TurboKV: Scaling Up the Performance of Distributed Key-value Stores with In-Switch Coordination

Hebatalla Eldakiky, David Hung-Chang Du, Eman Ramadan
Department of Computer Science and Engineering
University of Minnesota - Twin Cities, USA
Emails: {eldak002, du}@umn.edu, eman@cs.umn.edu

Abstract

The power and flexibility of software-defined networks lead to a programmable network infrastructure in which in-network computation can help accelerating the performance of applications. This can be achieved by offloading some computational tasks to the network. However, what kind of computational tasks should be delegated to the network to accelerate applications performance? In this paper, we propose a way to exploit the usage of programmable switches to scale up the performance of distributed key-value stores. Moreover, as a proof-of-concept, we propose *TurboKV*, an efficient distributed key-value store architecture that utilizes programmable switches as: 1) partition management nodes to store the key-value store partitions and replicas information; and 2) monitoring stations to measure the load of storage nodes, this monitoring information is used to balance the load among storage nodes. We also propose a key-based routing protocol to route the search queries of clients based on the requested keys to targeted storage nodes. Our experimental results of an initial prototype show that our proposed architecture improves the throughput and reduces the latency of distributed key-value stores when compared to the existing architectures.

1 Introduction

Programmable switches in software-defined network promise flexibility and high throughput [4, 37]. Recently, the Programming Protocol-Independent Packet Processor (P4) [6] unleashes capabilities that give the freedom to create intelligent network nodes performing various functions. Thus, applications can boost their performance by offloading part of their computational tasks to these programmable switches to be executed in the network. Nowadays, programmable networks get a bigger foot in the data center doors. Google cloud started to use the programmable switches with P4Runtime [30] to build and control their smart networks [13]. Some Internet Service Providers (ISPs), such as AT&T, have already integrated programmable switches in their networks [2]. These switches can be controlled by network operators to adapt to the current network state and application requirements.

Recently, there has been an uptake in leveraging programmable switches to improve distributed systems, e.g., NetPaxos [8, 9], NetCache [17], NetChain [16], DistCashe [24] and iSwitch [23]. This uptake is due to the massive evolution on the capailities of these network switches, e.g., Tofino ASIC from Barefoot [26] which provides sub-microsecond per-packet processing delay with bandwidth up to 6.5 Tbs and throughput of few billions of packets processed per second and Tofino2 with bandwidth up to 12.8 Tbs. These systems use the switches to provide orders of magnitude higher throughput than the traditional server-based solutions.

On the other hand, through the massive use of mobile devices, data clouds, and the rise of Internet of Things [3], enormous amount of data has been generated and analyzed for the benefit of society at a large scale. This data can be text, image, audio, video, etc., and is generated by different sources with un-unified structures. Hence, this data is often maintained in key-value storage, which is widely used due to its efficiency in handling data in key-value format, and flexibility to scale out without significant database redesign. Examples of popular key-value stores include Amazon's Dynamo [10], Redis [33], RAMCloud [29], LevelDB [20] and RocksDB [34].

Such huge amount of data can not be stored in a single storage server. Thus, this data has to be partitioned across different storage instances inside the data center. The data paritions and their storage node mapping (directory information) are either stored on a single coordinator node, e.g., the master coordinator in distributed Google file system [14], or replicated over all storage instances [10, 19, 33]. In the first approach, the master coordinator represents a single point of failure, and introduces a bottleneck in the path between clients and storage nodes; as all queries are directed to it to know the data location. Moreover, the query response time increases, and hence, the storage system performance decreases. In the second approach, where the directory information is replicated on all storage instances, there are two strategies that a client can use to select a node where the request will be sent to: *server-driven coordination* and *client-driven coordination*.

In server-driven coordination, the client routes its request

through a generic load balancer that will select a node based on load information. The selected node acts like the coordinator for the client request. It answers the query if it has the data partition or forwards the query to the right instance where the data partition resides. In this strategy, the client neither has knowledge about the storage nodes nor needs to link any code specific to the key-value storage it contacts. Unfortunately, this strategy has a higher latency because it introduces additional forwarding step when the request coordinator is different from the node holding the target data.

In client-driven coordination, the client uses a partition-aware client library that routes requests directly to the appropriate storage node that holds the data. This approach achieves a lower latency compared to the server-driven coordination as shown in [10]. In [10], the client-driven coordination approach reduces the latencies by more than 50% for both 99.9^{th} percentile and average cases. This latency improvement is because the client-driven coordination eliminates the overhead of the load balancer and skips a potential forwarding step introduced in the server-driven coordination when a request is assigned to a random node. However, it introduces additional load on the client to periodically pickup a random node from the key-value store cluster to download the updated directory information to perform the coordination locally on its side. It also requires the client to equip its application with some code specific to the key-value store used.

Another challenge in maintaining distributed key-value stores is how to handle dynamic workloads and cope with changes in data popularity [1, 7]. Hot data receives more queries than cold data, which leads to load imbalance between the storage nodes; some nodes are heavily congested while others become under-utilized. This results in a performance degradation of the whole system and a high tail latency.

In this paper, we propose *TurboKV* : a novel Distributed Key-Value Store Architecture that leverages the power and flexibility of the new generation of programmable switches. *TurboKV* scales up the performance of the distributed key-value storage by offloading the partitions management and query routing to be carried out in network switches. *TurboKV* uses a *switch-driven coordination* which utilizes the programmable switches as: 1) partition management nodes to store and manage the directory information of key-value store; and 2) monitoring stations to measure the load of storage nodes, where this monitoring information is used to balance the load among storage nodes.

TurboKV adapts a hierarchical indexing scheme to distribute the directory information records inside the data center network switches. It uses a key-based routing protocol to map the requested key in the query packet from the client to its target storage node by injecting some information about the requested data in packet headers. The programmable switches use this information to decide where to send the packet to reach the target storage node directly based on the directory information records stored in the switches' data plane. This in-switch coordination approach removes the load of routing the requests from the client side in the client-driven coordination without introducing an additional forwarding step introduced by the coordination node in the server-driven coordination.

To achieve both reliability and high availability, *TurboKV* replicates key-value pair partitions on different storage nodes. For each data partition, *TurboKV* maintains a list of nodes that are responsible for storing the data of this partition. *TurboKV* uses the chain replication model to guarantee strong data consistency between all partition replicas. In case of having a failing node, requests will be served with other available nodes in the partition replica list.

TurboKV also handles load balancing by adapting a dynamic allocation scheme that utilizes the architecture of software-defined network [12, 25]. In our architecture, a logically centralized controller, which has a global view of the whole system [15], makes decisions to migrate/replicate some of the popular data items to other under-utilized storage nodes using monitoring reports from the programmable switches. Then, it updates the switches' data plane with the new indexing records. To the best of our knowledge, this is the first work to use the programmable switches as the request coordinator to manage the distributed key-value store's partition information along with the key-based routing protocol. Overall, our contributions in this paper are four-fold:

- We propose the in-switch coordination paradigm, and design an indexing scheme to manage the directory information records inside the programmable switch along with protocols and algorithms to ensure the strong consistency of data among replica, and achieve the reliability and availability in case of having nodes failure.
- We introduce a data migration mechanism to provide load balancing between the storage nodes based on the query statistics collected from the network switches.
- We propose a hierarchical indexing scheme based on our proposed rack scale switch coordinator design to scale up *TurboKV* to multiple racks inside the existing data center network architecture.
- We implemented a prototype of *TurboKV* using P4 on top of the simple software switch architecture BMV2 [5]. Our experimental results show that our proposed architecture improves the throughput and reduces the latency for the distributed key-value stores.

