In-Memory Key-Value Store Live Migration with NetMigrate

Zeying Zhu[⋆], *Yibo Zhao*[†], *Zaoxing Liu*[⋆]
[⋆]*University of Maryland,* [†]*Boston University*

Abstract

Distributed key-value stores today require frequent key-value shard migration between nodes to react to dynamic workload changes for load balancing, data locality, and service elasticity. In this paper, we propose NetMigrate, a live migration approach for in-memory key-value stores based on programmable network data planes. NetMigrate migrates shards between nodes with zero service interruption and minimal performance impact. During migration, the switch data plane monitors the migration process in a fine-grained manner and directs client queries to the right server in real time, eliminating the overhead of pulling data between nodes. We implement a NetMigrate prototype on a testbed consisting of a programmable switch and several commodity servers running Redis, and evaluate it under YCSB workloads. Our experiments demonstrate that NetMigrate improves the query throughput from 6.5% to 416% and maintains low access latency during migration, compared to the state-of-the-art migration approaches.

1 Introduction

Modern internet services (e.g., e-commerce, mobile gaming, and social networks) depend on large-scale key-value stores as the backend to perform various jobs (e.g., web caching, real-time analytics, and machine learning) [2, 15, 24, 27, 38, 49]. These services often require databases to process queries over ever-growing data volumes and dynamic workload distributions. However, static sharding limits the ability of such systems to adapt to rapidly changing workloads. This may result in degraded performance and Service Level Agreement (SLA) violations due to load imbalance, poor data locality, and insufficient provisioning of cloud resources [28, 33, 37, 51, 52]. Live migration techniques are widely adopted for key-value store reconfiguration [28, 33, 37, 41, 53] that migrates data between nodes without service downtime.

Existing live migration techniques must assume one or more locations — migration source, destination, or both, as the main query serving point because the actual location of a key-value pair during migration is unknown. *Source-based solutions* [23, 33, 53] use source storage to serve all client queries during migration and incrementally migrate dirty data logs to the destination when the keys are updated at the source after their migration. Alternatively, *destination-based solutions* [28, 37] transfer data ownership at the beginning of migration and immediately routes newly arrived queries to the destination. The not-yet-migrated data queried by the client are pulled on demand from the source. However, while two techniques can continuously serve user queries during migration and aim to achieve minimal downtime, they suffer from fundamental performance limitations. Lack of insight about the migration process is a key roadblock in minimizing the overhead caused by accessing the data at the location that does not have ownership. For example, it takes another round-trip latency to fetch a key-value pair from the source if destination-based migration is adopted and this pair has yet migrated to destination.

Ideally, if client queries can always be served at the "right" location during migration, the cost to serve the queries would be minimized. Considering either source- or destination-based migration, extra data movements between the source and destination are necessary when the queried data are not present at the original location. To address this problem, *hybrid migration* techniques take advantage of both source and destination to serve user queries by tracking the migration process in clients [30] or replicate queries to both source and destination [41]. While leveraging both source and destination for query serving during migration is promising for achieving better performance, the cost at the client side to track data ownership and the potential overhead between clients and servers for maintaining consistency are nontrivial.

In this paper, we propose **NetMigrate** to rethink the problem of designing a hybrid live migration approach for in-memory key-value stores. NetMigrate aims at leveraging either source or destination who owns the accessed data chunks (migration state) to achieve pauseless migration and minimal impact on query performance. Compared to bookkeeping of detailed migration states on the client or with additional resources, we argue that the network itself (e.g., top-of-rack switches) would be a better place to track the migration process on the fly because they have a central view of all the data movements in the dedicated rack-scale clusters. With emerging programmable hardware, ToR switches can be programmed to track migration states (e.g., which key-value pairs are migrated) at line rates without latency overhead and can directly route the client queries to the right location (source or destination) who holds the up-to-date data. To our knowledge, NetMigrate is the first proposal to leverage in-network programmability to improve storage migration.

Realizing the promise of NetMigrate, however, is easier said than done, and the use of programmable switches for

migration has several key challenges:

- ***Tracking migration states with limited on-switch resources.*** While existing programmable switches guarantee high-line-rate packet processing capability (e.g., Tbps), the on-switch resources for performing packet-level operations are limited, e.g., O(10MB) SRAM and limited accesses to the SRAM [7, 45]. Given the resource constraints, we leverage probabilistic data structures, i.e., Bloom filters [25] to track if a key has finished migration and counting Bloom filters [29] to track if a key is currently under migration, as the "indexing service" to record the up-to-date migration status. Moreover, to migrate a storage instance with a large number of keys, we support dynamically adjusting the monitoring granularity from a per-key level to a level of a group with multiple keys. With these techniques, 64 MB SRAM on switch will be able to support up to a 34-billion-key storage cluster migration in the same rack[1](§4.2).
- ***Maintaining data consistency during migration.*** To ensure strong consistency, it is critical to understand the location that holds the most updated value of a key and route the new queries to it during migration. It is challenging to keep this information because of the pending state of ongoing migration between source and destination and additional errors from probabilistic migration state tracking. We design an error handling method to ensure correct query results, following this principle: If we have absolute confidence about data ownership, the switch routes the queries to the corresponding location; otherwise, the switch issues small numbers of replicated queries (e.g., double reads) when there is any possibility of imprecise information (§4.3).
- ***Supporting diverse migration policies.*** Some features of existing migration protocols can be useful for certain migration scenarios. For instance, operators may prefer to use destination-based Rocksteady [37] to ensure a short migration time because the resource on the source server can be used primarily for migrating data. NetMigrate can tune its on-switch data structures and resource budgets from the source side to optimize various performance goals, such as minimizing migration time and maximizing query throughput. With the help of switches, NetMigrate can be adjusted to achieve comparable migration time as destination-based solutions such as Rocksteady while offering better query throughput and latency (§4.4).

We implement a NetMigrate prototype in the P4 language [10] (switch side) and C++ (client and server side), and evaluate it on a testbed with an Intel Tofino switch and 3 commodity servers. We migrate a Redis key-value store [11] as an example consisting of 256 million key-value pairs with 4-Byte keys and 64-Byte values, and evaluate NetMigrate on YCSB workloads [17] with different write ratios and load-balancing scenarios against the state-of-the-art approaches. Experimental results demonstrate that NetMigrate achieves zero downtime during migration, improves the average query throughput by 6.5% to 416%, while maintaining low access latency during migration and incurring negligible extra bandwidth overhead. NetMigrate is open-sourced at [9].

[1]As in § 4.2, assuming 16 bits per element in Bloom filter and counting Bloom filter and each group has 2^{10} keys, 64MB SRAM can support 34 billion key-value pairs (64MB*8/16*1024). With 4-Byte keys and 64-Byte values, the total storage size is ∼2TB.

2 Background and Motivation

In this section, we first discuss the key-value store live migration problem and its requirements. We then analyze existing approaches and their advantages and limitations.

2.1 Key-Value Store Live Migration

In distributed key-value store systems, data sharding can be reconfigured over time for load balancing, access-locality improvement, and cluster horizontal scaling (e.g., when a new node joins the cluster, it "steals" keys from other nodes). Storage instance migration improves spatial locality to enhance item access throughput and reduce access latency to backend servers [20, 28, 34, 37] and provide load balancing among dynamic and skewed workloads between servers [28, 32, 42].Migration can also happen when there is an upgrade of the server hardware or cluster horizontal scaling.

Data migration between shards can introduce service downtime and performance degradation. However, during migration, storage cluster should still provide service reliability and meet the Service Level Agreement (SLA). For example, even a slight service outage has significant financial consequences for a large-scale e-commerce platform and can harm the customer's trust [27]. Thus, live migration techniques, which move data between nodes without causing client-observable downtime, become a key enabler to achieve elastic key-value stores in the cloud environment.

In this paper, we focus on live migration of in-memory key-value stores, such as Redis [11], Apache Cassendra [1], RAMCloud [46], and Memcached [8]. These key-value stores keep all data in DRAM and can scale across thousands of data center servers. Under these storage systems, they often construct hash-table data structure for storing and indexing key-value pairs. We focus on alleviating migration performance degradation to the minimum. We assume that there is an internal or external cluster scheduler that collects statistics of the storage cluster and generates reconfiguration plan on when and how the data should be re-sharded and migrated to fit the current workloads.

Performance requirements. Common metrics used to evaluate a live migration system include service downtime, query throughput and latency, transferring extra data (extra network bandwidth usage), and migration completion time. For migration approaches, there is a fundamental trade-off between migration completion time and migration cost (e.g., drop in query performance and transfer of extra data). The shorter the migration finish time, the higher the migration cost. An

Migration Protocols	Example Systems	Downtime	Latency	Throughput	Extra Data	Client Overhead	Migration Time
Stop-and-Copy	Redis MIGRATE [13], Slacker [23]	Yes	High	Low	No	No	Short
Source-based	RAMCloud [46], Remus [33], DrTM+B [53]	Minimal	Low	Medium	Yes	No	Long
Destination-based	Rocksteady [37]	No	High	Low	No	Yes	Short
Hybrid	Fulva [30]	No	Medium	Medium	No	Yes	Medium
Hybrid	**NetMigrate**	**No**	**Low**	**High**	**Negligible**	**Negligible**	**Medium (Adjustable)**

Table 1: **Overview of live migration approaches.**

ideal live migration approach is expected to provide minimal migration cost while maintaining an acceptable migration finish time.

