BESPOKV: Application Tailored Scale-Out Key-Value Stores

Ali Anwar*, Yue Cheng†, Hai Huang‡, Jingoo Han§, Hyogi Sim¶, Dongyoon Lee∥, Fred Douglis‡, Ali R. Butt¶

*IBM Research–Almaden, †George Mason University, ‡IBM Research–T.J. Watson,
§Virginia Tech, ¶Oak Ridge National Laboratory, ∥Perspecta Labs

*ali.anwar2@ibm.com, †yuecheng@gmu.edu, ‡haih@us.ibm.com, §jingoo@vt.edu, ¶simh@ornl.gov
∥dongyoon@cs.vt.edu, ‡fred.douglis@gmail.com, ¶butta@cs.vt.edu

Abstract—Enterprise KV stores are not well suited for HPC applications, and entail customization and cumbersome end-to-end KV design to extract the HPC application needs. To this end, in this paper we present BESPOKV, an adaptive, extensible, and scale-out KV store framework. BESPOKV decouples the KV store design into the control plane for distributed management and the data plane for local data store. BESPOKV takes as input a single-server KV store, called a datalet, and transparently enables a scalable and fault-tolerant distributed KV store service. The resulting distributed stores are also adaptive to consistency or topology requirement changes and can be easily extended for new types of services. Experiments show that BESPOKV-enabled distributed KV stores scale horizontally to a large number of nodes, and performs comparatively and sometimes better than the state-of-the-art systems.

I. INTRODUCTION

The underlying storage and I/O fabric of modern high performance computing (HPC) increasingly employ new technologies such as flash-based systems and non-volatile memory (NVM). While improving I/O performance, e.g., via providing more efficient and fast I/O burst buffer, such technologies also provide for opportunities to explore the use of in-memory storage such as key-value (KV) stores in the HPC setting. Distributed KV stores are beginning to play an increasingly critical role in supporting today’s HPC applications. Examples of this use include dynamic consistency control [1], coupling applications [2], [3], and storing intermediate results [4], among others. Relatively simple data schemas and indexing enable KV stores to achieve high performance and high scalability, and allow them to serve as a cache for quickly answering various queries, where user experience satisfaction often determines the success of the applications. Consequently, a variety of distributed KV stores have been developed, mainly in two forms: natively-distributed and proxy-based KV stores.

The natively-distributed KV stores [5], [6], [7], [8], [9] are designed with distributed services (e.g., topology, consistency, replication, and fault tolerance) in mind from the beginning, and are often specialized for one specific setting. For example, HyperDex [10] supports Master-Slave topology and Strong Consistency (MS+SC). Facebook relies on its own distributed Memcache [8] with Master-Slave topology and Eventual Consistency (MS+EC). Amazon employs Dynamo [6] with Active-Active1 topology and Eventual Consistency (AA+EC).

1Active-Active is also called multi-master in database literature.

Another key limitation of natively-distributed KV stores lies in their inflexible monolithic design where distributed features are deeply baked with backend data stores. The rigid design implies that these KV stores are not adaptive to ever-changing user demands for different backend, topology, consistency, or other services. For instance, Social Artisan [11] and Be-hance [12] moved from MongoDB to Cassandra for scalability and maintenance reasons [13]. Conversely, Flowdock [14] migrated from Cassandra to MongoDB due to stability issues. Unfortunately, this migration process is very frustrating and time/money-consuming as requires data remodeling and extra migration resources [13].

Alternatively, proxy-based distributed KV stores leverage a proxy layer to add distributed services into existing backend data stores. For example, Mcrouter [15], and Twemproxy [16] can be used as a proxy to enable a basic form of distributed Memcached [17] with partitioning, as shown in Figure 1(b). Twemproxy supports additional Redis [18] backend as well. Recently, Netlix Dynomite [19] extended Twemproxy to support high availability and cross-datacenter replication, as illustrated in Figure 1(c).

Unlike monolithic natively-distributed KV stores, the use of a separate proxy layer enables support for multiple backends. Each single-server KV store such as Memcached [17], Redis [18], LevelDB [20], and Masstree [21] has own its merit, so the ability to choose one or mix is an ample reward. However, existing proxy-based KV stores are still limited to a single topology and consistency: e.g., Dynomite supports

Fig. 1: Different approaches to enable distributed KV stores: (a) natively-distributed (b-d) proxy-based.
We demonstrate that BESPOKV-enabled distributed KV stores in a local testbed as well as in a public cloud (Google Cloud Platform [23]) and evaluate their performance.

II. MOTIVATION: KEY-VALUE STORES FOR HPC

This section discusses the benefits of using distributed KV stores for HPC applications, and how BESPOKV can bring the potential into reality.

Wang et al. [2] first studied the effectiveness and usefulness of distributed KV stores for HPC. By encapsulating the distributed system complexities into the KV stores, the authors showed how KV stores can simplify design and implementation of HPC services such as Job launch, monitoring/logging, and I/O forwarding. In response, HPC community has proposed different HPC-oriented KV stores such as SKV [4], PapyrusKV [1], MDHIM [24], and Sharp [3]; and further demonstrated other use cases of KV stores for online analysis, visualization, and coupling HPC applications.

Non HPC-oriented KV stores are mostly built for streaming workloads, they often use a lot of memory out of the box. This is not very friendly to HPC workloads. Matching of application as well as KV store memory demands, and throughput/latency needs require customizations that existing KV stores do not provide. BESPOKV’s control and data plane decoupled architecture, flexible configurability, and extensibility enable new solutions and such customizations for emerging HPC systems and workloads. First, BESPOKV makes it easy for HPC developers to explore different design trade-offs in future HPC systems with heterogeneous hardware resources. Prior solutions are developed for one architecture. For instance, SKV [4] is designed for the IBM Blue Gene Active Storage I/O nodes equipped with flash storage [25], while PapyrusKV [1] is designed to leverage non-volatile memory (NVM) in HPC systems. Future HPC architectures are expected to have hierarchical, heterogeneous resources such as DRAM, NVM, and high-bandwidth memory (HBM). BESPOKV seamlessly support the use of different datalets, each of which can be tuned, for different memory and storage architecture (evaluated in section VI-A).

Second, BESPOKV enables new HPC services for emerging workloads. (1) **Data layout**: While existing KV solution are rigid and pre-fixed for one setting, BESPOKV allows storing data in different datalets, adapt and switch datalets as needed, and thus can handle diverse characteristics of new data workloads. For example, datalet using B-tree as main data structure is better suited for read-intensive workloads [26] (e.g., deep learning), while LSM tree based datalet is a better choice for write-intensive workloads due to high write amplification and no fragmentation [27]. Unlike existing solutions BESPOKV provides option to switch datalets (evaluated in sections VI-A and VIII-B). (2) **Multi-tenancy and geo-distribution**: HPC applications built atop KV store may require dynamically switching the topology and consistency. For example, in case of a distributed metadata server or job launch system built using KV store [2], simple MS topology may be sufficient for handling metadata and resource contention for Jobs launched but...
on one cluster but AA topology may become more beneficial as we scale out to multiple clusters located at different geolocations. Existing KV stores do not provide such support (evaluated in section VIII-C). (3) Low latency: Ultra low latency is often required to take advantage of in-memory KV storage [28]. For this purpose we added support for DPDK [29] kernel bypassing in BESPOKV (evaluated in §E).