The remaining sections of this paper are organized as follows. Background and motivation is discussed in Section 2. Section 3 provides an overview of the *TurboKV* architecture, while the detailed design of *TurboKV* is presented in Section 4 and Section 5. Section 6 discusses how to scale up *TurboKV* inside the data center networks. *TurboKV* implementation is discussed in Section 7, while Section 8 gives an experimental evidence and analysis of *TurboKV*. Section 9 provides a short survey about the related work to us, and finally, the paper is concluded in Section 10.

2 Background and Motivation

2.1 Preliminaries on Programmable Switches

Software-Defined Network (SDN) simplifies network devices by introducing a logically centralized controller (control plane) to manage simple programmable switches (data plane). SDN controllers set up forwarding rules at the programmable switches and collect their statistics using OpenFlow APIs [25]. As a result, SDN enables efficient and fine-grained network management and monitoring in addition to allowing independent evolution of the controller and programmable switches.

Recently, P4 [6] has been introduced to enrich the capabilities of network devices by allowing developers to define their own packet formats and build the processing graphs of these customized packets. P4 is a programming language designed to program parsing and processing of user-defined packets using a set of match/action tables. It is a target-independent language, thus a P4 compiler is required to translate P4 programs into target-dependent switch configurations.

Figure 1(a) represents the five main data plane components for most of modern switch ASICs. These components include programmable parser, ingress pipeline, traffic manager, egress pipeline and programmable deparser. When a packet is first received by one of the ingress ports, it goes through the programmable parser. The programmable parser, as shown in Figure 1(a), is modeled as a simple deterministic state machine, that consumes packet data and identifies headers that will be recognized by the data plane program. It makes transitions between states typically by looking at specific fields in the previously identified headers. For example, after parsing the Ethernet header in the packet, the next state will be determined based on the Ethertype field, whether it will be reading a VLAN tag, an IPv4 or IPv6 header.

After the packet is processed by the parser and decomposed into different headers, it goes through one of the ingress pipes to enter a set of match-action processing stages as shown in Figure 1(b). In each stage, a key is formed from the extracted data and matched against a table that contains some entries. These entries can be added and removed through the control plane. If there is a match with one of the entries, the corresponding action will be executed, otherwise a default action will be executed. After the action execution, the packet with all the updated headers from this stage enters the next stages in the pipeline. Figure 1(c) illustrates the processing inside a pipeline stage, while Figure 1(d) shows an example IPv4 table that picks an egress port based on destination IP address. The last rule drops all packets that do not match any of the IP prefixes, where this rules corresponds to the default action.

After the packet finishes all the stages in the ingress pipeline, it is queued and switched by the traffic manager for an egress pipe for further processing. At the end, there is a programmable deparser that performs the reverse operation of the parser. It reassembles the packet back with the updated headers so that the packet can go back onto the wire to the

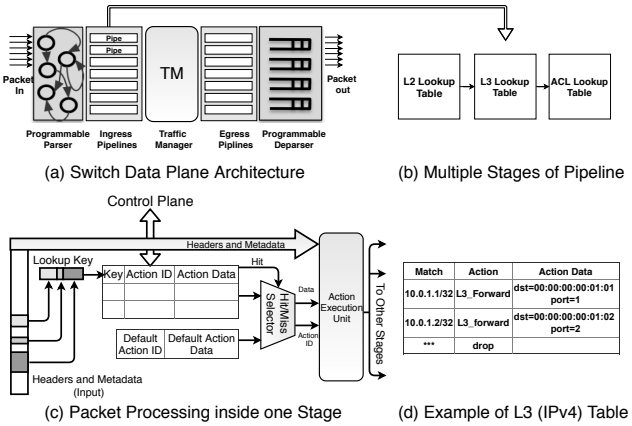


Figure 1: Primelinries on Programmable Switches

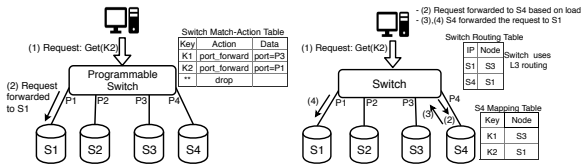
next hop in the path to its destination. Developers of P4 programs should take into consideration some constraints when designing a P4 Program. These constrains include: (i) the number of pipes, (ii) the number of stages and ports in each pipe, and (iii) the amount of memory used for prefix matching and data storage in each stage.

2.2 Why In-Switch Coordination?

We utilize the programmable switches to scale up the performance of distributed key-value stores with the in-switch coordination because of three reasons. First, programmable switches have become the backbone technology used in modern data centers or rack-scale clusters that allow developers to define their own functions for packet processing and provide flexibility to program the hardware. For example, Google uses the programmable switches to build and control their data centers networks [13].

Second, the partition mangement and request coordination are communication-bounded rather than being computation-bounded. So, the massive evolution in the capailities of the network switches, which provide orders of magnitude higher throughput than the highly optimized servers, makes them the best option to be used as partition mangement and request cordination nodes. For example, Tofino ASIC from Barefoot [26] provides few billions of packets processed per second with 6.5 Tbps bandwidth. Such performance is orders of magnitude higher than NetBricks [31] which processes millions of packets per second and has 10-100 Gbps bandwidth [16].

Third, client requests already pass through network switches to arrive at their target. So, offloading the partition management and query routing to be carried out in the network switches with the in-switch coordination approach will reduce the latency introduced by the server-based coordination approach; the number of hops that the request will travel from the client to the target storage node will be reduced as shown in Figure 2. In-switch coordination also removes the load from the client in the client-based coordination approach by making the programmable switch manage all the information for the request routing.



(a) In-Switch (2 hops) (b) Server-based (4 hops)

Figure 2: Request Coordination Models

3 *TurboKV* Architecture Overview

TurboKV is a new architecture of the future distributed key-value stores that leverages the capability of programmable switches. *TurboKV* uses an in-switch coordination approach to maintain the partition management information (directory information) of distributed key-value stores, and route clients' search queries based on their requested keys to the target storage nodes. Figure 3 shows the architecture of *TurboKV* which consists of the programmable switches, controller, storage nodes, and system clients.