2.2 Existing Approaches and Limitations

Stop-and-copy is a basic form of migration, which consists of freezing the storage server (with a read lock), copying the key-value data to the destination server, and then deleting them in the source server. For example, Redis MIGRATE command [13] implements this stop-and-copy at the per-key level. If migrating the entire store to the destination server, a faster way is to shutdown the source key-value store, create a snapshot file, perform a file-level copy of the compressed snapshot to destination, and then start a new key-value store instance on the destination server pointing to the copied snapshot directory. The main downside of stop-and-copy is the significant downtime caused by shutting down the storage instance, which affects the client execution logic. The length of copying period is proportional to the database size [23].

To perform live migration, there are three existing migration approaches: *source-based, destination-based, and hybrid (both source and destination)*.

Source-based approaches choose the source to own the data during migration, and thus all client read and write queries are served by the source [22, 23, 53]. The source node iteratively migrates "dirty data" (data in the source that are already migrated but later updated) to the destination, which transfers additional data. Although source-based approaches can serve client queries without adding query latency, they have to terminate the source server at some point to copy the last piece of dirty data to the destination, incurring unavoidable service pauses. Source-based approaches have low query latency, but long migration time because migration operations compete with client queries at the source node.

Destination-based approaches choose the destination server to hold data ownership and serve client queries [28, 37]. All read and write queries will be routed to the destination. To serve data that have not yet migrated, the destination needs to pull the data from the source, and the client will have to retry after a wait. Therefore, destination-based solutions incur high query latency on not-yet-migrated items, especially at the beginning of migration, because most of the data are still on the source node. Meanwhile, destination-based approaches usually migrate data faster than source-based ones due to more resources available at the source.

Hybrid approaches (e.g., [30, 41]) can choose the destination node to handle write queries, and send read queries of not-yet-migrated and on-the-fly data to both source and destination nodes to achieve data consistency. Hybrid approaches need to keep track of migration process somewhere. For example, Fulva [30] tracks completed migration ranges in their key-value store client libraries. This type of approaches avoids on-demand data transfer between the source and the destination but instead uses additional network bandwidth (due to two read packets). Double-read incurs large resource overheads ($\sim 50\%$) among clients and two storage nodes to guarantee the protocol consistency because there is no fine-grained migration status tracking. Clients see the reply results from the node with a newer version and thus it can increase the latency by waiting for two replies.

Summary. Table 1 summarizes the performance characteristics and strengths/weaknesses of existing data migration protocols. We posit that data migration tasks usually have upper resource limits, and thus foreground client queries should be put on a higher priority than migration in the storage cluster. All existing live migration approaches have performance degradation and trade-off between migration cost and migration time. Our NetMigrate design provides a new alternative to improve these dimensions and reevaluate the trade-off between performance and migration time. By comparing the migration protocols, hybrid approaches do provide better migration performance compared with simply destination-based or source-based approaches, as they reduce the number of queries going to the wrong nodes.

Opportunities of programmable switches. Our aim is to design a switch-accelerated hybrid migration approach. A ToR switch positions centrally in all inter-server communications and acts as the gateway to other racks. This allows it to see all the migration and query traffic, enabling migration status tracking without additional communications. Host-based alternatives typically require sending migration status to a specific location (e.g., clients as in Fulva) or dealing with multiple requests going to the wrong place and pulling from another (e.g., Rocksteady or Slacker/Remus). Unlike CPUs, most programmable switches (e.g., Broadcom [16], Juniper [5], Intel [7]) are ASIC-based and offer flexible programmability without performance loss when performing customized modules. They can also guarantee high line rates such as 12.6 Tbps, orders-of-magnitude higher throughput and lower latency than servers. Therefore, deploying migration indexing service on switches can alleviate clients' or cluster coordinators' bookkeeping overheads when using hybrid approaches.

3 NetMigrate Overview

System architecture. NetMigrate is a rack-scale key-value store live migration accelerator leveraging in-network programmability. NetMigrate enables ToR switch as a migration state tracking service and routes client queries to the "right" server (either source or destination) during migration based on the latest information on the switch about what data have been completed migration or under migration. Fig. 1 shows the overall architecture of NetMigrate, which consists of a ToR switch, a controller, clients, and servers:

- **ToR switch** provides the following functionalities for the live migration service: (1) a *migration state table* module tracks migration states of each group of key-value pairs, indicating the data ownership belongs to the source or the destination. It uses probabilistic data structures and serves as an indexing service to determine where the client queries should go (§ 4.2); (2) a *routing* module routes client queries to the "right" storage server under migration (best-effort) based on the migration state table (§ 4.3); and (3) a *migration instance table* module records multiple pairs of key-value stores that are under migration for enabling (re-)routing client queries to the right corresponding storage.
- **Storage servers** store key-value data and serve client queries. In NetMigrate, we consider migration can occur between multiple storage instances within the same rack. Storage servers host key-value stores and run a migration agent that (1) maps key-value store API to NetMigrate migration packets, (2) serves client queries with consistency guarantees, and (3) handles data transmission between migrating storage instances. The migration agent makes NetMigrate easy and general to integrate with different backend key-value stores.
- **Clients** issue storage queries. NetMigrate provides a client agent to support the switch-based migration system. The client agent maps queries (e.g., `GET`, `SET`, `DELETE` commands) to the switch-based query packets, and transform NetMigrate reply packets into the backend storage commands. Migration process is transparent to client applications.

Challenges and Key Insights. To realize NetMigrate, we need to address several key design challenges:

- ***Fine-grained migration state tracking with limited on-switch resources.*** There is a disconnect between the potentially large key-value data to migrate and limited on-switch resources (e.g., SRAM, TCAM, etc.) that can be used to track migration status. It is infeasible to record the status of every single key. Therefore, we combine a number of key-value pairs together as a *group* and and record migration state at the group level. However, there is a tradeoff here: a too large group size (i.e., a small number of groups) limits the switch's ability to accurately determine the right destination for many queries, thereby compromising performance benefits, and vice versa. We need a memory-efficient design on the switch to support a large number of migration groups and the group sizes are relatively small. NetMigrate specifies three group-level migration states, *not-yet-migrated*, *ongoing-migration*, and *complete-migration*, to support fine-grained routing operations for client queries. At a high level, we use probabilistic data structures – Bloom filters (BF) [25], counting Bloom filters (CBF) [25, 29], and the hybrid of the two. We choose BF and CBF because of their memory efficiency and the fact that they can cover the required state tracking. BF tracks migrated keys at group-level and once a membership of a key is inserted, it cannot be removed. CBF tracks only ongoing-migration keys because we can delete keys from it once the keys are migrated. Hybrid of the two can indicate not-yer-migrated state. This choice leads to memory efficiency and simplistic design because Bloom filters can probabilistically perform membership tracking and involve only hashes and simple arithmetic operations.

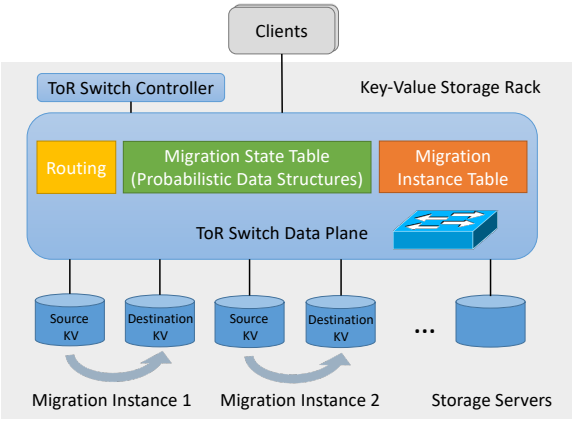


Figure 1: **NetMigrate system architecture.**

- ***Maintaining data consistency during migration.*** We consider the linearizability requirements [31] in consistency, including Read-After-Write (RAW), Write-After-Write (WAW), and Write-After-Read (WAR). To ensure consistency, it is critical to understand the right location that holds a key's up-to-date value and route the new queries to it during migration. The consistency issue becomes more challenging when our migration state tracking brings additional errors (e.g., false positives in BF and CBF). We propose a fine-grained error handling method to ensure correct query results as in § 4.3. At a high level, the key principle is that **we always route write queries to the destination unless we are sure that the migration has yet started, and issue read queries to both locations when we are definitely unsure about the migration state.** When we are *almost* confident that the data are on the destination but can have false positives, e.g., BF shows positive (the group has migrated) and CBF shows not positive (not under migration), we will route read queries to destination (and issue data pulls to the source if errors) instead of double

reads because the false positive rates are relatively small and we can reduce the workload for the source.