III. BESPOKV DESIGN

In this section, we describe the design of BESPOKV and how it provides compatibility, versatility, modularity, and high performance for supporting distributed KV stores. Figure 2 shows the overall architecture of BESPOKV comprising five modules: datalet, controlet, coordinator, client library, and optional components. A collection of datalets form the data plane, the rest of the modules makes up the control plane.

**Datalet** is supplied by the user and responsible for storing data within a single node. Datalet should provide the basic I/O interfaces (e.g., Put and Get) for the KV stores to be implemented. We refer to this interface as the datalet API. For example, a user can develop a simple form of in-memory hash table. Users can also mix and match datalets with each datalet using a different data structure.

**Controlet** is supplied by BESPOKV and provides a datalet with distributed management services to realize and enable the distributed KV stores associated with the datalet. The controlet processes client requests and routes the requests to the associated entities: e.g., to a datalet for storing data. BESPOKV provides a default set of controlets, and allows advanced users to extend and design new controlets as needed for realizing a service that may require specialized handling in the controlet.

BESPOKV allows an arbitrary mapping between a controlet and a datalet. A controlet may handle \( N \geq 1 \) instances of datalets, depending on the processing capacity of the controlet and its datalets, and can leverage physical resource (datacenter) heterogeneity [30], [31] for better overall utilization. For instance, a controlet running on a high-capacity node may manage more datalet nodes than a controlet running on a low-capacity node. For simplicity, we use one-to-one controlet-datalet mappings in the rest of the paper.

**Coordinator** provides three main functions. (1) It maintains the metadata regarding the whole cluster topology and provides a query service as a metadata server. (2) It tracks the liveness of the cluster by exchanging periodic heartbeat messages with the controlets. (3) It coordinates failover in case of a node failure. The coordinator can run on separate node or alongside other controlets.

BESPOKV implements the coordinator on top of ZooKeeper [32] for better resilience. Similar to designing specialized controlets, advanced users have the option to design customized coordinators if needed. It is also possible to design a new coordinator as a special form of controlet from scratch using the BESPOKV-provided controlet programming abstraction as shown in section III-B. Nonetheless, because it is widely used across many KV stores, BESPOKV includes the coordinator as a default module in the control plane.

**Client library** is provided by BESPOKV and used by the client applications to utilize the services created by BESPOKV. The library provides a flexible means for mapping data to controlets. The client application uses the library interface to consult with the coordinator and fetch data partitioning and mapping information, which is then used to route requests to appropriate controlets. BESPOKV allows different developers to choose their own partitioning techniques such as consistent hashing and range-based partitioning.

**Optional Components** BESPOKV provides two optional components facilitating the controlet development: 1) a distributed lock manager (DLM) for a locking service, and 2) a Shared Log for an ordering service. One can build such a distributed management service as a special form of controlets from the scratch, but given its common use in distributed KV store development, BESPOKV imports existing solutions (e.g., Redlock [33] for DLM, and ZLog [34], [35], [36] for Shared Log) and provides interface libraries (section III-B, Table III).

A. Data Plane

A collection of datalets running on different distributed nodes form the data plane for BESPOKV. A single-server

---

**TABLE II: APIs to Put, Get, and Del a KV pair. Datalet and Client APIs are for using pre-built controlets.**

<table>
<thead>
<tr>
<th>Client API (provided by application developers)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put(key, val)</td>
<td>Write the (key,val) pair to the datalet</td>
</tr>
<tr>
<td>val=Get(key)</td>
<td>Read val of key from the datalet</td>
</tr>
<tr>
<td>Del(key)</td>
<td>Delete (key,val) pair from the datalet</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Datalet API (provided by application developers)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateTable(T)</td>
<td>Create a table T to insert data</td>
</tr>
<tr>
<td>Put(key, val, T)</td>
<td>Write the (key,val) pair to table T</td>
</tr>
<tr>
<td>val=Get(key, T)</td>
<td>Read val of key from table T</td>
</tr>
<tr>
<td>Del(key, T)</td>
<td>Delete (key,val) pair from table T</td>
</tr>
<tr>
<td>DeleteTable(T)</td>
<td>Delete table T</td>
</tr>
</tbody>
</table>
datalet is completely unaware of other datalets. **Datalet Development.** BESPOKV supports multiple backends. Users can make use of off-the-shelf single-serve data stores such as Redis [18], SSDB [37], and Masstree [21]. In addition, BESPOKV provides datalet templates based on commonly used data structures: currently, a hash-table-based $tHT$, a log-based $tLog$, and a tree-based $tMT$. For the ease of development, BESPOKV furnishes an asynchronous event-driven network programming framework in which developers can design new datalets, starting from existing templates. We evaluate the reduced engineering effort in §VII.

**APIs and Protocol Parsers.** For compatibility and modularity, BESPOKV provides a clean set of datalet APIs (between controlet and datalet) and client APIs (between client app and client library). Table II presents example datalet and client APIs. As these APIs are consistent with existing I/O interfaces of existing KV stores. Datalet developers can adopt them in a straightforward manner to enable distributed services. This is much easier than library-based replication solutions such as Vsync [38] where developers should learn complex new APIs.

To offer compatibility and be able to understand application protocols to process incoming requests properly, BESPOKV’s communication substrate supports two options. (1) It provides a BESPOKV-defined protocol using Google Protocol Buffers [39]. This option is suitable for new datalets and is preferred due to its ease of use and better programmability. (2) BESPOKV allows developers to provide a parser for their own protocols. This option is mainly available for porting existing datalets such as Redis or SSDB.

**Control Plane.** BESPOKV provides a set of pre-built controlets that provide datalets with common distributed management. Given a datalet, BESPOKV makes distributed KV stores immediately ready-to-use. Developers can also extend these pre-built controlets or design new ones from scratch for advanced services. **Pre-built Controlets.** BESPOKV identifies four core components for distributed management, and provides pre-build controlets that support common design options in existing distributed KV stores. The choice is based on our comprehensive study of existing systems that revealed three key observations: (1) cluster topology, consistency model, replication, and fault tolerance generally define distributed features of KV stores; (2) for the topology, MS and AA are common; and (3) for the consistency model, SC and EC are popular. Detailed descriptions of exemplary controlets supporting MS+SC, MS+EC, AA+SC, and AA+EC options follow in §IV.

**Controlet Development.** To support advanced users and new kinds of services, BESPOKV provides an asynchronous event-driven network programming framework for controlet development as well. For each event (e.g., Put request, timeout, etc.), developer can define event handlers to instruct how the controlet should process the event to enable versatile distributed management services in the control plane. The aforementioned pre-built controlets indeed consist of a set of pre-defined event handlers for common distributed services.