Programmable Switches. Programmable switches are the essential component in our new proposed architecture. We augment the programmable switches with a key-based routing approach to deliver *TurboKV* query packets to the target key-value storage node. We leverage match-action tables and switch's registers to design the in-switch coordination where the partition management information will be stored on the path from the client to the storage node. This directory information represents the routing information to reach one of the storage nodes (Section 4.1). Following this approach, the programmable switches act as request coordinator nodes that manage the data partitions and route requests to target storage nodes. In addition to using the key-based routing module, other packets are processed and routed using the standard L2/L3 protocols which makes *TurboKV* compatible with other network functions and protocols (Section 4.2). Each programmable switch has a query statistics module to collect information about each partition's popularity to estimate the load of storage nodes (Section 5.1). This is vital to make data migration decisions to balance the load among storage nodes especially to handle dynamic workloads where the key-value pair popularity changes overtime.

Controller. The controller is primarily responsible for system reconfigurations including (a) achieving load balancing between the distributed storage nodes (Section 5.1), (b) handling failures in the storage network (Section 5.2), and (c) updating each switch's match-action tables with the new location of data. The controller receives periodic reports form switches about the popularity of each data partition. Based on these reports, it decides to migrate/replicate part of the popular data to another storage node to achieve load balancing. Through the control plane, the controller updates the match-action tables in the switches with the new data locations. *TurboKV* controller is an application controller that is different from the network controller in SDN, and it does not interfere with other network

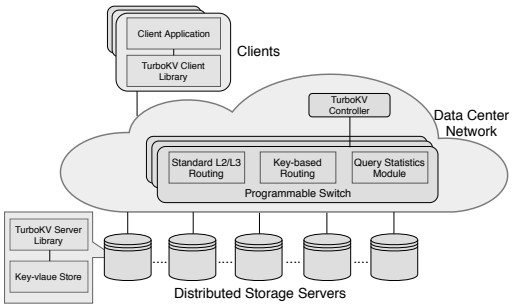


Figure 3: *TurboKV* Architecture

protocols or functions managed by the SDN controller. Our controller only manages the key-range based routing and data migrations and failures associated with them. Both controllers can be co-located on the same server, or on different servers.

Storage Nodes. They represent the location where the key-value pairs reside in the system. The key-value pairs are partitioned among these storage nodes (Section 4.1.1). Each storage node runs a simple shim that is responsible for reforming *TurboKV* query packets to API calls for the key-value store, and handling *TurboKV* controller's data migration requests between the storage nodes. This layer makes it easy to integrate our design with existing key-value stores without any modifications to the storage layer.

System Clients. *TurboKV* provides a client library which can be integrated with the client applications to send *TurboKV* packets through the network, and access the key-value store without any modifications to the application. Like other key-value stores such as LevelDB [20] and RocksDB [34], the library provides an interface for all key-value pair operations that is responsible for constructing the *TurboKV* packets and translates the reply back to the application.

4 *TurboKV* Data Plane Design

The data plane provides in-switch coordination model for the key-value stores. In this model, all partition management information and query routing are managed by the switches. Figure 4 represents the whole pipeline that the packet traverses inside the switch before being forwarded to the right storage node. In this section, we discuss how the switch data plane supports these functions.

4.1 On-Switch Partition Management

4.1.1 Data Partitioning

In large-scale key-value stores, data is partitioned among storage nodes. *TurboKV* supports two different partitioning techniques. Applications can choose any of these two partitioning techniques according to their needs.

Range Partitioning. In range partitioning, the whole key space is partitioned into small disjoint sub-ranges. Each sub-range is assigned to a node (or multiple nodes depending on the replication factor). The data with keys fall into a certain sub-range is stored on the same storage node that is responsible for this sub-range. The key itself is used to map the user

request to one of the storage nodes. Each storage node can be responsible for one or more sub-ranges. Each storage node has LevelDB [20] or any of its enhanced versions installed where keys are stored in lexicographic order on SSTs (Sorted String Tables). This key-value library is responsible for managing the sub-ranges associated to the storage node and handling the key-value pair operations directed to the storage node. The advantage of this partitioning scheme is that range queries on keys can be supported, but unfortunately, this scheme suffers from load imbalance problem discussed in Section 5.1. Some sub-ranges contain popular keys which receive more requests than others. The mapping table for this type of partitioning represents the key range and the associated nodes where the data resides as shown in Figure 5(a). Support of range partitioning in the switch data plane is discussed in Section 4.1.3.

Hash Partitioning. In hash partitioning, each key is hashed into a 20-byte fixed-length digest using RIPEMD160 [11] which is an extremely random hash function, ensures that records are distributed uniformly across the entire set of possible hash values. We developed a variation of the consistent hashing [18] to distribute the data over multiple storage nodes. The whole output range of the hash function is treated as a fixed space. This space is partitioned into sub-ranges, each sub-range represents a consecutive set of hashing values. These sub-ranges are distributed evenly on the storage nodes. Each storage node can be responsible for one or more sub-ranges based on its load, and each sub-range is assigned to one node (or multiple nodes depending on the replication factor). Like the consistent hashing, partitioning one of the sub-ranges or merging two sub-ranges due to the addition or removal of nodes affects only the nodes that hold these sub-ranges and other nodes remain unaffected. Each data item identified by a key is assigned to a storage node if the hashed value of its key falls in the sub-range of hashing values which the storage node is responsible for. This method requires a mapping table, shown in Figure 5(b), where each record represents a hashed sub-range and its associated nodes where data resides. Support of hash partitioning in the switch data plane is discussed in Section 4.1.3. The disadvantage of this technique, like all hashing partitioning techniques, is that range queries can not be supported. On the storage nodes side, data is managed in hash-tables and collisions are handled using separate chaining in the form of binary search tree.

For both partitioning techniques, we assume that there is no fixed space assigned to each sub-range (partition) on the storage node (i.e., the space that each partition consumes on a storage node can grow as long as there is available space on the storage node). Unfortunately, multiple insertions to the same partition may exceed the capacity of the storage node. In this case, The sub-range of this partition will be divided into two smaller sub-ranges. One of these small sub-ranges will be migrated to another storage node with available space. Other storage nodes that have the same divided sub-range

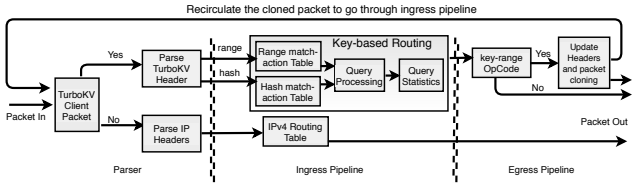


Figure 4: Logical View of *TurboKV* Data Plane Pipeline

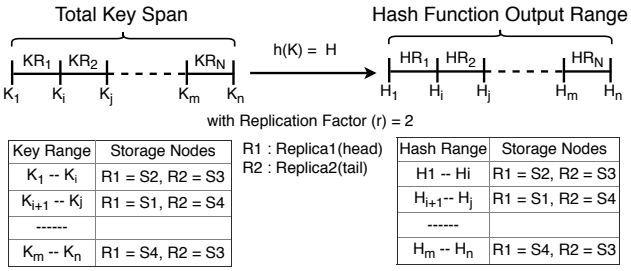


Figure 5: *TurboKV* Data Partitioning and Replication

with available space keep the data of the whole sub-range and manipulate it as it is. The mapping table will be updated with the new changes of the divided sub-ranges.