- ***Dynamic migration policies.*** One disadvantage of using existing live migration approaches is they have to sacrifice on or optimize towards a fixed set of dimensions (migration time or query performance). However, the operator may have different performance objectives when planning a migration (e.g., minimizing migration time or maximizing query performance) [43]. NetMigrate can optimize toward various performance goals (simulating any other protocols) and dynamically change the migration policy by tuning the probabilistic data structures in the switch and adjusting the CPU utilization of the source server. For example, we can simply set all BF entries to 1 to mimic destination-based solutions like Rocksteady [37]. To optimize migration time while offering better performance than source-based solutions, we can limit CPU usage to serve client queries and leave more CPU headroom for migration.

4 NetMigrate Design

In this section, we discuss the design of NetMigrate and describe how an in-network accelerator can help live migration.

4.1 Migration Workflow

Fig. 2 shows NetMigrate's general migration process. It starts from the current (source) server to a new server (destination) capable of accommodating the key-value shard. The source server initiates migration by notifying the destination server and registering the migration instance in the switch via *control packets* (Step ①). Throughout migration, clients remain unaware and continue sending queries to the original storage instances. The ToR switch decides whether a query should go to the source or destination based on migration status, and clients receive replies as if they are from the source server. During migration, the source server concurrently migrates data to the destination using *data packets*. The switch tracks the migration at the key-value *group* level, with each *group* containing multiple key-value pairs (Step ②). The destination server receives and replays these packets into its storage structures. Upon completion of the migration process, the source server agent issues a termination notification to the destination, switch, and clients (Step ③). In response, the source server cleans up its database, the switch removes the migration pair registration, the destination server takes over data ownership, and clients direct queries to the destination.

4.2 Migration State Tracking

NetMigrate accelerates key-value store live migration by tracking the migration states entirely in a central position (in the network) and using this in-network information to determine where to route the user queries as precisely as possible. However, it's impossible to store every migrated key given the large key space. Therefore, we make two design choices to shrink the tracking space requirement: (1) a combination

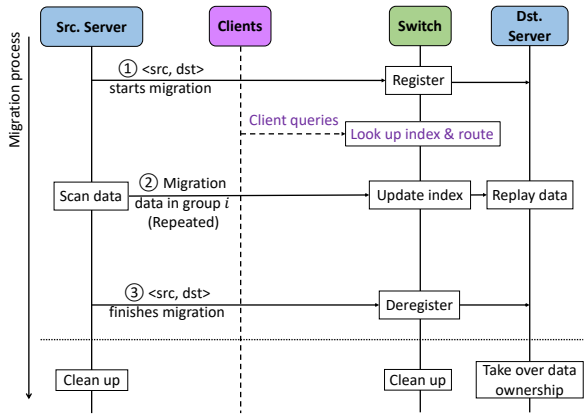


Figure 2: **NetMigrate migration workflow.**

of probabilistic data structures such as Bloom filers (BF) and counting Bloom filters (CBF) to track migration states with low false positives and fine-grained migration state, and (2) recording data ownership at a coarse granularity of *groups*. Each migration *group* represents a set of adjacent entries in the underlying key-value storage, e.g., several adjacent buckets in the hash table. These enable NetMigrate to scale to a large number of key-value pairs while maintaining high accuracy and low resource overhead.

Hybrid Bloom filters. The combination of BF and CBF is used to track three migration states: (`S1`) Entire group has not started migration; (`S2`) The group is under migration and only a subset of key-value pairs maybe migrated; (`S3`) Entire group has completed migration. Once a group (of key-value pairs) has completed migration, this group is recorded in BF. CBF tracks the "ongoing-migration" groups. When a group has started migration, this group is added to the CBF until migration is done. Compared to using a single BF to track which group(s) have completed migration, our hybrid design with both BF and CBF provides more fine-grained migration states and reduces false positives.

Specifically, when a group starts migration, the corresponding CBF entries are incremented by 1; when the group finishes migration, the same CBF entries will be deducted by 1. The states of each migration group are updated by its migration control packets – `GROUP-START-MIGRATION` and `GROUP-COMPLETE-MIGRATION` packets in the switch data plane (detailed packet format description is in Appendix A). Compared to using a single BF with the same memory space, NetMigrate's hybrid filters significantly reduce the false positives rate that can lead to throughput and latency degradation. We configure BF and CBF based on the following. For 2^x groups and 2^y migration parallelism (i.e., there are 2^y threads in total migrating key-value instances), the false positive rate upper bound of combining BF and CBF together is approximately $1-(1-(1-e^{\frac{-kn}{m}})^k)(1-(1-e^{\frac{-k'n'}{m'}})^{k'})$, where k and k' are the number of hash functions, n and n' are the total number of groups (elements), and m and m' are the number of entries used in BF and CBF respectively [25]. Here we

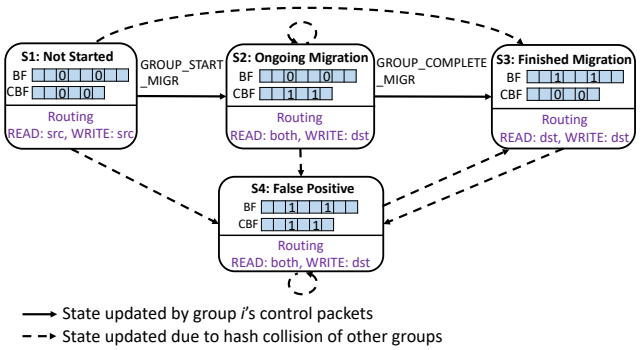


Figure 3: **NetMigrate migration state tracking and query handling state machine.** *Assuming two hash functions are used in BF and CBF, the values in BF and CBF arrays are located at the hashed indices of a particular group.*

consider the CBF's false positive is the same as BF's and use $k = k' = 4$ for both BF and CBF. If the target combined false positive rate is less than 1%, a good-enough configuration is using 2^{x+4}-bit Bloom filter and counting Bloom filter with 2^{y+4} entries (8 bits per entry) to achieve less than 0.5% combined false positive rate, resulting 16 bit-per-element space complexity. Also, 2^{x+3}-bit BF and 2^{y+3}-entry CBF (about 8 bits per element) can achieve a 5% false positive rate.

Group size tuning. The number of keys in a group is also a critical parameter that affects performance during migration. When the group size is smaller, the pending period where the switch cannot determine the location of the key is shorter. During the pending state (state (S2)), NetMigrate has to handle the wrong locations, adding up the performance degradation. In practice, there is an upper bound of the group size under which the performance impact is acceptable. As evaluated in § 6.4, the upper bound is around 2^{20} keys in a group. The lower bound of group size (namely, the upper bound of the number of groups given the total number of keys) is bounded by the BF and CBF sizes that the data plane can offer. Considering BF, CBF sizes and group sizes together, theoretically, 64 MB SRAM on switch will be able to support up to a 34-billion-key storage cluster migration in the same rack.

4.3 Data Consistency During Query Handling

It is important to avoid introducing additional inconsistency to the backend key-value stores during migration. As guaranteed by existing migration protocols, we consider the strongest data consistency (linearizability) [31] when designing our migration protocol – including Read-After-Write (RAW), Write-After-Write (WAW) and Write-After-Read (WAR) when handling client `READ` and `WRITE` queries. A migration state machine in Fig. 3 demonstrates the migration index tracking and query routing decisions in the switch data plane and to show how our protocol guarantees consistency during migration.

Migration packets updating indexes. Recall that there are three group migration states (`S1`-`S3` in Fig. 3). However, due to probabilistic errors and hash collisions in BF and CBF, there is a fourth migration state – false positive (`S4`). `S4` indicates a false positive because the BF and CBF entries of one group cannot be both positive simultaneously. We define migration control packets `GROUP-START-MIGRATION` to inform the switch that a specific group starts migration, and `GROUP-COMPLETE-MIGRATION` to notify the completion of the migration for a group. When a `GROUP-START-MIGRATION` packet arrives at the switch, it increments the CBF entries by 1, indicating a new ongoing migration. This transits state `S1` to `S2`. When a `GROUP-COMPLETE-MIGRATION` packet arrives, it sets the BF entries as 1 and decrements the corresponding CBF entries by 1, indicating that the group has finished migration. Thus, this action transits state `S2` to `S3`. Other transitions shown in Fig. 3 are false positives caused by hash collisions with other groups.

Query routing based on migration status look-up. Each state in Fig. 3 also shows index look-up results from BF and CBF and outlines query routing decisions. The general principle is: In the state machine, no state returns to a state with `READ` queries from the source (i.e., state `S1`). If a state `WRITE` to the destination, all possible following states are `READ` from either the destination or both nodes, ensuring access to the latest data. The states are explained as follows:

State `S1` means the group does not start migration, so both `READ` and `WRITE` queries are directed to the source. There are no false positives because both BF and CBF entries are 0.

State `S2` means the group is currently migrating. For `READ` query, the switch cannot determine whether the queried key-value pair is still on the source server, or on-the-fly, or on the destination server because the migration tracking granularity is larger than the per-key level. In this case, the switch generates double-read query packets via packet mirroring, where the original query is still forwarded to the source and a mirrored query is sent to the destination server. Thus, the client will receive two `READ` reply packets for this one `READ` query and merge two `READ` replies. If the destination has a successful reply, the client ignores the reply from the source because the destination may have updated values; otherwise, the key has not been migrated to the destination, so the client takes the source reply. We route the `WRITE` query to the destination in this state because the `READ` queries are doubled to both source and destination servers, and the client can always read the latest value. There can be false positives from CBF in this state, which are handled by double reads. The double-read ratio is low because the period when a group is under migration usually does not last long.

