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*

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 comparably 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-Active[1] topology and Eventual Consistency (AA+EC).

Another key limitation of natively-distributed KV stores lie in their *inflexible* monolithic design where distributed features

---

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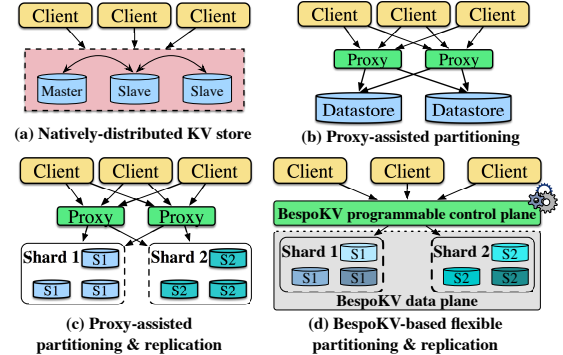

Fig. 1: Different approaches to enable distributed KV stores: (a) natively-distributed (b-d) proxy-based.

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 Behance [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, Netflix 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 AA+EC only. We see that existing solutions have not yet extracted the full potential of proxy-based distributed KV stores. Table I summarizes the limitations of existing proxy-based KV solutions such as Dynomite and Twmemproxy.

System	S	R	MB	MC	MT	AR	P
Single-server	✗	✗	✗	✗	✗	✗	✗
Twemproxy	✓	✗	✓	✗	✗	✗	✗
Mcrouter	✓	✗	✗	✗	✗	✗	✗
Dynomite	✓	✓	✓	✗	✗	✗	✗
BESPOKV (Our work)	✓	✓	✓	✓	✓	✓	✓

TABLE I: BESPOKV vs. state-of-the-art systems for KV stores. S: Sharding; R: Replication; MB: Multiple back-ends; MC: Multiple consistency techniques, e.g. strong, eventual, per-request, etc.; MT: Multiple network topologies, e.g. Master-Slave, Master-Master, Peer-to-Peer, etc.; AR: Automatic failover recovery; P: Programmable.

This paper presents BESPOKV, a flexible, ready-to-use, adaptive, and extensible distributed KV store framework. Figure 1(d) illustrates BESPOKV's distributed KV store architecture. BESPOKV takes as input a single-server KV store, which we call *datalet*, and transparently enables a distributed KV store service, supporting a variety of cluster topologies, consistency models, replication options, and fault tolerance (§III). To the best of our knowledge, BESPOKV is *the first system supporting multiple consistency techniques, multiple network topologies, dynamic topology/consistency adaptation, automatic failover, and programmability, all at the same time.*

Decoupling control and data planes brings three unique benefits to BESPOKV that are not possible in existing distributed KV stores. First, given a (single-server) datalet, BESPOKV enables *immediately-ready-to-use* distributed KV stores. Developers can simply "drop" a datalet to BESPOKV and offload the "messy plumbing" of distributed systems support to the framework. BESPOKV transparently supports multiple backends and different combinations of cluster topologies and consistency models: e.g., MS+SC, MS+EC, AA+SC, AA+EC, and more (§IV). Second, BESPOKV makes distributed KV stores *adaptive* to dynamic changes. BESPOKV supports seamless on-the-fly cluster topology and data consistency changes by updating controlets while keeping datalets unchanged: e.g., MS+EC to MS+SC, and AA+EC to MS+EC (§V). Lastly, BESPOKV enables *extensible* distributed KV stores. Starting from default controlets, developers can quickly synthesize new and innovative services: e.g. KV stores with hybrid topologies. In essence, BESPOKV leverages the reusability principles [22] to simplify the task of constructing systems with new designs.

This paper makes the following contributions:

- We propose a novel distributed KV store architecture that enables innovative uses of KV stores in HPC applications, Our implementation of BESPOKV is publicly available at https://github.com/tddg/bespokv.
- We demonstrate that BESPOKV can be easily extended to offer advanced features such as range query, per-request consistency, polyglot persistence, and more. To the best of our knowledge, BESPOKV is first to support a seamless on-the-fly topology/consistency adaptation. We also present several use cases to show effectiveness of BESPOKV.
- We deploy 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 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
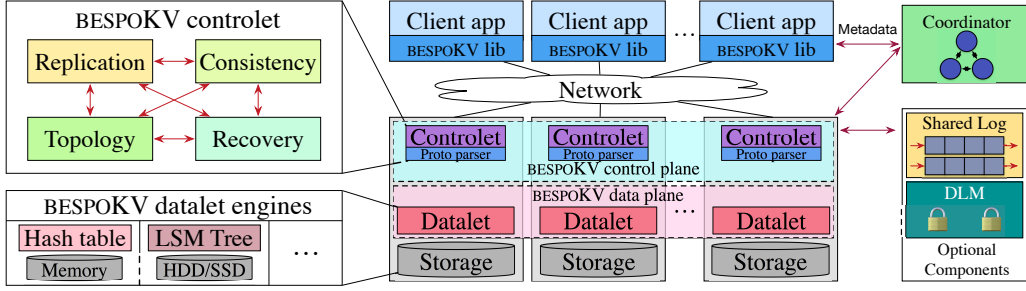
Fig. 2: BESPOKV architecture and the interactions between components. `LSM Tree`: Log-structured merge-tree. `DLM`: Distributed lock manager.

(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 simplest 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 (≥ 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

Datalet API (provided by application developers)	
`Put(key, val)`	Write the {`key`,`val`} pair to the datalet
`val=Get(key)`	Read `val` of `key` from the datalet
`Del(key)`	Delete {`key`,`val`} pair from the datalet
Client API (provided by BESPOKV)	
`CreateTable(T)`	Create a table `T` to insert data
`Put(key, val, T)`	Write the {`key`,`val`} pair to table `T`
`val=Get(key, T)`	Read `val` of `key` from table `T`
`Del(key, T)`	Delete {`key`,`val`} pair from table `T`
`DeleteTable(T)`	Delete table `T`

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

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 datalet is completely unaware of other datalets. **Datalet Development.** BESPOKV supports multiple backends. Users can

Events 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 onc time for conn `c`
`On(e,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 `L`
`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`
`map=GetShardInfo(s)`	Get controlet & datalet list for shard `s`
`c=LeaderElect(s)`	Elect new Master controlet for shard `s`

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

make use of off-the-shelf single-serve data stores such as Redis [18], SSDB [37], and Masstree [21]. In addition, BE-SPOKV 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.

B. 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-built 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
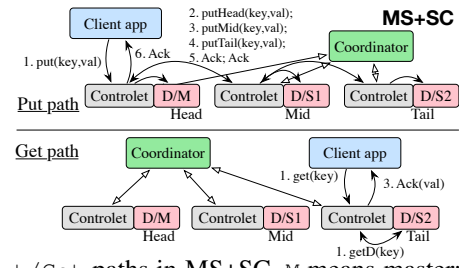

Fig. 3: `Put`/`Get` paths in MS+SC. `M` means master; `Sn` means the nth slave; `D` means datalet.

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. **Controlet 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.

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 *tMT* 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
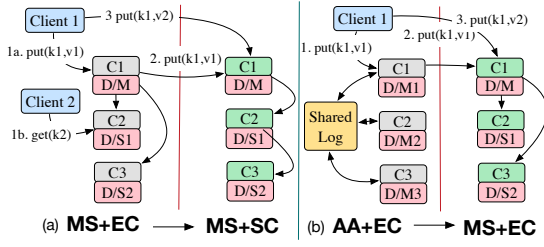
Fig. 4: Transition: MS+EC to MS+SC and AA+EC to MS+...

Fig. 5: `Put`/`Get` paths in MS+EC for HPC monitoring to perform I/O load balancing.

requested, may experience EC until the transition is c...
As controlet developers are responsible for developing ...
transition functionality for the various consistency/topo...
modes. A controlet developer can choose an alternative r...
to fence all writes as soon as the reconfiguration is reque...
so that all reads observe the same and latest applied value.

Figure 4 (a) shows transition from MS+EC to MS+SC [2].
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 `Put`s, an active
node relies on the Shared Log that propagates `Put`s 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 `Put`s from the Shared Log to the new master.
To this end, when the transition starts, the new master node
takes the in-flight `Put`s 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 forward 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 chal-
lenge. Statistical analysis [45] and Markov chain model [46]
have been used to predict shared resource usage. A KV store
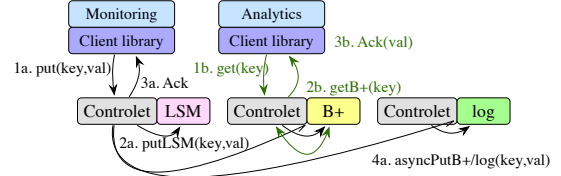
can be used to collect runtime statistics from HPC storage
systems for accurate prediction. However, existing KV stores
are designed for one type of storage architecture (in-memory,
SSD, NVM, etc.), leading to suboptimal performance.