4.1.2 Data Replication

To achieve high availability and durability, data is replicated on multiple storage nodes. The data of each partition (sub-range) is replicated as one unit on different storage nodes. The replication factor (r) can be adjusted according to application needs. The list of nodes that is responsible for storing a particular partition is called the *replica list*. This *replica list* is stored in the mapping table as shown in Figure 5. *TurboKV* follows the chain replication (CR) model [38] that exhibits high throughput and availability without sacrificing strong consistency guarantees among all partition's replicas. In CR, the storage nodes, holding replicas of data, are organized in a sequential chain structure. Read queries are handled by the tail of the chain, while write queries are sent to the head of the chain, the head processes the request and forwards it to its successor in the chain structure. This process continues till reaching the tail of the chain. Finally, the tail processes the request and replies back to the client. With r replicas, CR can sustain up to (r-1) node failures as requests will be served with other replicas on the chain structure.

4.1.3 Management Support in Switch Data Plane

In *TurboKV*, the switch data plane has three types of match-action tables: range partition management table, hash partition management table and the normal IPv4 routing table. We will discuss the design of the partition management tables which are related to the key-based routing protocol for the rack scale cluster shown in Figure 6(a). The design of both tables is the same but the value we used for matching and the value we match against are different based on the type of partitioning. We will refer to the value we use for matching as the *matching value*, this value represents the key itself in case of range

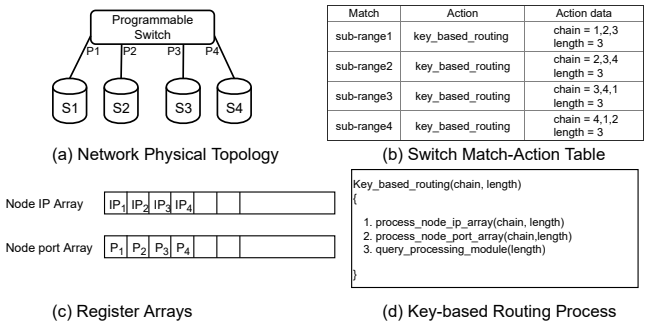


Figure 6: *TurboKV* Partition Management inside Switch

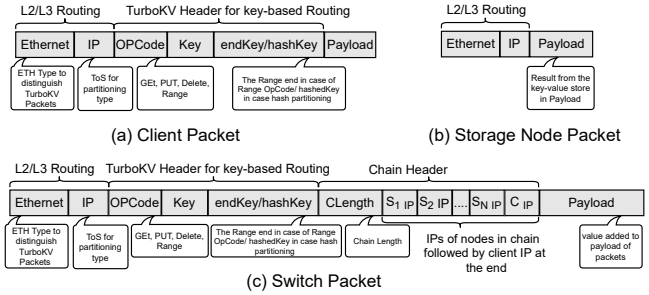


Figure 7: *TurboKV* Packet Format

partitioning and the hashed value of the key in case of hash partitioning. The partition management match-action table design is shown in Figure 6(b). Each record in the table consists of three parts: match, action and action data. The match represents the value that we match the *matching value* against, we refer to it as a *sub-range*. This *sub-range* represents the start and end keys of a sub-range from the whole key span in range partitioning, or the start and end hash values of a consecutive set of hash values in hash partitioning. The action represents the key-based routing that will be executed when a *matching value* falls within the *sub-range*. The action data consists of two parts: chain and length. Chain represents the forwarding information for nodes forming the chain of the sub-range. This information includes node's IP address and the port from the switch to the storage node. Nodes' information is sorted according to node's position in the chain structure (i.e., first node is the head and last node is the tail), and is used in updating packet during the action execution.

In *TurboKV*, each node holds the data of one or more sub-ranges. This makes each node's forwarding information appears more than once in the match-action table records. *TurboKV* uses two arrays of registers in the switch's data plane to save the forwarding information: node IP array, and node port array. For each storage node, the forwarding information is stored at the same index in the two arrays as shown in Figure 6(c). For example, the information of storage node *S1* is stored at index 1 in the two arrays and the same for the other storage nodes. The index of the storage nodes in the register arrays is stored as action data in the match-action table records to form the chain as shown in Figure 6(b). The key-based routing uses these indexes to process the register arrays and fetch the forwarding information for the replica nodes. This information is used by the query processing module to forward packets to their next hop.

4.2 Network Protocol Design

Packet Format. Figure 7(a) shows the format of *TurboKV* request packet sent from clients. The programmable switches use the `Ethernet Type` in the Ethernet header to identify *TurboKV* packets, and execute the key-based routing. Other switches in the network do not need to understand the format of *TurboKV* header, and treat all packets as normal IP packets. The `ToS` (Type of Service) in the IP header is used to distinguish between three types of *TurboKV* packets: range partitioned data packet, hash partitioned data packet, and *TurboKV* packet previously processed by the switch. The *TurboKV* header consists of three main fields: `OpCode`, `Key`, and `endKey/hashedKey`. The `OpCode` is a one-byte field which stands for the code of the key-value operation (*Get, Put, Del, and Range*). `Key` stores the key of a key-value pair. In *Range* operation, `Key` and `endKey/hashedKey` are used to represent the start and end of the key range, respectively. In case of hash partitioning, the `endKey/hashedKey` is set with the hashed value of the key to perform the routing based on it instead of the key itself.

After the client packet is processed by the programmable switch, the switch adds the chain header shown in Figure 7(c). This header is used by the storage nodes for the chain replication model. It includes two fields. The first field is the number of nodes which the packet passes by in the chain including the client IP (`CLength`). The second field has these nodes' IP addresses ordered according to their position in the chain followed by the client IP at the end. The packet format of the reply from storage node to client is a standard IP packet shown in Figure 7(b). The result is added to the packet payload.

Network Routing. *TurboKV* uses a key-based routing protocol to route packets from clients to storage nodes. The client sends the packet to the network. Once it reaches the programmable switch, the switch extracts the `Key` (in case of range partitioning) or the `endkey/hashedKey` (in case of hash partitioning) from *TurboKV* header, and looks up the corresponding key-based match-action table using the value of the extracted field (*matching value*). If there is a hit, the switch fetches the chain nodes' information from the registers based on the specified indexes in the match-action table. Then, the switch processes this information based on the query type (`OpCode`). After that, the switch updates the packet with the target node's information, and forwards it to the next hop on the path to this target node. The switch uses the *range matching* for table lookup, in which, it matches the *matching value* against the sub-range in the corresponding key-based match-action table. If the *matching value* falls within one of the sub-ranges (hit), the key-based routing action is processed using the action data. The programmable switches route the previously processed *TurboKV* packets and the storage node to client packets using the standard L2/L3 protocol without

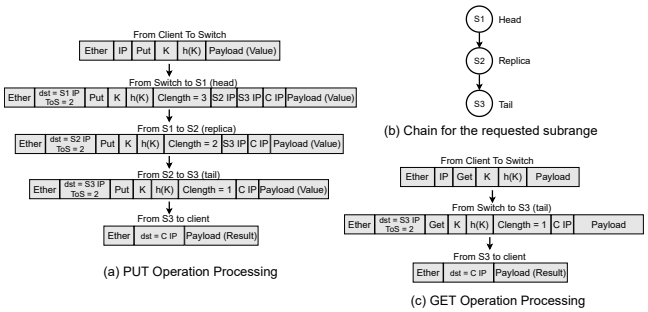


Figure 8: *TurboKV* KV Storage Operations

passing the key-based routing and perform the match-action based on the destination IP in the IP header.