State `S3` stands for the group that finishes migration. Both `READ` and `WRITE` queries can be directed to the destination. There can be false positives from BF, and `READ` queries will be falsely directed to the destination while the data have not been migrated. In this case, NetMigrate agent on the destination issues PriorityPulls on-demand [37] to retrieve the missing key-value pairs from the source and respond to the client. This step also corrects the false positives of BF for subsequent queries as the keys are already at the destination.

State `S4` represents a false positive case because a group cannot be in both a complete-migration and an ongoing-migration state simultaneously. To correct the false positive, we also use double-read for `READ` queries and direct `WRITE` queries to the destination.

Handling corner case. There is a corner case that some migrated key-value pairs can still be updated by `WRITE` queries in the source node. This happens when the `WRITE` query updates a key-value pair that is about to migrate. Specifically, this case occurs when a `WRITE` query arrives at the switch first and looks up the migration index. The index indicates that the key-value pair is still in the source node, and the switch forwards this query to the source. Next, the `GROUP-START-MIGRATION` packet is sent from the source, arrives at the switch, and updates the migration index. At the source, the data migration packet containing this key-value pair has been sent to the destination before the `WRITE` is executed. To guarantee data consistency in this case, we collect the late updates in memory at the source and periodically transfer the late dirty logs to the destination as a source-based protocol.

In summary, NetMigrate can ensure data consistency during migration. Moreover, experiments in § 6.4 show a low overhead to correct false positives, with the portion of double reads and PriorityPulls being less than 0.05%, and less than 4×10^{-5}% extra bandwidth overhead incurred by the late dirty logs.

4.4 Dynamic Migration Policies

There is a fundamental tradeoff between migration time and the query performance: The migration completion time is related to the source's CPU headroom left for migration, while the query performance also depends on the source's and destination's CPU cycles for query serving. By configuring migration CPU cycles and taking advantages of Bloom filters, NetMigrate has more flexibility to be configured to support various migration goals, such as minimizing migration time or optimizing query throughput and latency.

- **Minimize migration time.** As shown in the experiments, Rocksteady has the shortest migration time because all the source CPU cycles can be used for migration. To achieve the similar migration time as Rocksteady, one way is to simply pre-set all BF entries as 1, indicating that all queries should be routed to the destination. NetMigrate will issue PriorityPulls to fetch not-yet-migrated keys. Thus, NetMigrate's protocol is now essentially the same as that of Rocksteady. However, this strawman solution only gives NetMigrate the same query performance as Rocksteady. Alternatively, we can limit the CPU cycles for serving client query in the source to a *low* level and leave more CPU headroom for migration. By doing so, NetMigrate achieves a similar migration time as Rocksteady, while gaining some throught and latency benefits because BF correctly identifies the keys belonging to the source.
- **Maximize query throughput and minimize latency.** NetMigrate is designed to gain more benefits in query performance from the source and destination. To maximize query throughput and minimize query latency, we set the CPU cycles in the source for client queries to a *medium* level, and leave some CPU headroom for migration. By doing so, NetMigrate achieves the highest throughput and lowest median latency compared to other baselines while maintaining a similar migration time, as detailed in §6.2.
- **Mimic other migration protocols and take advantages of all.** An interesting feature of NetMigrate is that it can be configured to hybrid and source-based protocols in addition to Rocksteady because its design takes fine-grained migration states into consideration. To make it equivalent to Fulva (hybrid protocol), we can pre-increment all CBF entries by 1 so that it will be in state `S2` or `S4` forever. To make it the same as a source-based protocol, NetMigrate needs to disable the BF and CBF updates, which keeps its state in `S1`. NetMigrate also consists of transferring late dirty logs from the source to the destination, similar to a source-based protocol. In addition to these, we observe in the evaluation (§ 6.3) that a medium-size BF and CBF can give the best query performance, e.g., the 8-bit-per-element setting in Table 2, compared to the ones with more BF and CBF space. This is because some false positives in the switch index actually shift the query workload from the source to the destination, which gives more CPU headroom for the source to migrate data and helps move the workloads to destination faster. Thus, by adjusting some false positives of BF and the headroom of the source CPU for migration, NetMigrate can take the performance advantage of both destination-based and source-based approaches while maintaining data consistency.

5 Implementation

We have implemented a prototype of NetMigrate with Redis [11] as an example, including the programmable data plane serving as a migration index, the migration server agents, and the client running YCSB workloads. The indexing switch is implemented with 2K lines of P4-16 code and is compiled to Intel Tofino ASIC [7]. We implement the migration instance table using a P4 table and the migration state table using BF and CBF where each BF entry is 1 bit and each CBF entry has 8 bits. Both BF and CBF use 4 different hash functions. We implement L3 routing and redirect client queries by changing their destination or mirroring queries to both storage nodes. The switch control plane is implemented with 600 lines of Python code, which registers and deregisters the migration instances by modifying the migration instance table in the control plane. The migration server agents and clients are implemented with 15K lines of C++ code. In the prototype, we use the Redis-plus-plus library [14] to communicate with Redis instances in migration servers. We add three new User-Defined Functions to get the current hash ta-

ble information for migration and to scan key-value pairs in the order of hash values. Source server agents call the user-defined commands, scan key-value pairs in parallel, and send migration control and data packets via UDP sockets. Destination server agents receive migration packets and insert data into the destination instance. We modified the C++ YCSB client [17] for key-value store and UDP communication.

6 Evaluation

We conduct extensive experiments comparing NetMigrate to the latest live migration solutions and demonstrate:

- NetMigrate improves the query throughput by 6.5% to 416% under different YCSB workloads and load-balancing scenarios (§6.2 and §6.5) while keeping low tail latency.
- NetMigrate supports diverse migration policies with different performance goals, including query throughput, latency, and migration completion time. It improves the average query throughput from 32% to 78% compared to baseline protocols with similar migration time (§6.2).
- NetMigrate can achieve the above improved performance with slim BF and CBF space allocation within the switch memory limitation (§ 6.3).
- NetMigrate incurs negligible bandwidth overhead (§6.4).

6.1 Methodology

Testbed. The experiments are conducted on a testbed consisting of one 6.5 Tbps Intel Tofino switch and 3 commodity servers. Each server is equipped with an 8-core CPU (Intel Xeon E5-2620 @ 2.10GHz), 64GB total memory (two 32GB DRR4-2400 DRAMs), and one 40G NIC (Intel XL710).

KV workloads. By default, we create Redis key-value stores consisting of 256 million key-value pairs (∼16GB), occupying 52.7% memory of a server (33GB including Redis indexing data structures), with 4-Byte keys and 64-Byte values. We use YCSB benchmark [26] designed for key-value stores evaluation. The queried keys are generated randomly according to the Zipfian distribution with $\theta = 0.99$. We use 5% write ratio and 100% source Redis CPU usage budget to show the overall performance impact in § 6.2. We furthertune the workload write ratio among 0% (YCSB-C), 5% (YCSB-B), 10%, 20%, and 30%, and limit source Redis CPU to different budgets, i.e., 100% (not overloaded), 70% (slightly overloaded), and 40% (heavily overloaded), using `cpulimit` [3] to create different load balancing scenarios in § 6.5.

Evaluation parameters and metrics. By default, we set BF size to 512 KB and CBF size to 1024 KB, with which uses 4 hash functions. The default CBF size of 1024KB was a sufficiently large starting point as we were not sure how many KV pairs are on-the-fly during migration. Additional sensitivity tests in Table 2 show that 1024KB CBF is usually an overkill but it's significantly smaller than the total switch memory. We configure 2^{17} migration groups, each of which has up to 2048 key-value pairs. In the experiments, we show client-observed performance metrics, such as Queries per Second (QPS), end-to-end latency, and migration completion time. We use extra bandwidth percentages between a client and servers (denoted as client-size), and between the source and destination servers (denoted as server-size) to evaluate the extra migration overhead.