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 applica-
tions 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
need. 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 deploy-
ment. First application is Hardware Accelerated Cosmology
Code (HACC), second is IOR benchmark, and third is a
transaction processing application from a large financial in-
stitution. 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

---

[2]Reverse transition from MS+SC to MS+EC is trivial as the new master
just needs to start using asynch. propagation instead of chain replication.

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 through-put 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 BE-SPOKV's ability to map applications to appropriate datalet that best suits application needs improve performance.
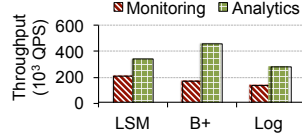

Fig. 6: Effect of using different data abstractions.

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], a 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 are 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
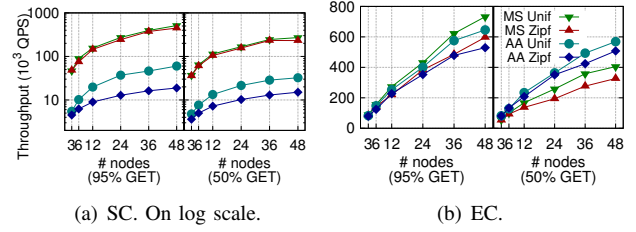

Fig. 7: BESPOKV scales *tHT* horizontally.

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 Get whereas results from the compute nodes back to the servers as 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 (Get:Put ratio of 50%:50%), a read-mostly workload (95% Get), and a scan-intensive workload (95% Scan and 5% 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

In this test, we evaluate the scalability of the BESPOKV-enabled distributed KV store using four datalets: tHT and tLog, as examples of newly developed datalets; and tSSDB and tMT, as representatives of existing persistent KV stores. 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% Get) and write-intensive (50% 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
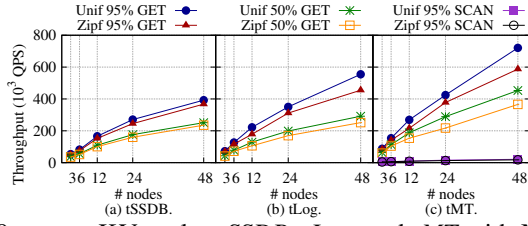
Fig. 9: BESPOKV scales *tSSDB*, *tLog*, and *tMT* with MS+EC. AA+EC. Performance comparison to existing distributed KV stores will follow in section VIII-F.

Figure 8 shows similar trend for HPC oriented workloads. We again observe that MS outperforms AA for SC whereas the trend is opposite for EC where AA performs better than MS. We also observe that performance of I/O forwarding is slightly better than Job launch. This is because I/O forwarding workload has 12% more reads than Job launch with Get:Put ratio of 62%:38%.
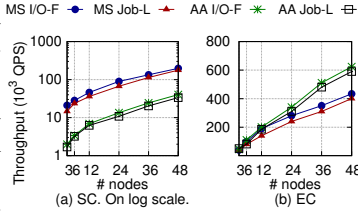

Fig. 8: BESPOKV scales HPC workloads.

Figure 9 shows the scalability when varying the number of nodes from 3 to 48, with tSSDB, tLog, and tMT as datalet. Due to space constraints, we only present the result with the MS+EC configuration. While enabling eventual consistency with fault tolerance, BESPOKV provides good scalability for all three. In terms of performance, tMT is an in-memory database and thus outperforms both tLog and tSSDB which persist data on disk. It is as expected that the throughput of Scans (range queries) is much lower than point queries. A 48 node tMT cluster gives $18k$ QPS on Zipfian 95% Scan, while Uniform yields slightly higher throughput ($21k$). Interestingly, this test covers a potential use case of BESPOKV+tLog for flash storage disaggregation, where users can exploit the scale-out capacity of an array of fast SSD (flash) devices/nodes with low-latency datacenter network [67], [68].

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 controlets. 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
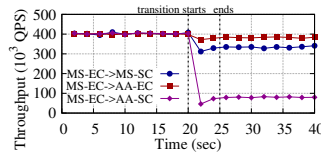

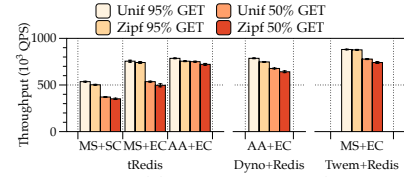Fig. 10: BESPOKV seamlessly adapts service from MS-EC to MS-SC, AA-EC, and AA-SC.