4.3 Key-value Storage Operations

PUT and DELETE Queries. In chain replication, PUT and DELETE queries are processed by each node along the chain starting from the head and replied by the tail. On the switch side, after processing one of the key-based match-action tables with the *matching value* and fetching the corresponding chain information from the register arrays, the query processing module sets the destination IP address in the IP header with the IP address of the chain head node and changes the `ToS` value to mark the packet as previously processed. The egress port is set with the forwarding port of the chain head node, then the chain header is added to the packet with `CLength` equals to the length of the chain and the chain nodes' IP addresses ordered according to their position in the chain followed by the client IP at the end as shown in Figure 8(a). Finally, the packet is forwarded to the head of chain node. As shown in Figure 8(a), when the packet arrives at a storage node, it is processed by *TurboKV* storage library. The node updates its local copy of the data. Then, it reads the chain header, sets the destination address in the IP header with the IP of its successor and reduces the `CLength` by 1, then forwards the packet to its successor. Packets received by the tail node, with `CLength` = 1, have their chain header and *TurboKV* header removed, and the result is sent back using the client IP as the destination address in the IP header.

GET Queries. Following the chain replication, GET queries are handled by the tail of the chain. After performing the matching on the *matching value* and fetching the corresponding chain information from the register arrays, the query processing module sets the destination IP address in the IP header with the IP address of the chain tail node and changes the `ToS` value to mark the packet as previously processed. The egress port is set with the forwarding port of the chain tail node, then the chain header is added to the packet with `CLength` equals to 1 and one node IP which represents the client IP as shown in Figure 8(c). Finally, the packet is forwarded to the storage node. When the packet arrives at the storage node, it is processed by *TurboKV* storage library. The query result is added to the payload of the packet. and the client IP is popped up from the chain header and put in the destination address in the IP header. The chain and *TurboKV* headers are removed from the packet and the result is sent back to the client.

Algorithm 1 Range Query Handling

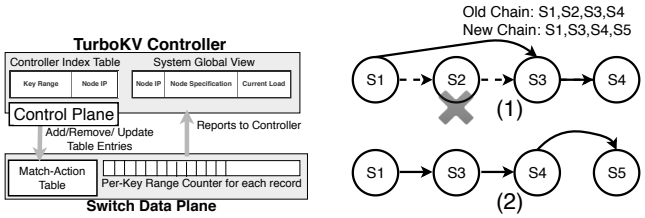
```
1:  Input pkt: packet entering the egress pipeline
2:  matched_subrange: the subrange where the start key of the requested range falls
3:  Output pkt_out: packet to be forwarded to the next hop
4:  Output pkt_cir: packet to be circulated and sent to ingress pipeline as new packet
5:  Begin:
6:  pkt_out = pkt     // clone the packet
7:  if pkt.OpCode == range then
8:      // check if range spans multiple nodes
9:      if pkt.request.endKey > matched_subrange.endKey then
10:         pkt_cir = pkt     // clone the packet
11:         pkt_out.request.endKey = matched_subrange.endKey
12:         pkt_cir.request.Key = Next(matched_subrange.endKey)
13:     end if
14: end if
15: if pkt_cir.exist() then
16:     circulate(pkt_cir)     // send packet to ingress pipeline again
17: end if
18: send_to_output_port(pkt_out)
```

Range Queries. If the data is range partitioned, *TurboKV* can handle range queries. The requested range in the *TurboKV* header may span multiple storage nodes. Thus, the switch divides the range into several sub-ranges, each sub-range corresponds to a packet. Each of these packets contains the start and end keys of the corresponding sub-range. Each packet is handled by the switch like a separate read query and forwarded to the tail node of its partition's chain. Unfortunately, the switch cannot construct new packets from scratch on its own. Therefore, in order to achieve the previous scenario, we placed the range operation check in the egress pipeline as shown in Figure 4. We use the `clone` and `circulate` operations, supported in the architecture of the programmable switches and P4, to solve the packet construction problem. When a range operation is detected in the egress pipeline, the packet is processed as shown in Algorithm 1.

5 *TurboKV* Control Plane Design

5.1 Query Statistics and Load Balancing

In *TurboKV*, the data plane has a query statistics module to provide query statistics reports to the *TurboKV* controller. Thus, the controller can estimate the load of each storage node, and make decisions to migrate part of the popular data to one of the under-utilized storage nodes. As shown in Figure 9(a), the switch's data plane maintains a per-key range counter for each key range in the match-action table. Upon each hit for a key range, its corresponding counter is incremented by one. The controller receives reports periodically from the data plane including these statistics, and resets these counters in the beginning of each time period. Then, the controller compares the received statistics with the specifications of the storage nodes. If a storage node is over-utilized, the controller migrates a subset of the hot data in a sub-range to another storage node and reconfigures the chain of the updated sub-range. Then, the controller updates records in the match-action table of the switches with the new chain configurations. After the sub-range's data is migrated to other storage nodes, the old copy is removed from the over-utilized one.



(a) Logical View of The Controller (b) Chain Reconfiguration

Figure 9: *TurboKV* Control Plane

Currently, the controller follows a greedy selection algorithm to select the least utilized node where data will be migrated.

5.2 Failures Handling

We assume that the controller process is a reliable process and it is different from the SDN controller. We also assume that links between storage nodes and switches in data centers are reliable and are not prone to failures.

Storage Node Failure. When the controller detects a storage node failure, it reconfigures the chains of the sub-ranges on the failed storage node and updates their corresponding records in the key-based match-action table through the control plane. The controller removes the failed storage node from its position in all chains. In each chain, the predecessor of the failed node will be followed by the successor of the failed node, reducing the chain length by 1 as shown in Figure 9(b). If the failed node was the head of the chain, then the new head will be its successor. If the failed node was the tail of the chain, then the new tail will be its predecessor. Reducing the chain length by 1 makes the system able to sustain less number of failures. That is why the controller distributes the data of the failed node in sub-range units among other functional nodes, and adds these new nodes at the end of these sub-ranges' chains in their corresponding records in the key-based match-action table. This process of sub-range redistribution restores the chain to its original length.