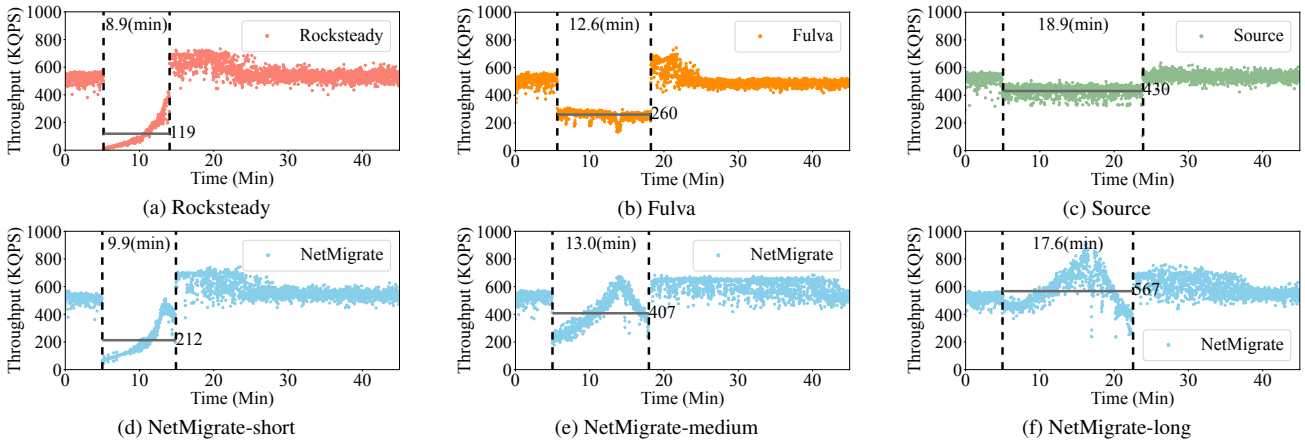
Baselines. We implement three types of migration systems and their protocols (as discussed in §2.2) in our testbed for a fair comparison. All baselines follow the same data I/O and network protocols, export the key-value pairs from the source, and use the migration agents at both the source and the destination to transmit the key-value data, as shown in Fig. 2. The difference is that they do not use switch indexing. We implement (1) *source-based protocols* including Slacker [23] and Remus [33]; (2) *destination-based protocols* including Rocksteady [37] with gRPC asynchronous API [6] to implement the PriorityPull RPCs; and (3) *hybrid protocols* including Fulva [30]. In particular, the client in Fulva keeps track of the migration progress and the hot keys are migrated with a higher priority based on sampling statistics.

6.2 Overall Performance

In this experiment, we consider a migration scenario where both the source and the destination are not overloaded and have 100% CPU budgets for Redis. We show the client-side throughput and latency during migration using YCSB-B workloads with a 5% write ratio.

Query throughput. This experiment highlights the throughput difference and performance trade-offs in different migration protocols. Fig. 4 (a), (b) and (c) show the throughput and migration time of the three baselines. Compared to the three baselines, NetMigrate improves the average query throughput by 78.2%, 56.5%, and 31.9% when it is configured as high, medium, and low migration speeds respectively. Among the baselines, (1) Rocksteady has the lowest throughput and also the shortest migration time because all queries are handled by the destination Redis, leaving the most CPU headroom for source Redis to perform migration. (2) Fulva's client throughput has been cut by a half compared to a fully-utilized Redis instance's throughput because of the overhead caused by double-reads and being bounded by the packet rate. (3) Source baseline's throughput keeps stable during migration and it is slightly lower than without migration because it uses the smallest portion of source Redis CPU for migration and thus its migration time is the longest.

When zooming into NetMigrate, we can see that NetMigrate's throughput first increases to a peak level and then drops as depicted in Fig. 4 (d), (e), and (f). This is because at the beginning of migration, client queries are mainly served by the source. During migration, increasing numbers of key-value pairs are migrated to the destination. The destination Redis can serve queries of the already-migrated data, and thus total query throughput increases. When most data are migrated to the destination, destination is pressured from both



(a) Rocksteady (b) Fulva (c) Source

(d) NetMigrate-short (e) NetMigrate-medium (f) NetMigrate-long

Figure 4: **Comparing Rocksteady, Fulva, Source-based protocol, and NetMigrate on query throughput with YCSB-B workload and 100% source CPU budget.** *We report the average throughput as the horizontal lines, label the migration start and finish timestamps as dotted lines, and show migration time of each protocol in the figures. NetMigrate-short, NetMigrate-medium, and NetMigrate-long denote the experimental results when NetMigrate is configured to different migration time policies.*

migrated data insertion and query serving, so the throughput drops when migration is nearing the completion and there is a peak throughput point during migration. The peak throughput is even larger than a single Redis' query throughput because the ToR switch lets the client leverage both the source and the destination Redis's query power.

Query latency. We then evaluate the latency percentiles as shown in Fig. 5 and 6. The average median latencies of Rocksteady, Fulva, and Source baseline are all larger than NetMigrate's under any migration time configurations. NetMigrate reduces the average median latency from 49% to 65% in all cases. For average 99%-tail latency, NetMigrate is better than Rocksteady and Fulva when it's configured to the similar migration time, reducing the latency by 27.0% and 39.5% correspondingly. NetMigrate-long's 99%-tail latency is almost twice than Source baseline's because in the worst case, NetMigrate still needs to wait for two replies from both source and destination Redis and it can have PriorityPulls for wrongly directed queries.

NetMigrate's adaptable migration policies. NetMigrate can adjust between migration cost and migration time based on the user needs. We limit source Redis client query processing CPU cycles to adjust the CPU headroom for migration, and NetMigrate can migrate data with high migration speed (similar as Rocksteady), medium migration speed (similar as Fulva), and low migration speed (similar but better than Source baseline) based on the configurations. Fig. 4, 5, 6 (d), (e), and (f) show NetMigrate is adaptable to different migration time requirements and demonstrates different query performance levels. Also, NetMigrate can achieve similar migration time while maintaining higher throughput and lower access latency compared to all three baselines (except for comparing to Source baseline in the case of 99%-tail latency).

6.3 Tuning Bloom Filter Sizes and Group Sizes

Bloom filter size tuning. We evaluate the impact of BF and CBF sizes on migration and query performance. We also run real migration experiments changing the BF and CBF sizes with totally 2^{17} migration groups and 4 threads to migrate in parallel. Table 2 shows that combining BF and CBF reduces false positives in practice significantly. The client-side extra bandwidth usage can reveal actual false positives. In Table 2, given a large enough BF size, we shrink the CBF sizes and find that 64 Bytes CBF is the tuning point before performance drops. Keeping the good-enough CBF size, we shrink the BF sizes. Results show that 8-bit-per-element gives the best performance while avoiding wasting too much space. When the actual false positive rate is too high, e.g., when space complexity is less than 2 bits per element, client-side extra bandwidth usage and 99%-tail latency are worse due to the increased number of PriorityPulls and double-reads to correct false positives.

Group size tuning. Given large enough BF and CBF sizes, e.g., 512KB BF and 64B CBF, we tune the group size (i.e., the total number of groups). Table 3 shows that when group size (i.e., the number of keys in the group) is larger than 2^{20}, the throughput and latency will be harmed because of the increased double-reads when the queried key is in an ongoing-migration group.

6.4 Extra Overhead for Migration

Extra bandwidth usage. Table 4 shows extra bandwidth usage under all write ratios and source CPU budget settings. Source protocol's extra bandwidth usage only comes from the server side, where the source server needs to transfer dirty logs to the destination when `WRITE` queries are later than the migration of their keys. Rocksteady's only comes from the client side, where client needs to retry queries with PriorityPulls. For Fulva, double-reads contribute to almost a

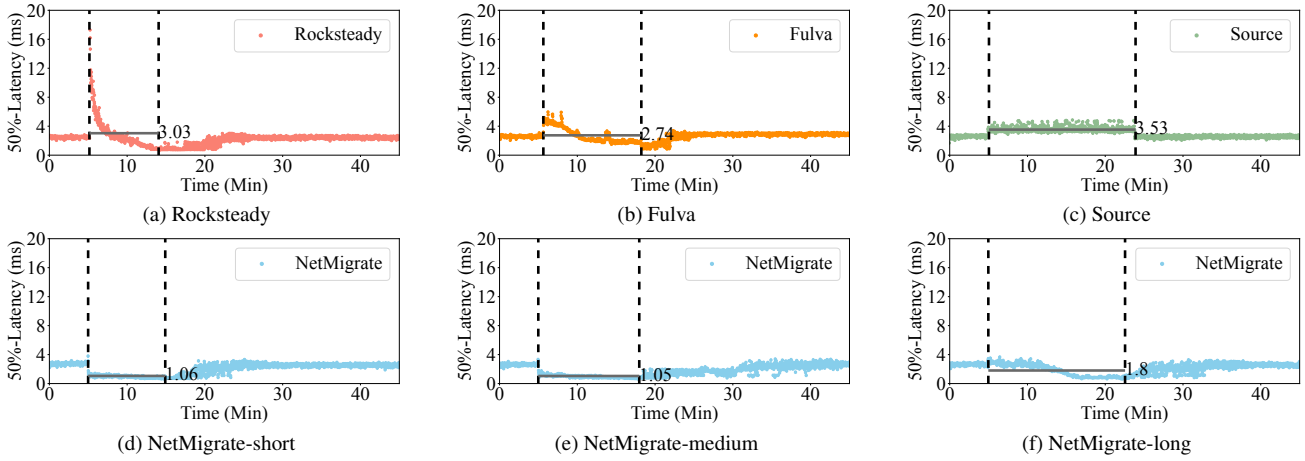


Figure 5: **Comparing Rocksteady, Fulva, Source-based protocol, and NetMigrate on median latency with YCSB-B workload and 100% source CPU budget.** *We report the average median latency as horizontal lines and label migration start and finish timestamps as dotted lines.*

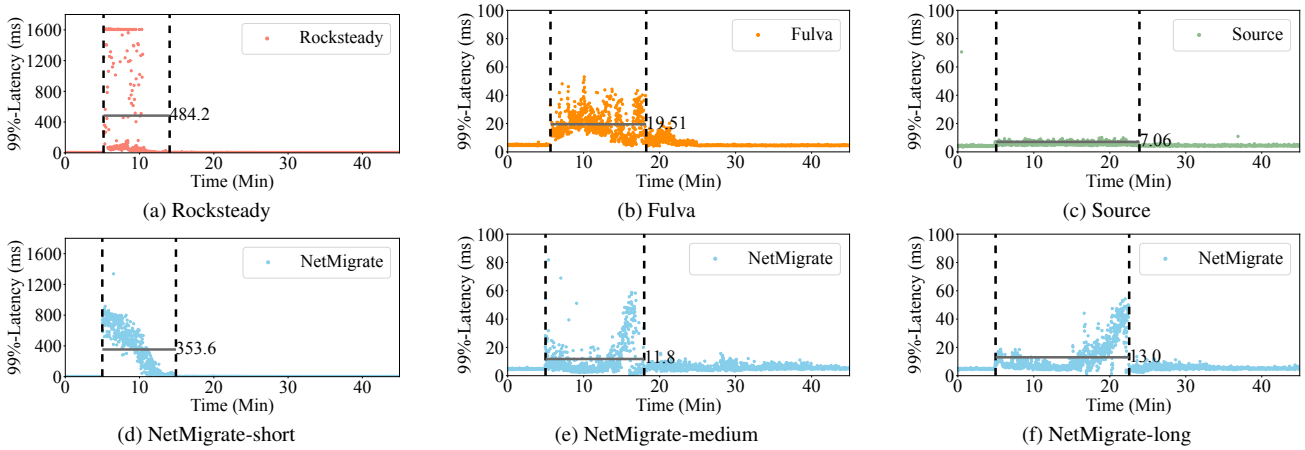


Figure 6: **Comparing Rocksteady, Fulva, Source-based protocol, and NetMigrate on 99%-tail latency with YCSB-B workload and 100% source CPU budget.** *We report the average 99%-tail latency as horizontal lines and label migration start and finish timestamps as dotted lines.*

BF, CBF size	FP	Bits/Ele	Throughput	Median, 99% Latency	Client BW
512 KB, 128 B	0.038%	32	564.1 KQPS	1.13 ms, 7.83 ms	0.06%
512 KB, 64 B	0.26%	32	572.7 KQPS	0.92ms , 7.23 ms	0.30%
512 KB, 32 B	2.42%	32	362.5 KQPS	0.89 ms, 26.5 ms	1.37%
512 KB, 16 B	16.0%	32	197.5 KQPS	1.15 ms, 190.2 ms	3.73%
256 KB, 64 B	0.48%	16	569.6 KQPS	0.94 ms, 5.61 ms	0.30%
128 KB, 64 B	2.63%	8	573.2 KQPS	0.93 ms, 4.47 ms	0.30%
64 KB, 64 B	16.2%	4	563.6 KQPS	0.95 ms, 5.42 ms	0.30%
32 KB, 64 B	56.0%	2	523.8 KQPS	0.93 ms, 4.38 ms	0.47%
16 KB, 64 B	92.9%	1	495.8 KQPS	1.01 ms, 5.73 ms	0.65%

Table 2: **Impact on migration when tuning BF and CBF sizes.** *"FP" represents the upper bound of combined false positive rates of BF and CBF. "Bits/Ele" stands for total BF and CBF bits per element. "Client BW" means the extra bandwidth usage between the client and servers, compared with total query traffic. In the settings listed in the table, the server-side extra bandwidth usages are all less than* 6×10^{-5}*% and negligible.*

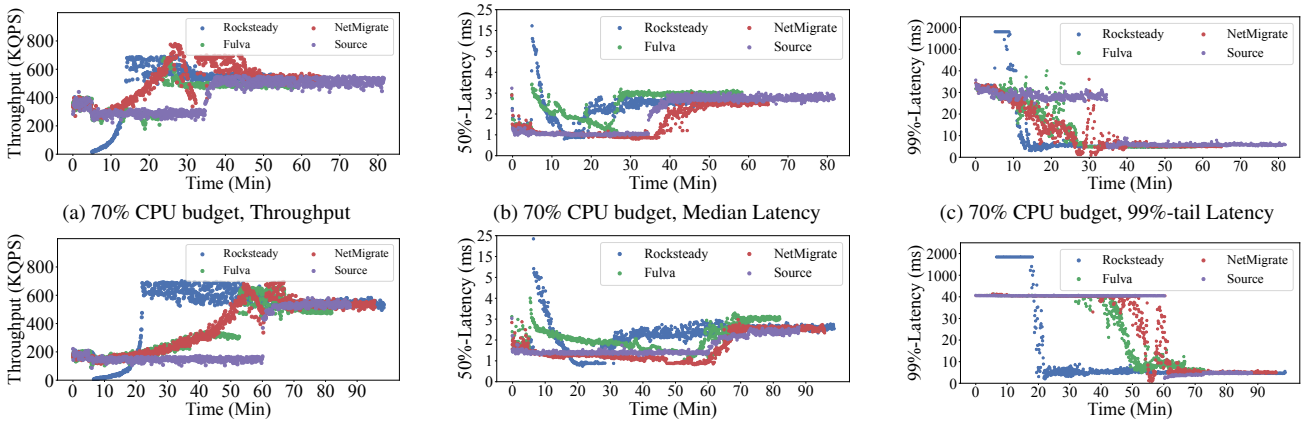
Group size	Throughput	Median, 99% Latency	Client BW
2^{11}	567 KQPS	2.33 ms, 8.46 ms	0.0060%
2^{14}	599 KQPS	1.05 ms, 5.19 ms	0.0165%
2^{18}	573 KQPS	0.95 ms, 10.97 ms	0.2613%
2^{19}	561 KQPS	0.91 ms, 3.82 ms	0.3733%
2^{20}	521 KQPS	0.9 ms, 3.63 ms	0.4858%
2^{21}	494 KQPS	0.9 ms, 4.01 ms	0.66%
2^{24}	255 KQPS	0.94 ms, 86.27 ms	2.76%
2^{25}	180 KQPS	1.07 ms, 250.52 ms	4.05%

Table 3: **Impact on migration when tuning group sizes.** *"Client BW" means the extra bandwidth usage between the client and servers.*

half of extra `READ` query packets in the client-side and there is no extra communication between servers. NetMigrate's extra bandwidth usage comes from both the client and server sides, but they are both negligible as shown in the results. The client-side extra usage comes from PriorityPull query retries as well as double-reads for undecidable conditions in the switch indexing; the server-side extra usage is from transferring late dirty logs to the destination as Source protocol. As shown in Table 4, Source protocol's extra bandwidth usage from the server side is proportional to the workload write ratio. Fulva's extra bandwidth usage all exceeds 35% and is several times higher than Rocksteady's. Rocksteady's and Fulva's client-side extra usages decrease with the write ratio increases because `WRITE` queries are directly served by the destination. NetMigrate achieves negligible extra bandwidth usage from both the client side and the server side. NetMigrate's client-

Write Ratio	0%						10%						20%						30%			
Source CPU Budget	100%		70%		40%		100%		70%		40%		100%		70%		40%		100%		70%		40%	
Extra Bandwidth Usage (%)	C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S
Source	0	0	0	0	0	0	0	10.5	0	10.2	0	9.8	0	20.7	0	20.6	0	20.5	0	31.3	0	30.9	0	30.6
Rocksteady	12.5	0	12.6	0	11.3	0	10.9	0	10.4	0	9.3	0	8.6	0	8.6	0	7.5	0	6.9	0	6.8	0	5.5	0
Fulva	58.1	0	60.9	0	50.3	0	49.4	0	55.0	0	46.6	0	43.6	0	58.2	0	53.4	0	36.9	0	39.2	0	44.1	0
NetMigrate (S $\times 10^{-5}$%)	0.025	3.11	0.031	3.39	0.05	2.65	0.005	3.27	0.005	2.69	0.005	1.05	0.004	1.36	0.004	3.74	0.004	2.37	0.003	1.71	0.003	2.88	0.003	2.1

Table 4: **Extra bandwidth usage between a client and servers and between two migration servers.** *'C' stands for Client-side extra bandwidth usage and 'S' stands for that of server-side. NetMigrate's server-side extra usage is at* 10^{-5}% *level.*



(a) 70% CPU budget, Throughput (b) 70% CPU budget, Median Latency (c) 70% CPU budget, 99%-tail Latency

(d) 40% CPU budget, Throughput (e) 40% CPU budget, Median Latency (f) 40% CPU budget, 99%-tail Latency

Figure 7: **Comparing Rocksteady, Fulva, Source-based protocol, and NetMigrate on throughput, median latency and 99%-tail latency with YCSB-B workload (5% write ratio), 70% and 40% source CPU budget.**

Write Ratio	0%			10%			20%			30%		
Source CPU Budget	100%	70%	40%	100%	70%	40%	100%	70%	40%	100%	70%	40%
Source	445.66	289.16	152.30	417.98	289.17	145.96	424.72	266.27	140.38	412.77	266.51	142.25
Rocksteady	113.17	114.52	67.73	119.11	125.15	77.53	133.76	132.34	84.07	147.79	145.57	99.28
Fulva	239.68	251.77	201.73	262.27	278.99	243.49	244.15	296.81	253.36	241.86	247.76	261.79
NetMigrate	**584.41**	**459.58**	**297.36**	**554.67**	**446.83**	**290.83**	**549.70**	**427.67**	**286.42**	**559.92**	**420.09**	**278.75**

Table 5: **Throughput under varied write ratios and source Redis CPU budgets.**

side extra usage indicates that double-reads issued by the switch and PriorityPulls from the destination happen less than 0.05%. Also, NetMigrate's server-side extra usage is less than 4×10^{-5} and negligible. Therefore, NetMigrate puts much less overhead to both the clients and the servers than other three baselines.