Fig. 11: BESPOKV adds MS+SC and AA+EC for Redis. Comparison with Dynomite (Dyno) and Twemproxy (Twem).

complete switching in extremely short time compared to 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 ~$300k$ QPS for 95% Get and ~$270k$ 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 $375k$ QPS for 95% Get and $200k$ 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 SSDB'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 Dynomite [19] supporting AA+EC only, and Twemproxy [16] supporting MS+EC only.

Figure 11 shows the throughput. BESPOKV enables new MS+SC (~$500k$ QPS under Zipfian 95% Get) and AA+EC (~$750k$ 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 Dynomite+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)
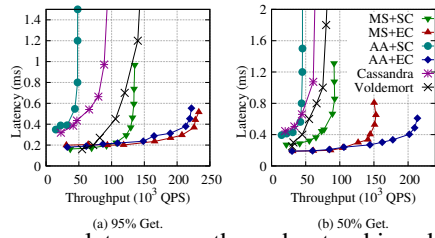
Fig. 12: Average latency vs. throughput achieved by various systems under Zipfian workloads.

KV stores: Cassandra [7] and LinkedIn's Voldemort [69]. 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 and show that BESPOKV-enabled KV stores can achieve comparable (sometime better) performance with state-of-the-art systems, proving its high efficiency.

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 persistence 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.[3] For AA+EC, BESPOKV outperforms Cassandra and Voldemort. For read-intensive workload ,BESPOKV's throughput gain over Cassandra and Voldemort is $4.5\times$ and $1.6\times$, respectively. For write-intensive workload , BESPOKV's throughput gain is $4.4\times$ over Cassandra and $2.75\times$ over Voldemort. In this experiment Cassandra was configured to use persistent storage. However even using *tLog* as a datalet for BESPOKV(also uses persistent storage) we observed a throughput gain of $2.6\times$ and $1.2\times$ over Cassandra and Voldemort, respectively. We suspect that this is because Cassandra uses compaction in its storage engine which significantly effects 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 achieves performance 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

---

[3]Uniform workloads show similar trend, hence are omitted.

both read- and write-intensive workloads. As a result, MS+SC performs $3.2\times$ better than AA+SC for read-intensive workload and ~$2\times$ 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 [38] 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.

REFERENCES

[1] J. Kim, S. Lee, and J. S. Vetter, "Papyruskv: a high-performance parallel key-value store for distributed nvm architectures," in *ACM/IEEE SC'17*.

[2] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Using simulation to explore distributed key-value stores for extreme-scale system services," in *ACM/IEEE SC'13*.

[3] Z. W. Parchman, F. Aderholdt, and M. G. Venkata, "Sharp hash: A high-performing distributed hash for extreme-scale systems," in *IEEE Cluster'17*.

[4] S. Eilemann, F. Delalondre, J. Bernard, J. Planas, F. Schuermann, J. Biddiscombe, C. Bekas, A. Curioni, B. Metzler, P. Kaltstein *et al.*, "Key/value-enabled flash memory for complex scientific workflows with on-line analysis and visualization," in *IEEE IPDPS'16*.

[5] "MongoDB," https://www.mongodb.com/.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," ser. ACM SOSP '07.

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

[8] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *USENIX NSDI '13*.

[9] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," in *ACM SOSP'15*.

[10] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," in *ACM SIGCOMM '12*.

[11] "Social Artisan," http://socialartisan.co.uk/.

[12] "Behance," https://www.behance.net/.

[13] "The Migration Process," https://academy.datastax.com/planet-cassandra//mongodb-to-cassandra-migration/#data_model.

[14] "Why flowdock migrated from cassandra to mongodb," http://blog.flowdock.com/2010/07/26/flowdock-migrated-from-cassandra-to-mongodb/.

[15] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook." in *USENIX NSDI'13*.

[16] "Twitter's Twemproxy," https://github.com/twitter/twemproxy.

[17] "Memcached," https://memcached.org/.

[18] "Redis," http://redis.io/.

[19] "Netflix's Dynomite," https://github.com/Netflix/dynomite.

[20] "LevelDB," https://github.com/google/leveldb.

[21] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *ACM EuroSys '12*.

[22] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, Jun. 1992.