6 Scaling Up to Multiple Racks

We have discussed the in-switch coordination for distributed key-value stores within a rack of storage nodes with the Top-of-Rack (ToR) switch as the coordinator. We now discuss how to scale out the in-switch coordination in the data center network. Figure 10 shows the network architecture of data centers. All the servers in the same rack are connected by a ToR switch. In the highest level, there are Aggregate switches (AGG) and Core switches (Core). To scale out distributed key-value stores with in-switch coordination, we develop a "hierarchical indexing" scheme. Each ToR switch has the directory information of all sub-ranges located on its connected storage nodes as described before in Figure 6. In addition to the IPv4 routing table, each AGG switch has two range match-action tables (range and hash), where each table consists of the sub-ranges in its connected ToR switches. The Core switches have the range match-action tables of sub-ranges in its connected AGG switches. With each sub-range in either the AGG

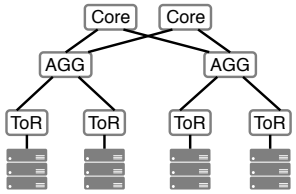


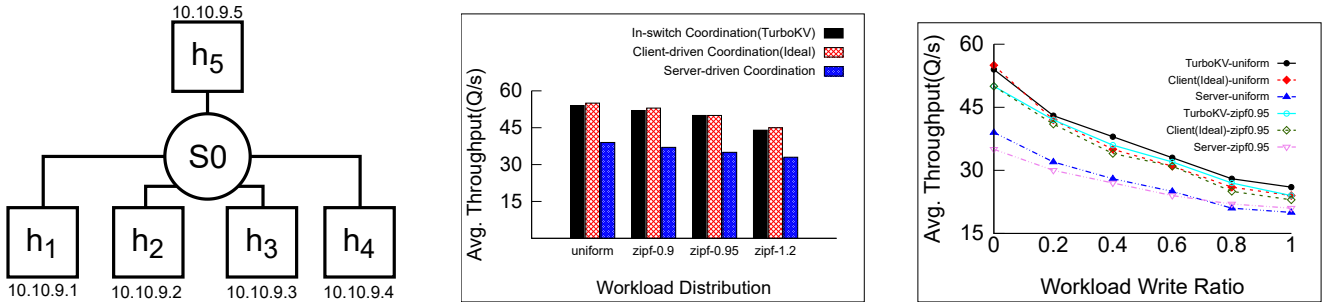
Figure 10: Scaling Up inside Data center Network

or Core switches, the action data represents only the forwarding port towards the head or the tail of the sub-range's chain. No chains are stored in these switches. When a packet is received by an AGG switch or core switch, this packet is processed by the key-based routing protocol without adding any chain header to the packet and forwarded to one of the ports towards the head or tail of the sub-range chain based on the query (write or read respectively). When the packet arrives at ToR switch, the switch processes the packet as discussed in Section 4.3. Replicas of a specific sub-range may be located on different racks. This design leverages the existing data center network architecture and does not introduce additional network topology changes.

7 Implementation Details

We have implemented a prototype of *TurboKV*, including all switch data plane and control plane features, described in Section 4 and Section 5. We have also implemented the client and server libraries that interface with applications and storage nodes, respectively, as described in Section 3. The switch data plane is written in P4 and is compiled to the simple software switch BMV2 [5] running on Mininet [27]. The key size of the key-value pair is 16 bytes with total key range spans from 0 to 2^{128}. This range is divided evenly into 128 records-index table saved on the switch data plane. We used 4 register arrays, one for saving the storage nodes' IP addresses, one for saving the forwarding port of the storage nodes, one for counting the read access requests of the indexing records and the last one for counting the update access requests of the indexing records. The controller is able to update/read the values of these registers through the control plane. It also can add or remove table entries to balance the load of the storage nodes. The switch data plane does not store any key-value pairs as these pairs are saved on the storage nodes. This approach makes *TurboKV* consumes a small amount of the on-chip memory leaving enough space for processing other network operations. The controller is written in Python and can update the switch data plane through the switch driver by using the Thrift API generated by the P4 compiler.

The client and server libraries are written in Python using Scapy [35] for packet manipulation. The client can translate a generated YCSB [7] workload with different distributions and mixed key-value operations into *TurboKV* packets' format and send them through the network. We used Plyvel [32] which is a Python interface for levelDB [20] as the storage agent. The server library translates *TurboKV* packets into Plyvel format and connects to levelDB to perform the key-



(a) Experiment Topology (b) Throughput vs Skewness - Read only (c) Impact of Write Ratio on System Throughput

Figure 11: Experiments Topology and Results

value pair operations. We used chain replication with chain length equals to 3 for data reliability.

8 Performance Evaluation

In this section, we provide the experimental evaluation results of *TurboKV* running on the programmable switches. These results show the performance improvement of *TurboKV* on the key-value operations latency and system throughput.

Experimental Setup. Our experiments consist of one simple software switch BMV2 [5], which connects 5 hosts as described in Figure 11(a). One of the hosts(h_5) represents the client who runs the client library and generates the key-value queries. This client represents the request aggregator server who is the direct client of the storage nodes. Other hosts (h_1, h_2, h_3, and h_4) represent the storage nodes which run the server library and use LevelDB as the storage agent. The whole topology is running on Mininet. The data is distributed over the storage nodes using the range partitioning described in Section 4.1.1 with 128 records index table. Each storage node is responsible for 64 sub-ranges and stores the data whose keys fall in these sub-ranges.

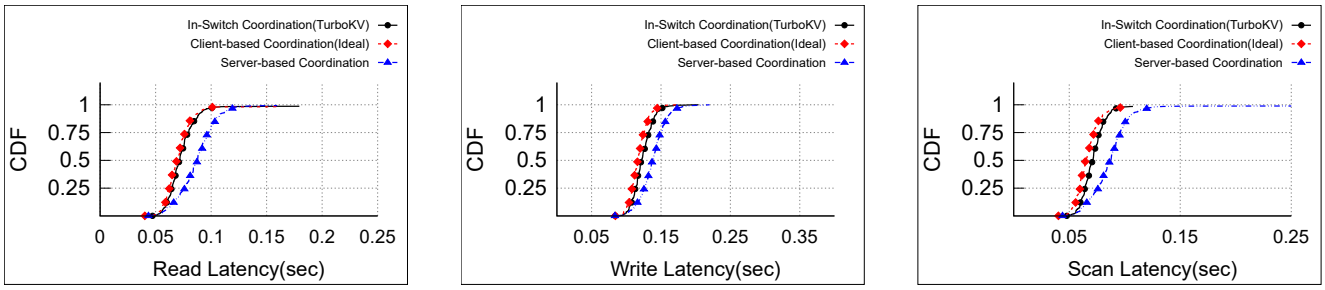
Comparison. We compared our in-switch coordination (*TurboKV*) with server-driven coordination and client-driven coordination described in Section 1. In *TurboKV*, the directory information is stored in the switch data plane and updated by *TurboKV* controller, also the key-based routing is used to route the query from client to target storage node. In server-driven coordination which is implemented on most of existing key-value stores [10, 19], all the storage nodes store the directory information and can act as the request coordinator. In client-driven coordination, the client acts as the request coordinator. The client has to download the directory information periodically from a random storage node, because, with lots of outdated directory information, the client-driven coordination tends to act as the server-driven coordination as the wrong storage node that the client contacts will forward the request again to the target storage node. Both of client-driven and server-driven approaches route the query using the standard L2/L3 routing protocols.