6.5 More Scenarios and Workloads

In this section, we evaluate more load-balancing scenarios and write-sensitive workloads by tuning the source Redis CPU limits and write ratios in the YCSB benchmark.

Load balancing scenarios. In a case that needs load balancing, the source node is usually overloaded and the destination node serves queries faster than the source. We mimic different overload levels by limiting the source Redis CPU to 70% and 40%. In this experiment, we configure NetMigrate to be throughput-optimized. Fig. 7 shows the throughput and latency comparisons among four migration protocols. Fig. 7 (a) and (d) show the throughput results. A lower source CPU budget for migration leads to a longer migration time for all protocols. NetMigrate's average throughput during migration are the highest for both 70% and 40% source CPU limitations. NetMigrate improves the throughput from 63% to 286% with 70% CPU limitation and from 29% to 305% with 40% CPU limitation. For Rocksteady, the less the CPU budget is given, the more slowly the throughput increases from the nearly zero QPS. For NetMigrate and Fulva, at the beginning of migration, the throughput improvement curves are similar, because both are limited by the migration speed. After that, NetMigrate is better than Fulva because our client is not bounded by the client-side packet rate. Source baseline keeps a low but stable throughput. Fig. 7 (b), (c), (e), and (f) show the latency results. For both 70% and 40% CPU budgets, median latency of NetMigrate and Source baseline remains low and stable, while Rocksteady's and Fulva's latency suddenly increases and gradually drops during migration. NetMigrate's median latency remains the lowest compared to other baselines. It reduces the median latency from 8% to 65% with 70% CPU limitation and from 32% to 97% with 40% CPU limitation. In terms of 99%-tail latency, Rocksteady is two orders of magnitude higher than other migration protocols during the entire migration due to its on-demand data fetching. However, it quickly falls to normal tail latency when migration finishes. Both NetMigrate's and Fulva's tail latencies drop gradually

Write Ratio	0%			10%			20%			30%		
Source CPU Budget	100%	70%	40%	100%	70%	40%	100%	70%	40%	100%	70%	40%
Source	3.38	**1.09**	1.40	3.65	1.23	2.38	3.55	1.31	1.33	3.62	1.21	1.28
Rocksteady	3.43	3.15	2.97	3.30	3.14	2.45	2.83	2.70	2.19	2.81	2.51	1.92
Fulva	3.11	2.07	1.93	2.58	2.10	1.80	2.72	2.09	1.80	2.79	2.38	1.82
NetMigrate	**2.12**	1.20	**1.10**	**2.26**	**1.11**	**1.05**	**2.27**	**1.05**	**1.08**	**2.20**	**0.97**	**1.05**

Table 6: **Median latency under varied write ratios and source Redis CPU budgets.**

Write Ratio	0%			10%			20%			30%		
Source CPU Budget	100%	70%	40%	100%	70%	40%	100%	70%	40%	100%	70%	40%
Source	**6.40**	29.32	62.11	**6.83**	30.05	66.01	**7.19**	31.03	68.22	8.11	30.31	63.29
Rocksteady	491.86	504.92	864.89	368.89	425.64	973.53	331.37	346.02	866.59	213.79	227.03	795.49
Fulva	21.75	23.34	48.75	21.04	24.59	42.36	18.51	23.42	46.06	21.50	22.65	38.79
NetMigrate	9.21	**23.09**	**48.31**	7.73	**21.74**	**33.89**	8.38	**19.19**	**41.28**	**7.50**	**15.89**	**34.75**

Table 7: **99%-tail latency under varied write ratios and source Redis CPU budgets.**

from the high latency level before migration while the tail latency of Source protocol remains the same as before migration until it is approaching migration completion. Overall, NetMigrate reduces 99%-tail latency from 18% to 56% with 70% CPU limitation and up to 94% with 40% CPU limitation.

Diverse write ratios. Changing the YCSB workloads among different write ratios (0%, 10%, 20%, and 30%), Table 5, 6, and 7 show that NetMigrate can achieve the highest throughput (improved from 6.5% to 416%) while maintaining the lowest latency as Source baseline. Rocksteady's 99%-tail latency is also much higher than other migration protocols when write ratios and source CPU budgets change.

7 Discussion and Related Work

Migration speed. In our experiments, migration time is limited by exporting key-values out from Redis and then sending through UDP socket. Key-value stores utilizing RDMA or other kernel-by-passing transmission (e.g., Intel DPDK [4], MICA [40], KV-Direct [39]) can increase migration speed by a lot. Despite kernel-bypassing, migration time remains non-negligible (e.g., 60 sec for 200GB data, 40Gbps links [37]). Migration degrades query performance significantly and migration happens fairly frequently in the storage clusters. NetMigrate can work with faster networking to improve the KV serving performance during migration.

Fault tolerance is also critical during migration, including server failures and switch failures. To handle server failures, enabling logs on both source and destination key-value storage servers is a viable solution. Recovery is achieved by merging logs from both sides to attain the latest version. RedPlane [35] and ExoPlane [36] provide fault-tolerant solutions for switch failures and resource augmentation.

Strong/weak data consistency. UDP-based protocol can have packet loss and out-of-order transmission, which weakens the data consistency. We can add a reliable transmission mechanism to our UDP-based migration protocol, and thus it can be robust to give a strong data consistency over network transmission. The migration control packet replies are generated by the switch and sent back to the source node. This avoids duplicated updates in the switch index structures. Also, when NetMigrate merges data insertion from migration and write queries at the destination, it needs the key-value store's version numbers to guarantee strong consistency. In practice, many key-value stores provide weak consistency and sacrifice consistency for availability and performance [12, 19]. NetMigrate is compatible with weak data consistency.

Key and value sizes and multi-key operations. We use 4-Byte keys and 64-Byte values in the prototyping experiments but NetMigrate can be extended to larger key and value lengths as long as a group id and a single key can be fit into switch metadata (128 bits at most). If the key size is relatively small, we can also extend the packet format to support multi-key operations in one packet and recirculate one query packet in the switch and treat each pass as serving a single-key query.

Clearing bloom filters in practice. When a shard of data migration information is updated to the storage cluster indexing proxy, the cluster scheduler can pause migrations for a bit (for synchronization) and clean the probabilistic indexing data structures shared in the switch periodically.

Migration plan. There are works generating reconfiguration plan based on cluster load status, migration time, performance impact and so on [18, 21, 41, 43, 44, 47, 48, 50], while NetMigrate focuses on live migration technique.

8 Conclusions

We present NetMigrate, a new live migration approach for in-memory key-value stores based on programmable data planes. NetMigrate migrates shards between nodes with zero service interruption and minimal performance impact using switches for migration status tracking. Extensive experimental results demonstrate the ability of NetMigrate to provide enhanced throughput and maintain low access latency under a variety of changing workloads and scenarios during migration.

Acknowledgments. We would like to thank the anonymous reviewers and our shepherd Jiri Schindler for their helpful comments. We thank Zhuolong Yu for help with the testbed setup and debugging. This work was supported in part by NSF grants CNS-2107086, SaTC-2132643, CNS-2106946.

References

[1] Apache Cassandra. https://cassandra.apache.org.

[2] Building a Gigascale ML Feature Store with Redis, Binary Serialization, String Hashing, and Compression. https://doordash.engineering/2020/11/19/building-a-gigascale-ml-feature-store-with-redis/.

[3] CPU limit tool. https://github.com/opsengine/cpulimit.

[4] DPDK. https://www.dpdk.org/.

[5] EX9200 Line of Ethernet Switches. https://www.juniper.net/us/en/products/switches/ex-series/ex9200-programmable-network-switch.html.

[6] gRPC. https://grpc.io/.

[7] Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[8] Memcached. https://memcached.org.

[9] NetMigrate key-value store live migration codebase. https://github.com/Froot-NetSys/NetMigrate.

[10] P4 Language. https://opennetworking.org/p4/.

[11] Redis. https://redis.io.

[12] Redis Cluster consistency guarantees. https://redis.io/docs/management/scaling/.

[13] Redis MIGRATE command. https://redis.io/commands/migrate/.

[14] Redis-plus-plus library. https://github.com/sewenew/redis-plus-plus.