**Control Plane Configuration.** To configure the system, each controlet takes as input (1) a JSON configuration file that specifies the basic system deployment parameters such as topology, consistency model, the number of replicas, and coordinator address; and (2) a datalet host file containing the list of datalets to be managed. BESPOKV loads the runtime configuration information at the coordinator, which serves as the query point for the client library and controlets to periodically retrieve configuration updates. Any change in configuration at runtime (e.g., topology/consistency switch) results in replacing old controlets with new ones. We describe dynamic adaptation mechanisms in §V in detail. **Controllet Programming Abstraction** BESPOKV uses asynchronous event-driven programming model to achieve high throughput. For each event (e.g., incoming network input, timer, etc.), developers are asked to define event handlers to process the event. There are two types of events in BESPOKV: basic and extended events. Basic events represent pre-defined conditions. Developers can create their own extended events by using basic or existing extended events.

**Other Controlet APIs.** BESPOKV provides a set of libraries and APIs with common features for controlet development, shown in Table III.

### IV. BESPOKV-Based Distributed KV Stores

BESPOKV, to be specific its control plane, transparently turns a user-provided single-server datalet to a scalable, fault-tolerant distributed KV store. This section presents support for MS+SC (See §C for MS+EC, AA+SC, and AA+EC) and four examples to enable new forms of distributed services by combining existing controlets or extending ones.

---

**TABLE III: APIs for Events, Shared Log, DLM, and Coordinator for new controlet development.** Due to space limitation, we list only important APIs.

- **Event API** (provided by BESPOKV)
  - Register(c, e, cb): Register basic event e for conn c to call func cb
  - Enable(c, e): Enable event e to be triggered on conf conn c
  - Get(c, cb): Register extended event e to call func cb
  - Emit(e): Emit event e

- **Shared Log API** (provided by BESPOKV)
  - CreateLog: Creates a new log instance
  - PutSharedLog(m, L): Append message m to log L
  - AsyncFetch(L): Asynchronous read from log L

- **DLM API** (provided by BESPOKV)
  - Lock(key): Acquire lock on key
  - Unlock(key): Unlock key

- **Coordinator API** (provided by BESPOKV)
  - LogHeartbeat(c, d): Log heartbeat for controlet c & datalet d
  - GetSharedList(s): Get controlet & datalet list for shard s
  - ElectLeader(s): Elect new Master controlet for shard s

**Fig. 3:** Put/Get paths in MS+SC. $\theta n$ means the $n^{th}$ slave; $D$ means datalet.
A. Master-Slave & Strong Consistency

We start from a KV store supporting the MS topology with the SC model (MS+SC). Perhaps the simplest way to ensure SC is to rely on a locking mechanism using ZooKeeper [32] at the cost of serialization. However, alternative scalable designs exist such as chain replication (CR) [40], value-dependent chaining [10], and their variants. The pre-built BESPOKV controlet for MS+SC leverages CR algorithm. Our modular design allows BESPOKV to adopt other optimizations for CR [41], [42] as well, but so far we have not implemented those. The original CR paper describes the tail sending a message directly back to the client; but similar to CRAQ [42], our implementation lets the head respond after it receives an acknowledgment from the tail, given its pre-existing network connection with the client.

Example. Figure 3 shows how MS+SC is implemented in BESPOKV. Here, clients route Puts to the head of the corresponding controlet–datalet chains via consistent hashing (step 1). The head controlet forwards the incoming Put request to its local datalet (step 2) and then to mid node (step 3), which forwards the request to its local datalet and then to tail (step 4). Tail first forwards the request to local datalet and then sends Ack back to mid, which sends Ack back to head (step 5). Once the head controlet receives the Ack from the mid, the head controlet marks the request completed and responds to the client (step 6). Gets are routed to the tail node of the corresponding chains. This provides the SC guarantee as clients are only notified of the successful completions of Puts after the data is persisted through the tail nodes.

Failover. In all cases (MS+SC, MS+EC, AA+SC, and AA+EC), when the coordinator detects a node failure using a periodic heartbeat message, it launches a new controlet–datalet pair in recovery mode on one of the standby nodes. The new controlet then recovers the data from one of the datalets.

In particular, for MS+SC using chain replication, the coordinator performs the chain recovery process and adds the new pair as the new tail to the end of the chain. The former chain recovery process depends on the location of the failure in the replica chain as follows. If a middle node fails, the coordinator notifies the head controlet to skip forwarding requests to the failed node. In case the tail node fails, the coordinator informs the head controlet to skip forwarding requests to the tail datalet and temporarily marks the second to the last node as the new tail so that future incoming Get requests can be redirected properly. If the head node fails, the coordinator appoints the second node in chain as the new head, and updates the cluster metadata. Upon seeing the change, the clients redirect future writes to the new head. Every node maintains a list of requests received but not yet processed by the tail, which is used to resolve in-flight requests [42], [40].

B. Range Query

We support range query or scan operations as follows. For datalets, the Masstree-based iMT template is used and extended to expose a range query API such as GetRange(Start, End). The client library supports range-based partitioning, e.g., dividing the name space by alphabetical order (e.g., A-C on one node, D-F on another node, and so on). The controlet divides a client request into sub-requests and forwards the sub-range query requests to corresponding datalets that store the specified range.

C. Per-request Consistency

We extend the client library GET API to support consistency/topology specification on a per-request basis. For instance, under MS+SC, if the user specifies a lower value of consistency level, GETs can go to any of the replicas, thus only eventual consistency is guaranteed.

D. Polyglot Persistence

A use case for KV store is to support businesses that may be divided into different components, and each component requires its own private data storage. BESPOKV supports such polyglot persistence [43] by launching custom controlets for cross-app lazy synchronization (eventual consistency).

E. Other Topologies

BESPOKV also supports an AA-MS hybrid topology by configuring an MS topology for each shard on top of the logical AA overlay. Similarly, a P2P-like topology can also be enabled by allowing clients to send a request to any controlet, which then routes the request to the actual controlet that manages the requested data. In this case, a controlet needs to maintain a routing map similar to a finger table [44] to determine the location of keys.

V. Dynamic Adaptation to Consistency and Topology Model Changes

Separating the control and data planes bring another benefit: BESPOKV-enabled distributed KV stores can seamlessly adapt to consistency and topology model changes at runtime by switching the controlets while keeping the datalets unchanged. At a high level, upon a consistency and/or topology change request, Coordinator launches a new set of controlets that will provide new services. Two old and new controlets are mapped to one datalet during the transition phase. The old controlet provides the old service with no downtime, and forwards some requests to the new controlet so that it can prepare the new service. When the transition completes, the new controlet takes over the old one. The transition protocol differs per each case. BESPOKV supports any transition between four aforementioned topology and consistency combinations, among which we describe two interesting cases in detail. Section VIII-C presents the experimental results on this aspect.

A. Transition from MS+EC to MS+SC

To make a transition from EC to SC, the master node needs to make sure that all the Put requests 1) that have arrived before the transition starts and 2) that arrive during transition are fully propagated to the slave nodes. For the former, the old master keeps flushing out any pending propagation. For the latter, the old master forwards an incoming Put request to the new master controlet which uses chain replication for SC, instead of propagating it asynchronously. When there is no more pending propagation left in the old controlet, the transition is over. SC guarantees will be enforced after
the transition has completed. During the transition, any node may respond to Get requests, providing EC guarantee. This means that a Get request, even after the reconfiguration was requested, may experience EC until the transition is over. As controlet developers are responsible for developing the transition functionality for the various consistency/topology modes, a controlet developer can choose an alternative route to fence all writes as soon as the reconfiguration is requested so that all reads observe the same and latest applied value.

Figure 4 (a) shows transition from MS+EC to MS+SC. Client 1 sends a Put request (Step 1a) to the old master controlet C1. A concurrent Get request (Step 1b) from Client 2 gets serviced as it used to be. The old master forwards Put request (Step 2) to the new master controlet which guarantees SC. When the new master completes its chain replication process, it acknowledges the old master, which in turn acknowledges Client 1. When the transition completes, a Put request (Step 3) is routed to the new master controlet.

**B. Transition from AA+EC to MS+EC**

In AA+EC, any active node can get a Put request. To maintain a global ordering between concurrent Puts, an active node relies on the Shared Log that propagates Puts to the other nodes on its behalf. On the other hand, in MS+EC, only the master node gets Put requests and is in charge of propagating them to the slaves. Therefore, the key operation in the transition from AA+EC to MS+EC is to move the role of propagating Puts from the Shared Log to the new master. To this end, when the transition starts, the new master node takes the in-flight Puts that have not been propagated yet from the Shared Log and starts propagating them by itself. When an old active controlet receives a Put request during transition, it does not consult with Shared Log, but forwards the request to the new master node which will eventually propagate the request. The Get requests are not affected. Figure 4 (b) shows an example where a Put request (Step 1) is forwarded to the new master (Step 2) during transition. When the transition completes, a Put request (Step 3) is serviced by the new master. The transition from MS+EC to AA+EC can be supported by the reverse step order.

**VI. BESPOKV’S USE CASES**

**A. Hierarchical and heterogeneous storage of HPC**

HPC big data problems require efficient and scalable storage systems, but load balancing I/O servers at scale remains a challenge. Statistical analysis [45] and Markov chain model [46] have been used to predict shared resource usage. A KV store can be used to collect runtime statistics from HPC storage systems for accurate prediction. However, existing KV stores are designed for one type of storage architecture (in-memory, SSD, NVM, etc.), leading to suboptimal performance.

**BESPOKV supports the use of different datalets to store replicas of a KV pair, where each of these datalet can be tuned for different memory and storage architecture. By doing so, BESPOKV unifies multiple data abstraction together and enables multifaceted view on shared data with configurable consistency and topology. Figure 5 shows an example of how BESPOKV unifies three different data abstractions – a log-structure merge-tree, Masstree, and log, and transparently provides master-slave topology (MS) and eventual consistency (EC). Data is replicated asynchronously in batch mode from master to slaves. In this design, it is possible to run applications with different properties (e.g., write-intensive and read-intensive apps) together.

There are two advantages of this design architecture. First, different applications can choose datalet that best suits their needs. As a typical use case, monitoring data collection is write-intensive workload, and prefers a scalable solution that is able to persist all data on persistent storage. Whereas, analytical models incur read intensive workload which could benefit from high read throughput. Second, replicas in different datalets are not evicted simultaneously. For instance, a replica of a KV pair may evict from in-memory based datalet due to size restriction but another replica may stay longer in NVM/SSD based datalet or stay forever in log based datalet that uses HDD.

To evaluate this scenario, we develop a monitoring system for Lustre parallel file system that collects monitoring data from different components of Lustre. The collected data is used by an analytics model to perform I/O load balancing for big data HPC applications. As monitoring workload is dominated by Put, we choose a log-structure merge-tree as our first datalet. Contrarily, analysis workload is read-intensive so we select Masstree as our second datalet. For persistence of data, we use log based datalet that stores on HDD.

The monitoring workload is obtained by running three HPC applications simultaneously on a production Lustre deployment. First application is Hardware Accelerated Cosmology Code (HACC), second is IOR benchmark, and third is a transaction processing application from a large financial institution. The workload consists of monitored data which includes system level stats from Metadata Server (MDS) and Object Storage Server (OSS), and overall metadata from
Object Storage Target (OST) and Metadata Target (MDT). The collected time series data is propagated as KV pairs to BESPOKV (deployed on 24 nodes setup) via the client side library integrated with probe agents. Analytics model captures two properties of HPC applications’ I/O requests namely, stripe count and number of bytes to be written. The predicted requests along with the current application requests are used to drive load balancer. The workload is completely read-intensive with uniform distribution.

Figure 6 shows the result of using different data abstractions for monitoring and analytics workloads. As shown, LSM outperforms B+ tree by 25% in terms of average throughput for write-intensive/monitoring workload. Whereas, the average throughput of B+ tree is 35% better compared to LSM for read-intensive/analytics workload. Furthermore, both B+ tree and LSM outperform log. These results clearly show that BESPOKV’s ability to map applications to appropriate datalet that best suits application needs improve performance.

B. Distributed cache for deep learning

Machine learning (ML) and deep learning (DL) techniques are becoming more popular in HPC for solving problems in human health, high-energy physics, material discovery and other scientific areas [47]. Accordingly, more HPC systems (e.g., Summit [48], [49]) are being designed to facilitate to run DL applications by employing a hybrid architecture (i.e., combining CPUs and GPUs) with the ample memory space and fast interconnects.

DL applications are characterized by their massively parallel and data-intensive workloads [50]. Especially, pre-processing and ingesting the training dataset from the I/O subsystem becomes a significant bottleneck, which causes GPUs to sit idle waiting for the next dataset to work on. Typical DL algorithms require the entire dataset (e.g., millions of image files) to be passed to the neural network through multiple iterations to reach the optimal result with a desired accuracy. Furthermore, the dataset tends to be extremely large to fit in a single pass and oftentimes has to be divided into multiple batches. Unfortunately, traditional HPC parallel file systems are not designed to accelerate such parallel accesses to massive number of small files, and scientists are seeking for alternative solutions using KV store to expedite the data ingestion process [51]. However, existing KV stores are not flexible enough to support diverse computing environments and application needs. Moreover, building and customizing a distributed KV store can be frustrating to domain scientists.

BESPOKV can be used to build a distributed cache to improve the I/O performance for training DL models. In particular, BESPOKV’s support for multiple backends makes it suitable for wide variety of ML problems where dataset can range from small KV pairs to large objects. BESPOKV also allows application developers to customize the network topologies and consistency models.

We have prototyped a distributed cache using BESPOKV and added support for DPDK kernel bypassing [29] for low latency DL queries. We evaluate the efficacy of our distributed cache by running an image segmentation model with a 100 GB training dataset. Our approach could complete the training 4× faster than the extant approach (40 images/sec. vs. 10 images/sec.).

C. Building burst buffer file systems

Burst buffer file systems are becoming an indispensable framework to quickly absorb application I/O requests in exascale computing [52], [53], [54], [55]. Many burst buffer file systems adopt KV stores to manage file system metadata. BESPOKV allows to develop similar file systems with less development effort. In particular, the dynamic and flexible nature of BESPOKV well suits with ephemeral burst buffer file systems [52]. An ephemeral burst buffer file system has to be dynamically constructed and destroyed within compute nodes assigned to a corresponding job. In such a scenario, BESPOKV can quickly initialize the distributed KV store for storing file system metadata.

Furthermore, BESPOKV also allows to dynamically tune the file system behavior. For instance, it is often preferred to relax the strong POSIX consistency semantics for certain HPC workloads (e.g., checkpointing) to maximize the parallel I/O performance [56]. BESPOKV can simplify the development of such a file system, because it natively supports an instantiation of the distributed KV store with desired consistency and reliability levels.

D. Accelerating the file system metadata performance

KV store is also widely adopted to enhance the performance of file system metadata operations in HPC systems. Metadata performance is one of the major limitations in HPC parallel file systems. A popular approach to address this limitation is to stack up a special file system atop the parallel file system [57], [58], [59]. The stacked file system then quickly absorbs the metadata operations by exploiting a distributed KV store. BESPOKV can accelerate the development of such a stacked file system (evaluated in section VIII-B). Specifically, BESPOKV allows to explore various datalets in backend, and also dynamically tune the file system behavior to comply with the desired performance, consistency and reliability levels.

E. Resource and process management

KV store has also been used to aid the resource and process management in HPC systems [2], [60]. BESPOKV can help develop an advanced job launching system, because it can adapt to different topology and consistency models on the fly. For example, the simple MS topology may be sufficient for handling jobs on a single cluster, but the AA topology may become more suitable when jobs spans multiple clusters (evaluated in sections VIII-B and VIII-C).

VII. BESPOKV IMPLEMENTATION

Current implementation of BESPOKV consists of ~69k lines of C++/Python code without counting comments or blank lines. Except controlets, BESPOKV consists of five components. (1) Control Core implements the control plane backbone
with support for event and message handling. (2) **Client library** helps clients route requests to appropriate controlets, and is extended from libmc [61], an in-memory KV store client library. (3) **Coordinator** uses ZooKeeper [32] to store topology metadata of the whole cluster and coordinates leader elections during failover. It includes a Python-written failover manager that directly controls the data recovery as well as handling BESPOKV process failover. (4) **Lock server APIs** implement two lock server options—ZooKeeper-based [62] and Redlock-based [33]. (5) **Shared Log handler** is implemented using ZLog [34], based on CORFU.

The BESPOKV prototype has four pre-built controlets as described in §IV. All controlet shares the sample event-handling controlet template of 150 LoC. In addition, BESPOKV supports multiple backend datalets with protocol parsers. Using the common datalet template of 966 LoC, we implemented three new datalets with a Protobuf-based [39] parser: \(tHT\), an in-memory hash table; \(tLog\), a persistent log-structured store that uses \(tHT\) as the in-memory index; and \(tMT\), a Masstree-based [63] store. In addition, BESPOKV is compatible with existing single-server KV stores SSDB [37] and Redis [18] that use a simple text-based protocol parser. With protocol parsers, we refer them \(tSSDB\) and \(tRedis\), respectively. Docker based BESPOKV is partially supported right now. We plan to use Kubernetes [64] to simplify deployment in near future.

Using the template-based design approach, we note that for developers (with few years of C/C++ programming experience) non familiar to BESPOKV it took almost three and six person-days time to develop datalet and controlet, respectively. This underscores BESPOKV’s ability to ease development of distributed KV stores.

### VIII. Evaluation

Our evaluation answers the following questions:

- Are BESPOKV-enabled distributed KV stores scalable (section VIII-B), adaptive to topology and consistency changes (section VIII-C), and extensible (section VIII-D)?
- How does BESPOKV compare to existing proxy-based (section VIII-E), and natively-distributed (section VIII-F) KV stores?
- How well BESPOKV handles a node failure? (§D)

#### A. Experimental Setup

**Testbeds and configuration** We perform our evaluation on Google Cloud Engine (GCE) and a local testbed. For larger scale experiments (section VIII-B – section VIII-E), we make use of VMs provisioned from the us-east1-b Zone in GCE. Each controlet–datalet pair runs on an n1-standard-4 VM instance type, which has 4 virtual CPUs and 15 GB memory. Workloads are generated on a separate cluster comprising nodes of n1-highcpu-8 VM type with 8 virtual CPUs to saturate the cloud network and server-side CPUs. A 1 Gbps network interconnect was used.

For performance stress test (section VIII-F) and fault tolerance experiments (§D), we use a local testbed consisting of 12 physical machines, each equipped with 8 2.0 GHz Intel Xeon cores, 64 GB memory, with a 10 Gbps network interconnect. The coordinator is a single process (backed-up using ZooKeeper [32] with a standby process as follower) configured to exchange heartbeat messages every 5 sec with controlets. We deploy the DLM, Shared Log, Coordinator and ZooKeeper on separate set of nodes. BESPOKV’s coordinator communicate with ZooKeeper for storing metadata.

**Workloads** We use two workloads obtained from typical HPC services: job launch, and I/O forwarding and three workloads from the Yahoo! Cloud Serving Benchmark (YCSB) [65].

We use approach similar to [2] to generate HPC workloads. The job launch workload is obtained by monitoring the messages between the server and client during a MPI job launch. Control messages from the distributed servers are treated as \texttt{Get} whereas results from the compute nodes back to the servers as \texttt{Put}. The I/O forwarding workloads is generated by running SeaweedFS [66], a distributed file system which supports KV store for metadata management. The clients first create 10,000 files, and then performs reads or writes (with 50% probability) on each file. We collect the log of the metadata server. We extend these workloads several times until reaching 10M requests with the goal to reflect the time serialization property of the obtained messages.

For YCSB we use an update-intensive workload (\texttt{Get:Put} ratio of 50:50), a read-mostly workload (95% \texttt{Get}), and a scan-intensive workload (95% \texttt{Scan} and 5% \texttt{Put}). All workloads consist of 10 million unique KV tuples, each with 16 B key and 32 B value, unless mentioned otherwise. Each benchmark process generates 10 million operations following a balanced uniform KV popularity distribution and a skewed Zipfian distribution (where Zipfian constant \(= 0.99\)). The reported throughput is measured in terms of thousand queries per second (kQPS) as an arithmetic mean of three runs.