Note that in our experiments, we compare with the ideal case of the client driven coordination where the client has the updated directory information and sends the query directly to the target storage node, ignoring the latency introduced by pulling this information periodically from a random storage node because this latency will depend on the client location and also the load of the storage node that the client contacts to pull this information. The ideal case of the client-driven coordination represents *the least latency* that the client request can achieve because it represents the direct path from the client to the target storage node ignoring any latency resulted from having outdated directory information which may cause extra forwarding steps in the path from the client to the target storage node. With lots of updates to the directory information of the key-value store, the ideal client-driven coordination can not be achieved in real life systems.

Workloads. We use both uniform and skewed workloads to measure the performance of *TurboKV* under different workloads. The skewed workloads follow Zipf distribution with different skewness parameters (0.9, 0.95, 1.2). These workloads are generated using YCSB [7] basic database with 16 byte key size and 128 byte value size. The generated data is stored into records' files and queries' files, then parsed by the client library to convert them into *TurboKV* packet format. We generate different types of workloads: read-only workload, scan-only workload, write-only workload and mixed workload with multiple write ratios. Each workload consists of 1000 records stored into the storage nodes and 1000 key-value queries performed on the stored records.

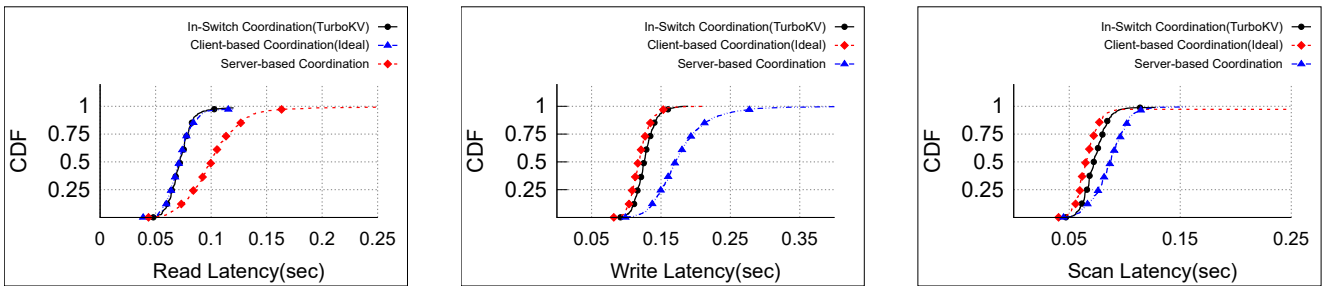
8.1 Effect on System Throughput

Impact of Read-only Workloads. Figure 11(b) shows the system throughput under different skewness parameters with read-only queries. We compare *TurboKV* throughput vs the ideal client-driven throughput and server-driven throughput. As shown in Figure 11(b), *TurboKV* performs nearly the same as the ideal client-driven coordination because the in-switch coordination manages all directory information in the switch data plane and uses the key-based routing to deliver the requests to the storage nodes directly. Moreover, the in-switch coordination eliminates the load of downloading the updated directory information periodically from storage nodes, as this part is managed by the controller who will update the directory information in the switch data plane through the control plane. In addition, the in-switch coordination outperforms the server-driven coordination and improves the system through-



(a) Read Latency (b) Write Latency (c) Scan Latency

Figure 12: Key-value operations Latency for uniform Workload



(a) Read Latency (b) Write Latency (c) Scan Latency

Figure 13: Key-value operations Latency for zipf-1.2 Workload

put with minimum of 33% and maximum of 42%. This result is because the in-switch coordination eliminates the overhead of the load balancer and skips a potential forwarding step introduced in the server-driven coordination when a request is assigned to a random storage node.

Impact of Write Ratio. Figure 11(c) shows the system throughput under uniform and skewed workload with varying the workload write ratio. As shown in Figure 11(c), the throughput decreases as the write ratio increases for the three approaches, because each write query has to update all the copies of the key-value pair following the chain replication approach before returning the reply to the client. *TurboKV* performs roughly the same as the ideal client-driven coordination in the workloads with low write ratio, but *TurboKV* outperforms the client-driven coordination as the write ratio increases. This behavior is because of the chain replication implemented in the system for availability and fault tolerance. In in-switch coordination, the switch inserts all the chain nodes in the packet. When the packet arrives at a storage node, the node updates its local copy and forwards the request to the next storage node directly without any further mapping to know its chain successor. But, in the client-driven coordination, the client sends the write query to the head of chain's node. When the query arrives at the storage node, the node updates its local copy and then accesses its saved directory information to know its chain successor, then forwards the packet to it. So, when the write ratio increases, this scenario is performed for larger portion of queries which affects the system throughput. Also, *TurboKV* outperforms the server-driven coordination by minimum of 30% and maximum of 38% in case of uniform workload, and by minimum of 14% and maximum of 42% in case of the skewed workload. This improvement is because of the elimination of the forwarding step when the request is assigned to a random storage node and also the elimination of further mapping steps on each storage node for knowing the chain successor.

8.2 Effect on Key-value Operations Latency

Figures 12 and 13 show the CDF of all key-value operations latencies under uniform and Zipf-1.2 workloads, respectively, for *TurboKV*, ideal client-driven coordination and server-driven coordination. The analysis of these two figures is shown in Table 1 and Table 2. Figures 12(a), 13(a), 12(b), and 13(b) show that the read and write latencies in *TurboKV* are very close to the ideal client-driven coordination for uniform and skewed workload, because *TurboKV* skips potential forwarding step like the client-driven coordination by managing the directory information in the switch itself. Compared to server-driven coordination, *TurboKV* reduces the read latency by 16.3% on the average and 19.2% for the 99^{th} percentile for the uniform workload, and by 30% on the average and 49% for the 99^{th} percentile for the skewed workload. *TurboKV* also reduces the write latency by 11% on the average and 12.3% for the 99^{th} percentile for the uniform workload, and by 29% on the average and 48% for the 99^{th} percentile for the skewed workload. The reduction in case of skewed workload is better than the reduction of uniform workload, because *TurboKV* does not only skip the excess forwarding step but also removes the load from the storage nodes from being the request coordinator, and makes them focus only on answering the queries themselves, which reduces tail latencies at the storage nodes.

Figures 12(c), and 13(c) shows that *TurboKV* reduces the

	Read (Get) (msec)			Write (Put) (msec)			Scan (Range) (msec)		
	Mean	50th Percentile	99th Percentile	Mean	50th Percentile	99th Percentile	Mean	50th Percentile	99th Percentile
In-Switch Coordination (TurboKV)	72.5	71.2	103.3	123.5	121.8	165.8	71.1	70.1	96.3
Client-driven Coordination	69.8	68.8	98.6	117.5	116.1	153.5	66.1	64.4	96
Server-driven Coordination	86.6	87.7	127.8	138.2	138	189	87	87.2	128.3

Table 1: Request Latency Analysis under Uniform Workload

	Read (Get) (msec)			Write (Put) (msec)			Scan (Range) (msec)		
	Mean	50th Percentile	99th Percentile	Mean	50th Percentile	99th Percentile	Mean	50th Percentile	99th Percentile
In-Switch Coordination (TurboKV)	72.2	71.8	105.3	126.8	125.4	172.4	72.9	72	101.7
Client-driven Coordination	71.4	70.9	104	119.7	117.2	167.3	66.2	64.5	91.7
Server-driven Coordination	102.8	99.8	206.8	178.3	170.9	330.6	86.2	87.3	125.6

Table 2: Request Latency Analysis under Zipf-1.2 Workload

scan latency by 18.3% on average and 24.7% for 99^th percentiles for uniform workloads, and by 15.4% on average and 19% for 99^th percentile for skewed workloads when compared with the server-driven coordination. But, when compared with the ideal client-driven coordination, *TurboKV* increases the latency of the scan operation by range of 7-10% for uniform and skewed workloads, because of the latency introduced inside the switch from packet circulation and cloning to divide the requested range when it spans multiple storage nodes.

9 Related Work

Distributed Key-value Stores. Key-value storage is widely used to support lots of large-scale applications. Some key-value stores, e.g., Redis [33], RAMCloud [29], and memcached [28], manage data in DRAM for faster data access. Other key-value stores, e.g., Dynamo [10], Cassendra [19], LevelDB [20] and RocksDB [34] are presistent key-value stores which save data on presistent storage devices, while other key-value stores, e.g., AeroSpike [36] use hybrid storage(DRAM and SSD). Distributing data over several key-value store instances has been widely studied. Some systems use hash functions to distribute the data among storage nodes. Dynamo [10] and Cassandra [19] use consistent hashing, while Redis [33] and Aerospike [36] use a hash function to distribute data into several hashing slots. Other key-value stores, e.g., LevelDB [20] and RocksDB [34], use alternate approach to distribute the data, they use the key range partitioning where keys are on Sorted String Tables. *TurboKV* supports hash partitioning of the data and range partitioning. Applications can decide the way to partition the data and the corresponding directory information will be stored in switches' data plane.

To achieve durability and high availability, data partitions are replicated over several storage nodes. Dynamo, Cassendra, Aerospike, and Redis replicate the data partitions over several storage nodes and save this information on a mapping table which is replicated also on all storage nodes. *TurboKV* also replicates the data partitions over several storage nodes and follows the chain replication model to guarantee strong consistency. It stores the chain of each partition on the switch data plane. All existing key-value stores use either client-based coordination [10, 36], server-based coordination [10, 19, 33] or a single elected node coordination [29] to deliver requests from clients to storage nodes. *TurboKV* uses in-switch coordination with the key-based routing protocol to route requests from clients to storage nodes directly.

Hardware Acceleration. Lots of work used hardware to speed up the performance of the distributed systems. NetPaxos [8,9] implements Paxos on switches. NetCache [17] implements a rack-scale on-switch cache, while DistCache [24] scales up the NetCache design to multiple racks inside the data center network. NetChain [16] uses the network switches to implement in-network key-value store, but it is bounded by the limited storage in the network switches. There is also SwitchKV [22] which is a cluster scale system to balance the key value stores via caching. SwitchKV uses the OpenFlow switches to save a forwarding rule for each cached key-value pair to route the request to the right caching node. *TurboKV* uses the switches to act as the request coordination nodes that save the partition management information along with the key-based-routing protocol to route the request to the target storage nodes. In *TurboKV*, the actual key-value pairs are saved on storage servers which makes it not limited to applications of small data sizes. Other examples that use the hardware to increase the perfromance include, but not limited to, JoiNS [39] which uses the OpenFlow switches to prioritize I/O packets to meet their latency SLO, KVDirect [21], a high performance KVS that leverages programmable NIC to extend RDMA primitives and enable remote direct key-value access to the main host memory, iSwitch [23] which uses the network switches to improve the performance of the distributed reinforcement learning, and Ibex [40] which supports advanced SQL offloading using FPGA.

10 Conclusion

In this paper, we presented *TurboKV*; a novel distributed key-value store architecture that leverages the power and flexibility of the new programmable switches. *TurboKV* uses the in-switch coordination approach that utilizes the switches as partitions management nodes to store the key-value store partitions locations and replicas information along the path from clients to storage nodes. The programmable switches use key-based routing to route packets from clients to storage nodes. *TurboKV* decreases the query response time and improve system throughput. We believe that *TurboKV* can be deployed on the programmable switches currently integrated in the data center's network to improve the performance of distributed key-value stores.

References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 40(1), June 2012.

[2] ATT usage of Programmable switches: https://www.sdxcentral.com/articles/news/att-picks-barefoot-networks-programmable-switches/2017/04/.

[3] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, pages 2787–2805, 10 2010.

[4] Barefoot Networks: https://www.barefootnetworks.com/press-releases/barefoot-networks-unveils-tofino-2-the-next-generation-of-the-worlds-first-fully-p4-programmable-network-switch-asics/.

[5] P4 Software Switch Website: https://github.com/p4lang/behavioral-model.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44, 2014.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10. ACM, 2010.

[8] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2):18–24, May 2016.

[9] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, NY, USA, 2015. ACM.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07. ACM, 2007.

[11] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. Ripemd-160: A strengthened version of ripemd. In *Proceedings of the Third International Workshop on Fast Software Encryption*, pages 71–82, London, UK, UK, 1996. Springer-Verlag.

[12] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44, 2014.

[13] Google Cloud using P4Runtime to build smart networks: https://cloud.google.com/blog/products/gcp/google-cloud-using-p4runtime-to-build-smart-networks.

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[15] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38, 2008.

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, 2018. USENIX Association.

[17] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017.

[18] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[19] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010.

[20] LevelDB: A light-weight single-purpose library for persistent key-value store, http://leveldb.org/, https://github.com/google/leveldb.

[21] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[22] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 31–44, Berkeley, CA, USA, 2016. USENIX Association.

[23] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 279–291, New York, NY, USA, 2019. ACM.

[24] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, Boston, MA, 2019. USENIX Association.

[25] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38, 2008.

[26] "Barefoot Tofino.": https://www.barefootnetworks.com /technology/#tofino.

[27] Mininet Website: http://mininet.org/.

[28] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, 2013.

[29] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4), January 2010.

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

[31] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, Savannah, GA, 2016. USENIX Association.

[32] Python Interface for LevelDB: https://plyvel.readthedocs.io/en/latest/.

[33] S. Sanfilippo and P. Noordhuis, Redis: in-memory Key-value store, http://redis.io, 2009.

[34] RocksDB. A facebook fork of leveldb which is optimized for flash and big memory machines, 2013. https://rocksdb.org.

[35] Scapy for Packet Manipulation Website: https://scapy.readthedocs.io/en/latest/.

[36] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow.*, 9(13):1389–1400, September 2016.

[37] Brent Stephens, Aditya Akella, and Michael M. Swift. Your programmable nic should be a programmable switch. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18. ACM, 2018.

[38] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[39] Hao Wen, Zhichao Cao, Yang Zhang, Xiang Cao, Ziqi Fan, Doug Voigt, and David Du. Joins: Meeting latency slo with integrated control for networked storage. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 194–200. IEEE, 2018.

[40] L. Woods, Z. István, and G. Alonso. Ibex-an intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment*, 7:963–974, 01 2014.