[15] Redis use cases. https://redis.com/blog/5-industry-use-cases-for-redis-developers/.

[16] Trident 5 / BCM78800 Series. https://www.broadcom.com/products/ethernetconnectivity/switching/strataxgs/bcm78800.

[17] YCSB Benchmark written in C++. https://github.com/ls4154/YCSB-cpp.

[18] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 739–753, USA, 2016. USENIX Association.

[19] Eric Anderson, Xiaozhou Li, Mehul A Shah, Joseph Tucek, and Jay J Wylie. What consistency does your {Key-Value} store actually provide? In *Sixth Workshop on Hot Topics in System Dependability (HotDep 10)*, 2010.

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

[21] Saurabh Bagchi, Somali Chaterji, Paul Curtis Wood, and Ashraf Mahgoub. Clustered database reconfiguration system for time-varying workloads, September 6 2022. US Patent 11,436,207.

[22] Sean Barker, Yun Chi, Hakan Hacigümüs, Prashant Shenoy, and Emmanuel Cecchet. {ShuttleDB}:{Database-Aware} elasticity in the cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 33–43, 2014.

[23] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. " cut me some slack" latency-aware live migration for databases. In *Proceedings of the 15th international conference on extending database technology*, pages 432–443, 2012.

[24] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[25] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[26] 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*, pages 143–154, 2010.

[27] 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. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[28] Aaron J Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 299–313, 2015.

[29] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293, 2000.

[30] Jiewen Hai, Cheng Wang, Xusheng Chen, Tsz On Li,

Heming Cui, and Sen Wang. Fulva: Efficient live migration for in-memory key-value stores with zero downtime. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 83–8309. IEEE, 2019.

[31] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[32] 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*, pages 121–136, 2017.

[33] Junbin Kang, Le Cai, Feifei Li, Xingxuan Zhou, Wei Cao, Songlu Cai, and Daming Shao. Remus: Efficient live migration for distributed databases with snapshot isolation. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2232–2245, 2022.

[34] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: Locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 145–161, 2021.

[35] Daehyeok Kim, Jacob Nelson, Dan RK Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 223–244, 2021.

[36] Daehyeok Kim, Vyas Sekar, and Srinivasan Seshan. {ExoPlane}: An operating system for {On-Rack} switch resource augmentation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1257–1272, 2023.

[37] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 390–405, 2017.

[38] Herman Lee Kwiatkowski, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, and David Stafford. Scaling memcache at facebook.

[39] 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*, pages 137–152, 2017.

[40] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. {MICA}: A holistic approach to fast {In-Memory}{Key-Value} storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[41] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. Mgcrab: transaction crabbing for live migration in deterministic database systems. *Proceedings of the VLDB Endowment*, 12(5):597–610, 2019.

[42] 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, 2019.

[43] Chenyang Lu. Aqueduct: Online data migration with performance guarantees. In *Conference on File and Storage Technologies (FAST 02)*, 2002.

[44] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. SOPHIA: Online reconfiguration of clustered NoSQL databases for Time-Varying workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 223–240, Renton, WA, July 2019. USENIX Association.

[45] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[46] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[47] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046, 2014.

[48] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10(4):445–456, nov 2016.

[49] Ravindra Kumar Singh and Harsh Kumar Verma. Redis-based messaging queue and cache-enabled parallel processing social media analytics framework. *The Computer Journal*, 65(4):843–857, 2022.

[50] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained

elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, nov 2014.

[51] Bo Wang, Changhai Wang, Ying Song, Jie Cao, Xiao Cui, and Ling Zhang. A survey and taxonomy on workload scheduling and resource provisioning in hybrid clouds. *Cluster Computing*, 23:2809–2834, 2020.

[52] Ke Wang, Xraobing Zhou, Tonglin Li, Dongfang Zhao, Michael Lang, and Ioan Raicu. Optimizing load balancing and data-locality with data-aware scheduling. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 119–128. IEEE, 2014.

[53] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 335–347, 2017.

A NetMigrate Network Protocol

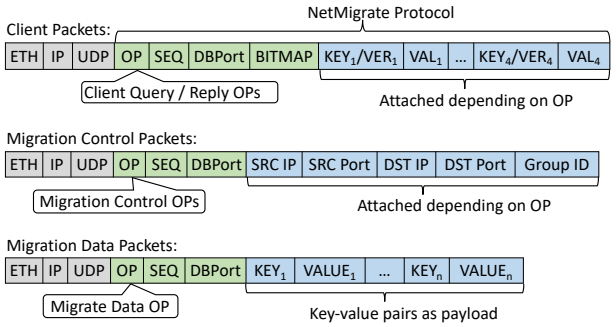


Figure 8: **NetMigrate packet format for migration packets and client query/reply packets.**

Packet format. Fig. 8 shows the packet format in NetMigrate protocol. NetMigrate is an application-layer protocol inside L4 payload. A set of dedicated UDP ports are reserved for key-value storage server/client agents. In the switch, these ports indicate the packets are NetMigrate migration packets, client query, or reply packets, to invoke the custom packet processing logic. The NetMigrate header fields are `OP`, `SEQ`, and `DBPort`. `OP` fields represents client query or reply operators, or migration-related operators. `SEQ` can be used as a sequence number for reliable transmissions with UDP protocol. `DBPort` refers to which key-value store instance the packet is responsible for, filled with the application port. For packets deal with client queries and data migration, they have `KEY`, `VALUE`, or `VERSION` fields. `KEY` and `VALUE` carry the key and value of a key-value pair and `VERSION` indicates the version number in reply packets for data consistency guarantees during migration. NetMigrate supports `GET`, `SET`, and `DELETE` client query types and can be extended to other type of queries. `GET` and `DELETE` query operators only have `KEY` fields; `SET` packets have both `KEY` and `VALUE` fields; and all reply packets has `VERSION` fields indicating whether the operation is successful and the reply is from the source or the destination storage instance. The header fields for migration control packets are `OP`, and migration instance `(SRC_IP, SRC_Port, DST_IP, DST_Port)` or `group_id` which is attached depending on the `OP`. `OP` can be `MIGRATE_INIT`, `MIGRATE_TERMINATE`, `MIGRATE_GROUP_START`, `MIGRATE_GROUP_COMPLETE`, `MIGRATE_DATA`, and their corresponding reply operators. `MIGRATE_INIT` and `MIGRATE_TERMINATE` packets have fields indicating the source and destination key-value store migration instance, filled with the server IP address and transport layer port pair `(SRC_IP, SRC_Port, DST_IP, DST_Port)`. `MIGRATE_GROUP_START` and `MIGRATE_GROUP_COMPLETE` packets notify switch that a migration group from DBPort instance has started migration or has completed migration, to update migration status tracking indexing in the switch. `MIGRATE_DATA` packets simply carry the key-value pairs in

the packets and are transferred from the source server to the destination server.

Network Routing. NetMigrate leverages existing routing protocols to forward packets. For migration control and migration data packets, they are routed as normal packets from the source server to the destination server, in addition to updating indexing data structure in the switch. NetMigrate switches are placed on the path from the clients to the storage clusters. Client query and reply packets are forwarded based on indexing look-up results to determine the "right" storage server as described in § 4.3.

Merging read replies with version control. NetMigrate has an 8-bit version control field in reply packets, identifying: (1) whether the query is successfully executed in the backend storage server, (2) the reply packet is from the source server or the destination server, (3) whether the reply is from double-read, and (4) whether the query needs PriorityPull. Two more bits are reserved for more controls. To handle double-reads, the client agent merges two replies received from the source and the destination to the one with a newer version.

B Artifact Appendix

Abstract

NetMigrate is a key-value store live migration protocol by leveraging programmalbe switches. NetMigrate migrates KVS shards between nodes with zero service interruption and minimal performance impact. During migration, the switch data plane monitors the migration process in a fine-grained manner and directs client queries to the right server in real time.

Our artifact provides code and scripts to reproduce experimental results in the paper, especially in replicating Figures 4-7. We demonstrated experimental results on three commodity machines and a Barefoot Tofino switch.

Scope

The artifact can be used as a prototype of NetMigrate migration protocol with the backend KVS as Redis, and to validate the experimental results on performance improvement compared with other migration baselines in the paper.

Contents

The artifact contains four migration protocols' server agents in `cpp/server` folder, YCSB client implementation for four migration protocols in `cpp/YCSB-client`, and switch data-plane and control-plane code for NetMigrate in `tna_kv_migration`, with experiment steps in `README.md` and `experiment_steps` folder.

Hosting

The artifact is hosted on GitHub (https://github.com/Froot-NetSys/NetMigrate, main branch, commit c977bfa2c8eeec7d77fb4a834cebfb1e3f819e24).

Requirements

We developed and tested the artifact on the below platform:

- **Hardware:** A Barefoot Tofino switch, and three servers each with a NIC (we used an Intel XL710 for 40GbE QSFP+) and multi-core CPU, connected by the Tofino switch.
- **Software:** Tofino SDK (version 9.4.0) on the switch, Python2.7 on the switch, and gRPC 1.50.0 and protobuf 3.21.6.0 for PriorityPulls in KV servers.