.

[3] "Intel Tofino." https://barefootnetworks.com/products/brief-tofino/. Accessed: 2022-01-29.

[4] "Broadcom Trident4." https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series. Accessed: 2022-01-29.

[5] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 279–291, 2019.

[6] D. R. K. Ports and J. Nelson, "When should the network be the computer?," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, (New York, NY, USA), p. 209–215, Association for Computing Machinery, 2019.

[7] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, (New York, NY, USA), p. 25–33, Association for Computing Machinery, 2019.

[8] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, (New York, NY, USA), p. 150–156, Association for Computing Machinery, 2017.

[9] "etcd: A distributed, reliable key-value store for the most critical data of a distributed system.." https://etcd.io/. Accessed: 2022-01-29.

[10] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, p. 181–192, Nov. 2013.

[11] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, "F1: A distributed sql database that scales," in *VLDB*, 2013.

[12] A. Jeffery, H. Howard, and R. Mortier, "Rearchitecting kubernetes for the edge," in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '21, (New York, NY, USA), p. 7–12, Association for Computing Machinery, 2021.

[13] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, (Renton, WA), pp. 35–49, USENIX Association, Apr. 2018.

[14] H. Takruri, I. Kettaneh, A. Alquraan, and S. Al-Kiswany, "FLAIR: Accelerating reads with consistency-aware network routing," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, (Santa Clara, CA), pp. 723–737, USENIX Association, Feb. 2020.

[15] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin, "Harmonia: Near-linear scalability for replicated storage with in-network conflict detection," *Proc. VLDB Endow.*, vol. 13, p. 376–389, Nov. 2019.

[16] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, p. 133–169, May 1998.

[17] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004* (E. A. Brewer and P. Chen, eds.), pp. 91–104, USENIX Association, 2004.

[18] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," vol. 45, p. 795–809, Apr. 2017.

[19] H. Harkous, M. Jarschel, M. He, R. Pries, and W. Kellerer, "P8: P4 with predictable packet processing performance," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2020.

[20] J. Terrace and M. J. Freedman, "Object storage on CRAQ: High-throughput chain replication for read-mostly workloads," in *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, (San Diego, CA), USENIX Association, June 2009.

[21] "Kubernetes components." https://kubernetes.io/docs/concepts/overview/components/. Accessed: 2022-05-30.

[22] "Prometheus." https://prometheus.io/. Accessed: 2022-05-30.

[23] "P4runtime specification." https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html. Accessed: 2022-05-30.

[24] "Bmv2." https://github.com/p4lang/behavioral-model. Accessed: 2022-01-29.

[25] "Mininet." http://mininet.org/. Accessed: 2022-01-29.

[26] "P4-utils." https://github.com/nsg-ethz/p4-utils. Accessed: 2022-01-29.

[27] "Thrift api." https://thrift.apache.org/docs/. Accessed: 2022-01-29.