[23] "Google Cloud Platform," https://cloud.google.com/compute/.

[24] H. N. Greenberg, J. Bent, and G. Grider, "Mdhim: A parallel key/value framework for hpc," in *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2813749.2813759

[25] Y. Cheng, F. Douglis, P. Shilane, M. Tratchman, G. Wallace, P. Desnoyers, and K. Li, "Erasing belady's limitations: In search of flash cache offline optimality," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[26] "Workload Analysis of KV Stores," https://www.snia.org/sites/default/files/SDC/2017/presentations/Storage_Architecture/Verma_Vishal_Gohad_Tushar_Workload_Analysis_of_Key-Value_Stores_on_Non-Volatile_Media.pdf.

[27] "B-Trees, Fractal Trees, Heaps and Log Structured Merge Trees, Where did they all come from and Why?" https://www.percona.com/live/17/sites/default/files/slides/Heaps%20B-trees%20log%20structured%20merge%20trees%202017-04-25.pptx_.pdf.

[28] "The CIFAR-10 dataset," https://www.cs.toronto.edu/~kriz/cifar.html.

[29] "DPDK," http://dpdk.org/.

[30] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM ASPLOS '13*.

[31] J. Mars, L. Tang, and R. Hundt, "Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity," *IEEE CAL'11*.

[32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC'10*.

[33] "Distributed locks with Redis," http://redis.io/topics/distlock.

[34] "ZLog," https://github.com/noahdesu/zlog.

[35] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, "Corfu: A shared log design for flash clusters," in *USENIX NSDI '12*.

[36] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, "Malacology: A programmable storage system," in *ACM EuroSys '17*.

[37] "SSDB," https://github.com/ideawu/ssdb.

[38] "Vsync," https://vsync.codeplex.com/.

[39] "Google Protocol Buffers," https://developers.google.com/protocol-buffers/.

[40] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *USENIX OSDI'04*.

[41] S. Almeida, J. a. Leitão, and L. Rodrigues, "Chainreaction: A causal+ consistent datastore based on chain replication," in *ACM EuroSys'13*.

[42] J. Terrace and M. J. Freedman, "Object storage on craq: High-throughput chain replication for read-mostly workloads," in *USENIX ATC'09*.

[43] "Polyglot persistence," https://en.wikipedia.org/wiki/Polyglot_persistence.

[44] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM '01*.

[45] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing output bottlenecks in a supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 8.

[46] A. K. Paul, A. Goyal, F. Wang, S. Oral, A. R. Butt, M. J. Brim, and S. B. Srinivasa, "I/o load balancing for big data hpc applications," in *Big Data (Big Data), 2017 IEEE International Conference on*. IEEE, 2017, pp. 233–242.

[47] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 193–205.

[48] "ORNL Launches Summit Supercomputer," //www.ornl.gov/news/ornl-launches-summit-supercomputer.

[49] "Summit," https://www.olcf.ornl.gov/summit/.

[50] L. Zhou, S. Pan, J. Wang, and A. V. Vasilakos, "Machine learning on big data: Opportunities and challenges," *Neurocomputing*, vol. 237, pp. 350–361, 2017.

[51] S.-H. Lim, S. R. Young, and R. M. Patton, "An analysis of image storage systems for scalable training of deep neural networks," *system*, vol. 5, no. 7, p. 11, 2016.

[52] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 69.

[53] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "Burstmem: A high-performance burst buffer system for scientific applications," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 71–79.

[54] D. Shankar, X. Lu, and D. K. D. Panda, "Boldio: A hybrid and resilient burst-buffer over lustre for accelerating big data i/o," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 404–409.

[55] X. Shi, M. Li, W. Liu, H. Jin, C. Yu, and Y. Chen, "Ssdup: a traffic-aware ssd burst buffer for hpc systems," in *Proceedings of the International Conference on Supercomputing*. ACM, 2017, p. 27.

[56] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. Panda, "A 1 pb/s file system to checkpoint three million mpi tasks," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM, 2013, pp. 143–154.

[57] S. Patil and G. Gibson, "Scale and Concurrency of GIGA+: File System Directories with Millions of Files," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST '09, 2011.

[58] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, Piscataway, NJ, USA, 2014.

[59] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "Deltafs: Exascale file systems scale better without dedicated servers," in *Proceedings of the 10th Parallel Data Storage Workshop.* ACM, 2015, pp. 1–6.