#### B. Scalability

Figure 7 shows the scalability of BESPOKV-enabled distributed \(tHT\). We measure the throughput when scaling out \(tHT\) from 3 to 48 nodes on GCE. The number of replicas is set to three. We present results for all four topology and consistency combinations: MS+SC, MS+EC, AA+SC, and AA+EC. For all cases, BESPOKV scales \(tHT\) out linearly as the number of nodes increases for both read-intensive (95% \texttt{Get}) and write-intensive (50% \texttt{Get}) workloads. For SC, MS+SC using chain replication scales well, while AA+SC performs worse as expected in locking based implementation. For EC, the results show that our EC support scales well for both MS+EC and AA+EC. Performance comparison to existing distributed KV stores will follow in section VIII-F.

![Graph](image-url)
A 48 node tMT cluster gives Scan of persist data on disk. It is as expected that the throughput database and thus outperforms both tLog and tSSDB which all three. In terms of performance, tMT is an in-memory with fault tolerance, MS+EC configuration. While enabling eventual consistency

Due to space constraints, we only present the result with the MS-EC->AA-EC transition starts ends Time (sec)

Unif 95% SCAN
Zipf 95% SCAN

Fig. 10: BESPOKV seamlessly adapts service from MS-EC to MS-SC, AA-EC, and AA-SC.

Fig. 11: BESPOKV adds MS+SC and AA+EC for Redis. Comparison with Dynomite (Dyno) and Twemproxy (Twem). existing solutions because BESPOKV does not require data migration or down time.

D. Extensibility and New Services

As sketched in §IV, BESPOKV can be extended to support new forms of distributed services. This section evaluates two examples: per-request consistency and polyglot persistence.

We evaluate the per-request consistency service (Section IV-C) under MS+SC and a Zipfian workload with a 25:75% ratio of SC:EC as the desired consistency. We observed the performance to be between MS+SC and MS+EC as shown in Figure 7; for example, with 24 nodes, we obtain ~300 k QPS for 95% Get and ~270 k QPS for 50% Get workloads. We also evaluate the average latency of each request. With a weaker consistency requirement, the GET latency is 0.67 ms. We get an average of 1.02 ms latency with default strong consistency.

We test polyglot persistence (section IV-D) by storing each replica in a different type of datalet. We use tHT, tLog and tMT in MS topology with eventual consistency. The performance of the resulting configuration under Uniform workload is very similar to the numbers in Figure 7 and 9; for example, with 24 nodes, we obtain 375 k QPS for 95% Get and 200 k QPS for the 50% Get workload.

E. Comparison to Proxy-based Systems

This section shows that BESPOKV can support new topologies and consistency models for existing single-server KV store, and them compares BESPOKV with two state-of-the-art Proxy-based KV stores. We test BESPOKV+Redis (tRedis) running in MS+SC, MS+EC and AA+EC modes, reusing SSSDB’s text-based protocol parser for Redis. We measure the throughput of tRedis on eight 3- replica shards across 24 nodes on GCE, and compare it with Dynamoite [19] supporting AA+EC only, and Twemproxy [16] supporting MS+EC only.

Fig. 11 shows the throughput. BESPOKV enables new MS+SC (~500 k QPS under Zipfian 95% Get) and AA+EC (~750 k QPS under Zipfian 95% Get) configurations with reasonable performance. As expected, MS+SC is more expensive than MS+EC. Twemproxy is just a proxy to route requests using consistent hashing to a pool of backend servers. Hence, Twemproxy+Redis in supporting MS+EC performs slightly better than BESPOKV in supporting MS+EC. However, we observed the same performance for Dynamoite+Redis in supporting AA+EC configuration for Redis as BESPOKV in supporting AA+EC.

F. Comparison to Natively-Distributed Systems

In this experiment, we compare BESPOKV-enabled KV stores with two widely used natively-distributed (off-the-shelf) KV stores: Cassandra [7] and LinkedIn’s Voldemort [69].

C. Adaptability

We evaluate BESPOKV’s adaptability in switching online consistency levels and topology configurations (§V). In all the tests we use 3 shards with a Zipfian workload of 95% Get. As shown in Figure 10, the transition is scheduled to be triggered at 20 sec. The throughput drops to the lowest point for all three cases. This is because clients switch connection to the new controllers. Performance stabilizes in ~5 sec, because all the in-flight requests are handled during this process. We observe similar trends for other possible transitions that can be enabled by BESPOKV. This demonstrates BESPOKV’s flexibility and adaptability in switching between different key designs & configurations. This also shows that BESPOKV is able to complete switching in extremely short time compared to
These experiments were conducted on our 12-node local testbed in order to avoid confounding issues arising from sharing a virtualized platform. We launch the storage servers on six nodes and YCSB clients on the other four nodes to saturate the server side. The coordinator, lock server (only for AA+SC), ZLog (only for AA+EC), and ZooKeeper are launched on separate nodes. We use tHT as a datalet to show high efficiency of BESPOKV-enabled KV stores.

For Cassandra, we specify consistency level of one to make consistency requirements less stringent. Cassandra’s replication mechanism follows the AA topology with EC [70]. For Voldemort we use a server-side routing policy, all-routing as the routing strategy, a replication factor of three, one as the number of reads or writes that can succeed without client getting an exception, and persistent set to memory.

Figure 12 shows the latency and throughput for all tested systems/configurations when varying the number of clients to increase the throughput in units of kQPS. For AA+EC, BESPOKV outperforms Cassandra and Voldemort. For read-intensive workload, BESPOKV’s throughput gain over Cassandra and Voldemort is 4.5× and 1.6×, respectively. For write-intensive workload, BESPOKV’s throughput gain is 4.4× over Cassandra and 2.75× over Voldemort. In this experiment Cassandra was configured to use persistent storage. However even using Log as a datalet for BESPOKV(also uses persistent storage) we observed a throughput gain of 2.6× and 1.2× over Cassandra and Voldemort, respectively. We suspect that this is because Cassandra uses compaction in its storage engine which significantly affects the write performance and increases the read latency due to use of extra CPU and disk usage [71]. Voldemort uses the same design and both are based on Amazon’s Dynamo paper [6]. Furthermore, our findings are consistent with Dynomite in terms of the performance comparison with Cassandra [71].

As an extra data point, we also see interesting tradeoffs when experimenting with different configurations supported by BESPOKV. For instance, MS+EC performs achievement comparable to AA+EC under 95% Get workload since both configurations serve Gets from all replicas. AA+EC achieves 47% higher throughput than MS+EC under 50% Get workload, because AA+EC serves Puts from all replicas. For AA+SC, lock contention at the DLM caps the performance for both read- and write-intensive workloads. As a result, MS+SC performs 3.2× better than AA+SC for read-intensive workload and ~2× better for the write-intensive workload.

IX. RELATED WORK

Dynomite [19] adds fault tolerance and consistency support for simple data stores such as Redis. Dynomite only supports eventual consistency with AA topology. It also requires the single-server applications to support distributed management functions such as Redis’ streaming data recovery/migration mechanism. BESPOKV’s datalet is completely oblivious of the upper-level distributed management, which offers improved flexibility and programmability.

Pileus [72] is a cloud storage system that offers a range of consistency-level SLAs. Some storage systems offer tunable consistency, e.g., ManhattanDB [73]. Flex-KV [74] is another flexible key-value store that can be configured to act as a non-persistent/durable store and operates consistently/inconsistently. Morphus [75] provides support towards reconfigurations for NoSQL stores in an online manner. MOS [76], [77] and hatS [78], [79] provide flexible and elastic resource-level partitioning for serving heterogeneous object store workloads. ClusterOn [80] proposes to offer generic distributed systems management for a range of distributed storage systems. To the best of our knowledge, BESPOKV is the first generic framework that offers a broad range of consistency/topology options for both users and KV store application developers.

Vsync [80] is a library for building replicated cloud services. BESPOKV embeds single-node KV store application code and automatically scales it with a rich choice of services. Going one step further, BESPOKV can be an ideal platform to leverage library support such as Vsync to further enrich flexibility. EventWave [81] elastically scales inelastic cloud programs. PADS [82] provides policy architecture to build distributed applications. Similarly, MOS [83] provides reusable networking stack to allows developers to focus on the core application logic instead of dealing with low-level packet processing. BESPOKV focuses on a specific domain with a well-defined limited set of events–KV store applications.

Using distributed log to facilitate data management has also been studied. CORFU [35], vCorfu [84], and Tango [85] enable flexible data management by leveraging a Shared Log over an SSD array. BESPOKV utilizes a shard log to not only guarantee ordering but also to provide seamless transition between different topologies (i.e., MS and AA).

X. CONCLUSION

We have presented the design and implementation of BESPOKV, a framework, which takes a single-server data store and transparently enables a scalable, fault-tolerant distributed KV store service. Evaluation shows that BESPOKV is flexible, adaptive to new user requirements, achieves high performance, and scales horizontally. BESPOKV has been open-sourced and is available at https://github.com/tddg/bespokv.

ACKNOWLEDGEMENT

We thank our shepherd, Ioan Raicu, and the reviewers for the valuable feedback. This work is sponsored in part by the NSF under the grants: CNS-1565314, CNS-1405697, CNS-1615411, and CNS-1814430.
APPENDIX A

ARTIFACT DESCRIPTION: bespoKV

Abstract—This description contains the information needed to launch experiments of the SC’18 paper "BESPOKV: Application Tailored Scale-Out Key-Value Stores". More precisely, we explain how to compile, run, deploy, configure, and benchmark BESPOKV to reproduce the results. Additionally, we also explain how to develop controlets used in the examples section of the paper.

A. How to deliver?
bespoKV source can be cloned or downloaded from GitHub repository at https://github.com/tddg/bespokv.

B. Dependencies:
1) gcc 4.8 (required by folly)
2) autoconf2-6.9 (requires v ≥ 2.6.9)
3) google-glog
4) protobuf
5) libopenssl
6) gflags-master
7) boost_1_55_0/ /b2 install
8) folly (requires double_conversion)
9) libuuid
10) libevent
11) lz4 1.7.1 (redlock requires v ≥ 1.7.1)
12) zookeeper (server + C binding lib)

C. How to compile?

Compile bespoKV Go to the src directory, for debugging mode, run make. To enable compiler level optimization, run make opti.

Compile dataset application Go to the apps directory, for debugging mode, run make. To enable compiler level optimization, run make opti.

Compile client lib To compile client lib, go to the libckv dir and compile proto file:
Fig. 13: Initialization code of the MS+SC controllet.

![Initialization code of the MS+SC controllet.](image)

To run the YCSB trace bench:

```bash
$ cd apps/apps/*
$ make
```

And a datalset list file specifying all datalsets, as below:

```json
{
  "zk": "192.168.0.173:9092",
  "mq": "192.168.0.173:2181",
  "ms": "192.168.0.173:9092",
  "cr": "192.168.0.173:9092",
  "topology": "cr",
  "num_replicas": 5,
}
```

Fig. 14: Using a template to create a MS+SC controllet.

![Using a template to create a MS+SC controllet.](image)
Fig. 15: The Put/Get paths in MS+EC (a), AA+SC (b), and AA+EC (c). The Get path is same in all three, except in AA+SC, where the difference is that each Get needs to acquire a read lock before proceeding. Mn means the nth master.

Where hosts.cfg is an example host file including all hosts, -n indicates how many hosts, -r indicates how many replicas, -R specifies which replica to serve READ (-1 means any replica), and -W indicates which replica to serve WRITE (-1 means any replica for active/active topology). Example host file is shown as below:

```
```

Redis benchmark To run the Redis benchmark:

```
./redis-benchmark -h hulk0 -p 12345 -c 50 -n 100000 -t set.get -P 32 -r 100000
```

This will send requests to conproxy, which will serve as a proxy forwarding requests between benchmark clients and Redis backend datalets.

G. How to deploy?

ZK and MQ To launch zk and MQ on cloud, run:

```
bin/zookeeper-server-start.sh -daemon config/zookeeper.properties
bin/MQ-server-start.sh -daemon config/server.properties
```

BESP0KV and datalet nodes To launch a cluster of BESP0KV+ conk nodes, first add the data node info in slap.sh, then run:

```
cd scripts
./slap.sh runckv
./slap.sh runcon
```

H. Docker container based execution

Docker based BESP0KVVis PARTIALLY supported. To run containerized deployment:

```
cd scripts
./slap.sh docker_runckv
./slap.sh docker_runcon
```

APPENDIX B

CONTROLET DEVELOPMENT

Figure 13 shows the code snippet for the controlet initialization. Developer can register callback functions and define extended events during initialization. Figure 14 shows the code snippet for MS+SC controlet built atop our controlet template. The first half of the code snippet (white background) shows the controlet code template. It uses basic events to construct message forwarding logic where a request is accepted from a client connection and forwarded to a datalet connection. Similarly, a response is accepted from the datalet connection and forwarded to the client. The template also provides logic to parse the request to find out the request type. Line 25 and lines 35—57 are developer-defined logic (colored background). Developers provide callback functions OnPut, OnEnqueue, and OnGet to implement Figure 3.

APPENDIX C

BESP0KV-BASED DISTRIBUTED KV STORES

Using hash-based tHT datalet and consistent hashing for the client library as an example, this section presents support for three more widely-used topology and consistency combinations: MS+EC, AA+SC, and AA+EC.

A. Master-Slave & Eventual Consistency

BESP0KV’s pre-built controlet takes a simple approach to support MS+EC where the master copies the data to slaves asynchronously. Example. Figure 15(a) shows an example for MS+EC. Here, upon receiving an incoming Put request (step 1), the master node commits the request to the local datalet (step 2) before it sends an acknowledgement back to the client (step 3). Unlike the previous SC case, the master does not wait until the propagation finishes. Subsequently, BESP0KV provides EC by asynchronously forwarding Put requests to other datalets (step 4).

4Please note that these examples present just one way to implement each combination. Controlet developers can easily implement their own versions.

5This way at least one datalet is written straight away as in Cassandra [7]. An alternative design choice is to forward the request to more than one datalet and then acknowledge back. However, this decision solely depends on the type of eventual consistency that is desired.
**Failover.** Upon a node failure, the coordinator launches a new controlet–datalet pair, and then the new controlet recovers the requests from another datalet. For MS+EC, the new pair is added as a slave. If the master node fails, the coordinator promotes one of the slave nodes to master after a leader election process. The coordinator then updates the cluster topology metadata so that future incoming writes can be routed to the new master, similar to the case of head failure in MS+SC.

**B. Active-Active & Strong Consistency**

Supporting AA and SC is expensive in general. AA allows multiple nodes to handle Put requests and SC requires global ordering (serialization) between them. Thus, CR-like optimization is not applicable under AA. For simplicity and comparison purposes, the current BESPOKV’s AA+SC controlet takes the distributed locking based implementation, using the DLM library (section III-B). For performance improvement, optimistic concurrency control [86] and inconsistent replication [9] can be added. Instead of using DLM, one can also enable SC using a Shared Log to maintain a global and sequential order of concurrent requests, which we used for AA+EC later in a relaxed manner.

**Example.** Figure 15(b) shows a DLM-based AA+SC example. Clients’ Put requests are routed to any controlet (step 1 and step 2). Concurrent Puts from another client (step 2 in our example) are synchronized via the distributed locking service. The first receiving controlet acquires a write lock (step 3) on the key and updates all the relevant datalets (step 4 & 5), releases the lock (step 6), and finally acknowledges to the client (step 7). For a Get request, the controlet that receives the request acquires a read lock on that key, reads the value from the local datalet, releases the lock, and then sends a response back to the client.

**Failover.** Like the previous cases, when a node fails the coordinator launches a new controlet–datalet pair. The new controlet then performs data recovery from another datalet. As AA+SC uses locking, ensuring SC for the new node and adding it as an active node are trivial because all writes are synchronized using locks. However, deadlock freedom should be guaranteed. Thus, BESPOKV enforces that locks are released after a configurable period of time. If a controlet fails after acquiring a lock, the lock is auto-released after it expires. Note that if a lock is auto-released, but a controlet has not failed and was simply unresponsive for a while, it is terminated to ensure proper continuation of operations. Also, one of the master nodes cleans up the in-flight requests.

**C. Active-Active & Eventual Consistency**

For an AA topology, relaxed data consistency is more widely used in practice for performance as in Dynamo [6], Cassandra [7]), and Dynomite [19]. In particular, these systems use gossip-based protocols and provide a weaker data consistency model, e.g., acknowledging back to the client if a Put request is written to one node, N nodes, or a quorum [87].

In order to ensure EC, when multiple masters receive concurrent PUT requests, AA should be able to resolve conflicts and agree on the global order of them, unlike MS where one master gets all the writes. In this sense, Dynomite does not support (a strict form of) EC when conflicting PUT requests arrive within a time period less than the latency of replication [88].

To address this issue, BESPOKV’s AA+EC controlet uses a Shared Log to keep track of the request ordering. From the Shared Log, asynchronous propagation of writes occur to support EC. One disadvantage of this approach is that we need to scale the Shared Log setup as BESPOKV scales. Alternative approach is to add anti-entropy/reconciliation [89].

**Example** Figure 15(c) depicts how BESPOKV supports AA+EC. In AA, clients can route Get/Put to any of the master controlets (step 1a). On a Put, the receiving controlet (in our example the leftmost one) writes to the Shared Log first (step 2a), commits the request on its local datalet (step 3a), and then responds back to the client (step 4a). All the controlets asynchronously fetch the request (step 5). Gets can be handled by any of the corresponding controlets by retrieving the data from their local datalets. The duration to keep the requests in Shared Log is configurable.

**Failover.** For AA+EC, the failover is handled like with MS+EC, except that leader election is not needed in this case.

**Discussion.** Load imbalance due to hot keys (i.e., hotspots) can be solved by integrating a small metadata cache at BESPOKV’s client library to keep track of hot keys [90]; once the popularity of hot keys exceeds a certain pre-defined threshold, client library replicates this key on a shadow server that is rehashed by adding a suffix to the key. In fact, our proxy-based architecture naturally fits for adding a controlet-side small cache or data migration/replication for load balancing purpose [91], [92], [93], [94], [95].

**APPENDIX D**

**FAILOVER & DATA RECOVERY**

We also evaluate how BESPOKV performs in case of a node failure, and compare it with Redis’s replication used by Dynomite for failover recovery. In this set of tests, we use 3 shards (each with 3 replicas) to clearly reflect the impact of a failure on throughput. The workload consists of 1 million KV tuples generated with a Zipfian distribution. We intentionally crash a node to emulate a failure, and Figure 16 shows the resulting throughput change.

**MS topology** For MS+SC, we bring down the head node under the write-intensive workload (50% Put, as shown in the bottom half of Figure 16 (a)) and the tail node for the
read-intensive workload (95% Get, as shown in top half of the figure), to maximize the performance disruption on the respective workloads. For MS+EC, we take down the master node for the write-intensive workload and a random slave node for the read-intensive workload.

We observe that for MS+SC, Put throughput goes down by about 1/3 when the head node crashes at 20 sec, as we have 3 shards. The coordinator detects the node failure from the lack of heartbeat message before assigning the master role to the second node in the chain. The coordinator then launches a new controllet–datalet pair in recovery mode, and inserts the pair to the end of the chain once data recovery completes at around 35 sec. Meanwhile the throughput stabilizes. MS+EC failover shows a similar trend. The top half of Figure 16 (a) shows the impact of node failure on Get performance under MS topology. For MS+SC, killing the tail brings down Get throughput by 1/3. Once failure is detected, the coordinator makes the 2
\textsuperscript{nd}–from-last node in the chain the new tail, and updates the topology metadata. Once clients see the update, they reroute the corresponding Get requests to the new tail. Hence, the throughput goes back to normal in ~5 sec. MS+EC behaves differently as Gets are served by any of the 3 replicas. Thus, the slave failure does not affect the performance as much (throughput drops by ~1/9).

**AA topology** In BESPOKV’s AA and Dynomite (with Redis) failover test, we randomly kill a node at 20 sec and record the overall throughput. As shown in Figure 16(b), the throughput is slightly impacted in all cases, because both BESPOKV AA and Dynomite serve reads and writes from all replicas. Dynomite leverages Redis’ master-slave replication to recover data directly from the surviving nodes. We observe trend similar to Dynomite as BESPOKV also uses datalet’s callback functions to import and export the data.

**APPENDIX E**

**DPDK OPTIMIZATION**

We recently added support for DPDK based communication between clients, controlets, and datalets in to BESPOKV. In this experiment, we show performance of socket vs. DPDK based communication. We deployed a single shard on our local testbed and measured latency and throughput. Each node in our local setup is equipped with Intel ethernet controller X540-AT2. Figure 17 shows that DPDK reduces latency by up to 65%. We also observe 3\times improvement in throughput compared to socket based communication. Another interesting finding is that DPDK based communication results in more stable performance.