[60] R. H. Castain, D. G. Solt, J. Hursey, and A. Bouteiller, "Pmix: process management for exascale environments," in *Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI/USA 2017, Chicago, IL, USA, September 25-28, 2017*, 2017, pp. 14:1–14:10.

[61] "libmc," https://github.com/douban/libmc.

[62] "ZooKeeper Recipes and Solutions," https://zookeeper.apache.org/doc/r3.1.2/recipes.html.

[63] "Embedded Masstree," https://github.com/rmind/masstree.

[64] "Kubernetes: Production-Grade Container Orchestration," https://kubernetes.io/.

[65] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *ACM SoCC '10.*

[66] "SeaweedFS," https://github.com/chrislusf/seaweedfs.

[67] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," ser. ACM EuroSys '16.

[68] N. Zhao, A. Anwar, Y. Cheng, M. Salman, D. ping Li, J. Wan, C. Xie, X. He, F. Wang, and A. R. Butt, "Chameleon: An adaptive wear balancer for flash clusters," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[69] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *USENIX FAST'12.*

[70] N. Carvalho, H. Kim, M. Lu, P. Sarkar, R. Shekhar, T. Thakur, P. Zhou, and R. H. Arpaci-Dusseau, "Finding consistency in an inconsistent world: Towards deep semantic understanding of scale-out distributed databases," in *USENIX HotStorage '16.*

[71] "Why not Cassandra," http://www.dynomitedb.com/docs/dynomite/v0.5.6/faq/.

[72] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *ACM SOSP'13.*

[73] "Manhattan, our real-time, multi-tenant distributed database for Twitter scale," http://goo.gl/7EThfo.

[74] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini, "Flex-KV: Enabling high-performance and flexible KV systems," in *Workshop on Management of Big Data Systems'12.*

[75] M. Ghosh, W. Wang, G. Holla, and I. Gupta, "Morphus: Supporting online reconfigurations in sharded nosql systems," *IEEE Transactions on Emerging Topics in Computing*, 2015.

[76] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, "Taming the cloud object storage with mos," in *ACM PDSW*, 2015.

[77] ——, "Mos: Workload-aware elasticity for cloud object stores," in *ACM HPDC*, 2016.

[78] K. Krish, A. Anwar, and A. R. Butt, "hats: A heterogeneity-aware tiered storage for hadoop," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on.* IEEE, 2014, pp. 502–511.

[79] ——, "[phi] sched: A heterogeneity-aware hadoop workflow scheduler," in *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on.* IEEE, 2014, pp. 255–264.

[80] A. Anwar, Y. Cheng, H. Huang, and A. R. Butt, "Clusteron: Building highly configurable and reusable clustered data services using simple data nodes," in *USENIX HotStorage'16.*

[81] W.-C. Chuang, B. Sang, S. Yoo, R. Gu, M. Kulkarni, and C. Killian, "Eventwave: Programming model and runtime support for tightly-coupled elastic cloud applications," in *ACM SOCC'13.*

[82] N. M. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm, "Pads: A policy architecture for distributed storage systems." in *USENIX NSDI'09.*

[83] M. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mos: A reusable networking stack for flow monitoring middleboxes," in *USENIX NSDI'17.*

[84] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman *et al.*, "vcorfu: A cloud-scale object store on a shared log," in *USENIX NSDI'17.*

[85] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed data structures over a shared log," in *ACM SOSP '13.*

[86] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[87] "Cassandra Configuration," http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html.

[88] "How Dynomite handles the data conflict," https://github.com/Netflix/dynomite/issues/274.

[89] R. Van Renesse, D. Dumitriu, V. Gough, and C. Thomas, "Efficient reconciliation and flow control for anti-entropy protocols," in *ACM Workshop on Large-Scale Distributed Systems and Middleware'08.*

[90] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *ACM SOCC'13.*

[91] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect: Provable load balancing for randomly partitioned cluster services," in *ACM SOCC '11.*

[92] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with switchkv," in *USENIX NSDI '16.*

[93] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," ser. ACM EuroSys '15.

[94] Y. Cheng, A. Gupta, A. Povzner, and A. R. Butt, "High performance in-memory caching through flexible fine-grained services," in *ACM SOCC '13.*

[95] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles.* ACM, 2017, pp. 121–136.

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) autoconf-2.69 (requires v $\geq$ 2.69)
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 $\geq$ 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 datalet 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: