# Customizable Scale-Out Key-Value Stores

Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, *Fellow, IEEE*, and Ali R. Butt

**Abstract**—Enterprise KV stores are often not well suited for HPC applications, and thus cumbersome end-to-end KV design customization is required to meet the needs of modern HPC applications. To this end, in this article 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. For the control plane, BESPOKVprovides pre-built modules, called *controlets*, supporting common distributed functionalities (e.g., replication, consistency, and topology) and their various combinations. This decoupling allows BESPOKV to take a user-provided 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. Such specializations enable innovative uses of KV stores in HPC applications, especially for emerging applications that utilize KV-friendly workloads. We evaluate BESPOKV in a local testbed as well as in a public cloud settings. Experiments show that BESPOKV-enabled distributed KV stores scale horizontally to a large number of nodes, and performs comparably and sometimes 1.2× to 2.6× better than the state-of-the-art systems.

**Index Terms**—Key-value stores, HPC KV stores, scale-out KV stores, application tailored storage

## 1 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], shown in Fig. 1a, 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).

The key limitation of natively-distributed KV stores lie in their *inflexible* monolithic design where distributed features are deeply baked with backend data stores. Such a design allows the developers to highly optimize the KV store performance. However, such optimizations are not portable to any other KV store. 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 Fig. 1b. Twemproxy supports additional Redis [18] backend

• *A. Anwar is with IBM Research–Almaden, San Jose, CA 95120-6099. E-mail: ali.anwar2@ibm.com.*
• *Y. Cheng is with George Mason University, Fairfax, VA 22030. E-mail: yuecheng@gmu.edu.*
• *H. Huang is with IBM Research–T.J. Watson, Ossining, NY 10562. E-mail: haih@us.ibm.com.*
• *J. Han and A.R. Butt are with Virginia Tech, Blacksburg, VA 24061. E-mail: jingoo@vt.edu, butta@cs.vt.edu.*
• *H. Sim is with Oak Ridge National Laboratory, Oak Ridge, TN 37830. E-mail: simh@ornl.gov.*
• *D. Lee is with Stony Brook University, Stony Brook, NY 11794, and also with Virginia Tech, Blacksburg, VA 24061. E-mail: dongyoon@cs.stonybrook.edu.*
• *F. Douglis is with Perspecta Labs, Basking Ridge, NJ 07920. E-mail: fd-ic@douglis.org.*

*Manuscript received 11 June 2019; revised 25 Feb. 2020; accepted 8 Mar. 2020. Date of publication 30 Mar. 2020; date of current version 24 Apr. 2020. (Corresponding author: Ali Anwar.) Recommended for acceptance by J. Wang. Digital Object Identifier no. 10.1109/TPDS.2020.2982640*

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

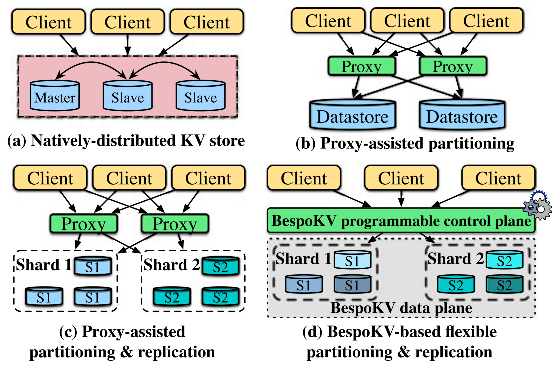

Client
Client
Client
Master
Slave
Slave
(a) Natively-distributed KV store
Client
Client
Client
Proxy
Proxy
Datastore
Datastore
(b) Proxy-assisted partitioning
Client
Client
Client
Proxy
Proxy
Shard 1
S1
Shard 2
S2
(c) Proxy-assisted partitioning & replication
Client
Client
Client
BespoKV programmable control plane
Shard 1
S1
Shard 2
S2
BespoKV data plane
(d) BespoKV-based flexible partitioning & replication

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

as well. Recently, Netflix Dynomite [19] extended Twemproxy to support high availability and cross-datacenter replication, as illustrated in Fig. 1c.

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 1 summarizes the limitations of existing proxy-based KV solutions such as Dynomite and Twemproxy.

This paper presents BESPOKV, a flexible, ready-to-use, adaptive, and extensible distributed KV store framework. Fig. 1d 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 (Section 3). For the control plane, BESPOKV provides a set of distributed

TABLE 1
BESPOKV versus State-of-the-Art Systems for KV Stores

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

*S: Sharding; R: Replication; MB: Multiple backends; 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.*

management units, referred to as *controlets*. 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.*

Table 2 shows the benefit of the proposed BESPOKV framework. Here, four snippet implementations of the core functions for a simple KV store are presented in pseudocode. To implement everything from scratch (`(a) Vanilla`), a developer creates her own concurrency control functionality (`Lock()`, `Unlock()`), and consistency and quorum management logic (`Sync()`, `Quorum()`). Using a distributed lock server (`(b) Lockserver-based`), the developer can avoid implementing synchronization functions. Similarly, using Vsync [22] library (`(c) Vsync-based`) for consistency management further reduces engineering effort. However, there are two limitations. First, developers still need to familiarize themselves with a large collection of system/library interfaces to use them appropriately in the application code. Second, such approaches often provide only a single technique for replication or consistency: e.g., Vsync uses only a virtual synchrony to replicate data. In contrast, using BESPOKV (option (d)), developers only need to implement a non-distributed version of the KV store (*datalet*), and then BESPOKV transparently scales it out to a variety of distributed environments with different requirements.

TABLE 2
An Example of Four Possible Approaches to Developing a Distributed KV Store With the Last One Being the Proposed Approach

**(a) Vanilla**

```
void Put(Str key, Obj val) {
  if (this.master):
     Lock(key)
     Table.insert(key, val)
     Unlock(key)
  Sync(master.slaves)
}

Obj Get(Str key) {
  if (this.master)
     Obj val = Quorum(key)
  Sync(master.slaves)
  return val
}

void Lock(Str key) {
  ... // Acquire lock
}

void Unlock(Str key) {
  ... // Release lock
}

void Sync(Replicas peers) {
  ... // Update replicas
}

void Quorum(Str key) {
  ... // Select a node
}
```

**(b) Lockserver-based**

```
void Put(Str key, Obj val) {
  if (this.master):
     ls.Lock(key) // lockserver
     Table.insert(key, val)
     ls.Unlock(key) // lockserver
  Sync(master.slaves)
}

Obj Get(Str key) {
  if (this.master)
     Obj val = Quorum(key)
  Sync(master.slaves)
  return val
}

void Sync(Replicas peers) {
  ... // Update replicas
}

void Quorum(Str key) {
  ... // Select a node
}
```

**(c) Vsync-based**

```
#include <vsynclib>

void Put(Str key, Obj val) {
  if (this.master):
     ls.Lock(key) // lockserver
     Table.insert(key, val)
     ls.Unlock(key) // lockserver
  Vsync.Sync(master.slaves)
}

Obj Get(Str key) {
  if (this.master)
     Obj val = Vsync.Quorum(key)
  Vsync.Sync(master.slaves)
  return val
}
```

**(d) BESPOKV-based**

```
void Put(Str key, Obj val) {
  Table.insert(key, val)
}

Obj Get(Str key) {
  return Table(key)
}
```

*In case of (a) vanilla, LoC of Lock, Unlock, Sync, and Quorum is not shown. Similarly, LoC to implement Lock and Unlock recipe for ZooKeeper is not shown. Vsync is available in C# and requires use of proper APIs but for the sake of simplicity and consistency we assume a C++ language grammar.*

BESPOKV's decoupled control and data plane architecture, configurability, and extensibility enable new solutions 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, 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 supports the use of different datalets, each of which can be tuned for different memory and storage architecture. BESPOKV's proxy-based design may add performance overhead with an additional layer in theory, but we found them they remain small during our evaluation.

Second, BESPOKV enables new HPC services for emerging workloads such as deep learning and massive IoT data processing: (1) Data layout: While existing KV solutions are rigid/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, a datalet using B-tree as main data structure is better suited for read-intensive workloads (e.g., deep learning), while Log Structured Merge (LSM) tree based datalet is a better choice for write-intensive workloads due to high write amplification and no fragmentation. (2) Multi-tenancy and geo-distribution: IoT applications increasingly require multi-tenancy support, e.g., smart road big data used by different applications. Different tenant would require different consistency and topologies. Even for a single tenant the topology requirements may change. For example, simple MS topology may be sufficient for sensors deployed in one building but as the scale of deployment increases, AA may become more beneficial. Existing systems do not provide such support. (3) Low latency: deep learning queries require ultra low latency to take advantage of in-memory KV storage. For this purpose, we added support for DPDK kernel bypassing in BESPOKV.

This paper makes the following contributions:

- We propose a novel distributed KV store architecture that follows best architectural practices such as decoupling of control and data planes. Decoupling allows BESPOKV to transparently turn a user-provided (single-server) datalet into scalable, fault-tolerant distributed KV stores. Such specialization will enables innovative uses of KV stores in HPC applications, especially for emerging applications that utilize KV-friendly workloads. 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. As examples, we present a novel mechanism to make transitions from MS+EC to MS+SC, and from AA+EC to MS+EC. 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. Using five (two new and three existing) datalets, We show that with all the aforementioned benefits, BESPOKV-enabled distributed KV stores scale horizontally and performs comparable (and sometimes $1.2\times$ to $2.6\times$ better) to state-of-the-art distributed KV stores.

## 2 CHALLENGES

Several challenges arise when designing BESPOKV to meet the competing goals of compatibility, versatility, modularity, and performance.

*Compatibility.* BESPOKV strives to transparently make a non-distributed KV store into a distributed one. It should be easy to use, such that a developer simply "drops" the non-distributed version of the store into BESPOKV; in turn, BESPOKV will automatically clone and convert the store into various types of highly scalable and reliable distributed clusters.

However, in reality, every datalet is different, resulting in compatibility issues. Moreover, KV stores use different communication protocols. For instance, Redis's protocol is different from Cassandra's. This implies that BESPOKV's communication substrate should be designed to understand the basic message semantics, e.g., request routing. We describe this in Section 3.1.

*Versatility.* Due to the diversity of data storage and retrieval requirements, almost all the points on the cluster topology (MS, AA, etc.), consistency (strong, eventual, etc.), replication, and fault tolerance spectrum are valid. However, existing systems only support a fixed single design point, which limits flexibility and adaptability. Therefore, BESPOKV architecture should be versatile enough to cover various design options, and be flexible to support reconfiguration.

Different storage applications implement their distributed management and protocols with preference on diverse dimensions such as cluster topology and consistency. To support applications with tradeoffs among these different dimensions through a generic framework, one should ensure that each configurable dimension has a clear boundary and well defined interface. Hence, different dimensions can be seamlessly combined with each other to form a highly versatile choice of options for application developers. Moreover, the distributed network architecture should be flexible enough to support these wide range of options. Section 3 presents this aspect of BESPOKV's versatile architecture.

*Modularity.* Building various design options using different implementations is simply a matter of putting in more engineering effort and not as challenging. In fact, such a naive monolithic redesign approach would essentially be similar to the current approach of per-application implementations. Instead, BESPOKV should be designed in a modular fashion, which makes it possible to reuse a previously developed component. For instance, a controlet supporting MS+SC or AA+EC can be reused for multiple backend data stores. Furthermore, the modules in BESPOKV should be expandable to meet the ever-growing needs for advanced features.

*Performance.* Achieving the above goals is the major focus of our work. However, BESPOKV should be realized without

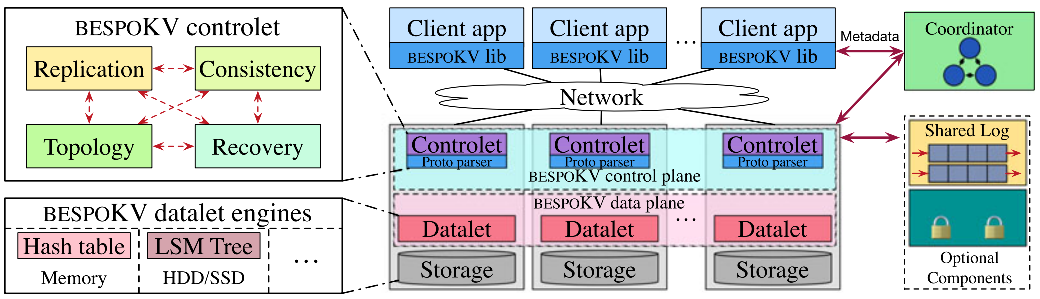


Fig. 2. BESPOKV architecture and the interactions between components. `LSM Tree`: Log-structured merge-tree. `DLM`: Distributed lock manager.

sacrificing performance. Design choices for protocol handling, network architecture, and diverse components should be carefully made with efficiency in mind.

## 3 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. Fig. 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 $N$ $(\geq 1)$ instances of datalets, depending on the processing capacity of the controlet and its datalets, and can leverage physical resource (datacenter) heterogeneity [24], [25] 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 [26] 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 3.2. 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 [27] for DLM, and ZLog [28], [29], [30] for Shared Log) and provides interface libraries (Section 3.2, Table 4).

### 3.1 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 make use of off-the-shelf single-serve data stores such as Redis [18], SSDB [31], 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 Section 8.

*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 3 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 [22] where developers should learn complex new APIs.

TABLE 3
APIs to `Put`, `Get`, and `Del` a KV Pair

| 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` |

*Datalet and Client APIs are for using pre-built controlets.*

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 [32]. 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.

## 3.2 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 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 Section 4.

*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.

*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 [33]; once the popularity of hot keys exceeds a certain predefined 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

TABLE 4
APIs for Events, Shared Log, DLM, and Coordinator for New Controlet Development

| 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` |

*Due to space limitation, we list only important APIs.*

adding a controlet-side small cache or data migration/replication for load balancing purpose [34], [35], [36], [37], [38].

*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 Section 5 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 4.

# 4 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. Using hash-based *tHT* datalet and consistent hashing for the client library as an example, this section presents support for MS+SC, MS+EC, AA+SC, AA+EC and four examples to enable new forms of distributed services by combining existing controlets or extending ones.[2]

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

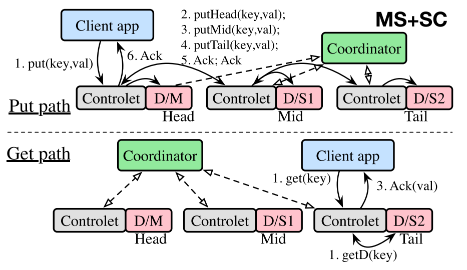


Fig. 3. `Put/Get` paths in MS+SC. `M` means master; `S`$n$ means the $n$th slave; `D` means datalet.

### 4.1 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 [26] at the cost of serialization. However, alternative scalable designs exist such as chain replication (CR) [39], 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 [40], [41] 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 [41], our implementation lets the head respond after it receives an acknowledgment from the tail, given its pre-existing network connection with the client.

*Example.* Fig. 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 [39], [41].

### 4.2 Master-Slave & Eventual Consistency

BESPOKV's pre-built controlet takes a simple approach to support MS+EC where the master copies the data to slaves asynchronously.

*Example.* Fig. 4a 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.[3] Subsequently, BESPOKV provides EC by asynchronously forwarding `Put` requests to other datalets (step 4).

*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.

### 4.3 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 3.2). For performance improvement, optimistic concurrency control [42] 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.* Fig. 4b 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

3. This 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.

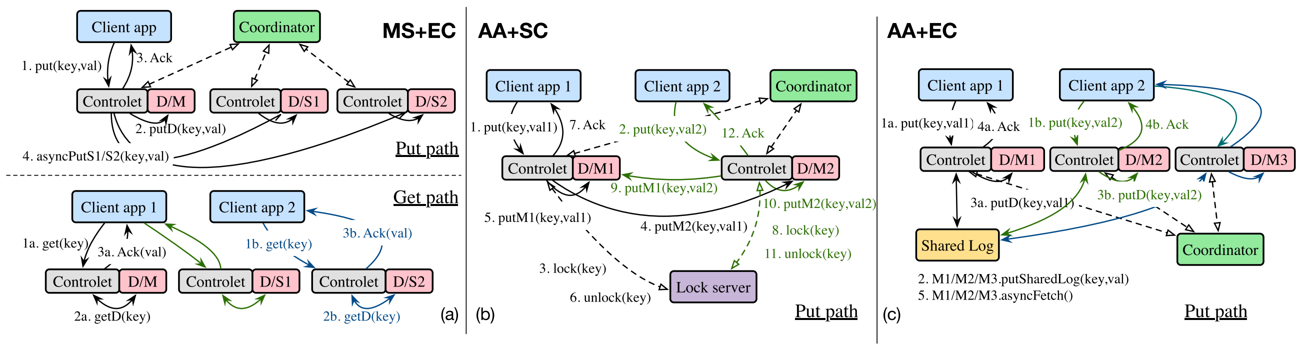


Fig. 4. The `Put/Get` paths in MS+EC (a), AA+SC (b), and AA+EC (c). The `Get` path is same in all three, except in AA+SC, where the difference is that each `Get` needs to acquire a read lock before proceeding. `Mn` means the $n$th master.

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.

### 4.4 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 [43].

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 [44].

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 [45].

*Example*. Fig. 4c 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.

## 5 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 9.4 presents the experimental results on this aspect.

### 5.1 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.

Fig. 5a shows transition from MS+EC to MS+SC.[4] 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.

4. 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.

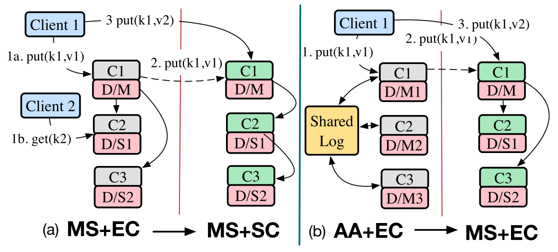


Fig. 5. Transition: MS+EC to MS+SC and AA+EC to MS+EC.

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.

## 5.2 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 propagates the request. The Get requests are not affected. Fig. 5b 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.

## 6 Extensions to KV Stores

BESPOKV is immediately ready-to-use for popular distributed KV store use cases. If desired, BESPOKV's control plane can be extended to enable new forms of distributed services by combining existing controlets or extending ones. This section demonstrates four examples. We evaluated performance of Scan requests (range query) in Section 9.2, and the next two per-request consistency and polyglot persistence in Section 9.5.

### 6.1 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.

### 6.2 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.

### 6.3 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 [46] by launching custom controlets for cross-app lazy synchronization (eventual consistency).

### 6.4 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 [47] to determine the location of keys.

*Variants of AA*. BESPOKV can support Adding routing flexibility to the AA topology is straightforward. Clients can simply send a request to any of the controlets (or use some load balancing techniques such as round robin), which then routes the request to the actual controlet–datalet that holds the requested data. The extra logic we need to add into controlets is a routing map similar to a finger-table [47] to determine the location of keys.

### 6.5 Other Consistency Models

Our Shared Log-based asynchronous fetches for eventual consistency can be easily configured to support bounded staleness. Developers simply specify a $T$-sec polling period, so that clients are guaranteed not to see the stale data for more than $T$ sec. Similarly, causal consistency can also be supported if the controlet serving the Get request fetches all the pending data from the shared log and communicates it to other controlets before replying back to the client.

## 7 BESPOKV's Use Cases

### 7.1 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 [48] and Markov chain model [49] 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. Fig. 6 shows an example of

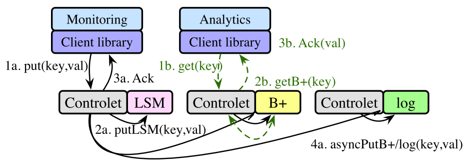


Fig. 6. `Put/Get` paths in MS+EC for HPC monitoring to perform I/O load balancing.

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 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.

### 7.2 Building Burst Buffer File Systems

Burst buffer file systems are becoming an indispensable framework to quickly absorb application I/O requests in exascale computing [50], [51], [52], [53]. 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 [50]. 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 [54]. 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.

### 7.3 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 [55], [56], [57]. 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 9.2). 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.

### 7.4 Resource and Process Management

KV store has also been used to aid the resource and process management in HPC systems [2], [58]. 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 9.2 and 9.4).

## 8 BESPOKV IMPLEMENTATION

Current implementation of BESPOKV consists of $\sim 69k$ 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 [59], a in-memory KV store client library. (3) *Coordinator* uses ZooKeeper [26] 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 [60] and Redlock-based [27]. (5) *Shared Log handler* is implemented using ZLog [28], based on CORFU.

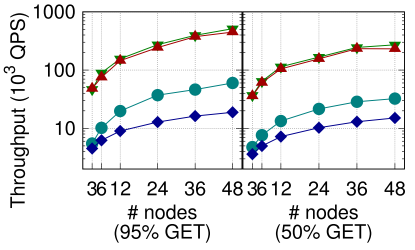
The BESPOKV prototype has four pre-built controlets as described in Section 4. All controlet shares the sample event-handling controlet template of 150 LoC. In addition, BESPOKV supports multiple backend datalets with protocol parsers. Using the common datalet template of 966 LoC, we implemented three new datalets with a Protobuf-based [32] 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 [61] store. In addition, BESPOKV are compatible with existing single-server KV stores SSDB [31] 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 [62] 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.

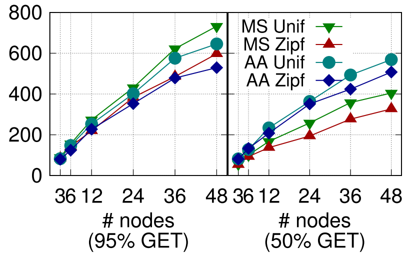
## 9 EVALUATION

Our evaluation answers the following questions:

- Are BESPOKV-enabled distributed KV stores scalable (Section 9.2), adaptive to topology and consistency changes (Section 9.4), and extensible (Section 9.5)?



(a) SC. On log scale.



(b) EC.

Fig. 7. BESPOKV scales *tHT* horizontally.

- How does BESPOKV compare to existing proxy-based (Section 9.6), and natively-distributed (Section 9.7) KV stores?
- How well BESPOKV handles a node failure? (Section 10)

## 9.1 Experimental Setup

*Testbeds and Configuration.* We perform our evaluation on Google Cloud Engine (GCE) and a local testbed. For larger scale experiments (Sections 9.2, 9.3, 9.4, 9.5, and 9.6), 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 9.7) and fault tolerance experiments (Section 10), 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 [26] 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) [63].

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 [64], 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 percent 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.

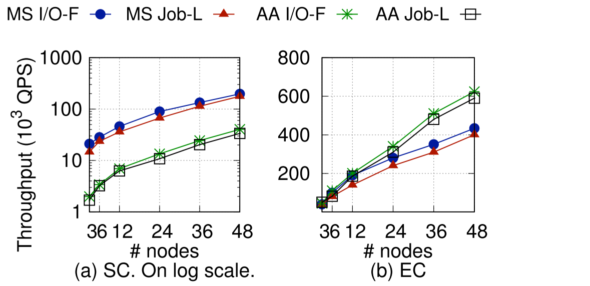


Fig. 8. BESPOKV scales HPC workloads.

For YCSB we use an update-intensive workload (`Get`:`Put` ratio of 50%:50%), a read-mostly workload (95 percent `Get`), and a scan-intensive workload (95 percent `Scan` and 5 percent `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 $\text{Zipfian constant} = 0.99$). The reported throughput is measured in terms of thousand queries per second (kQPS) as an arithmetic mean of three runs.

## 9.2 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. Fig. 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 percent `Get`) and write-intensive (50 percent `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 9.7.

Fig. 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 percent more reads than Job launch with `Get`:`Put` ratio of 62%:38%.

Fig. 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

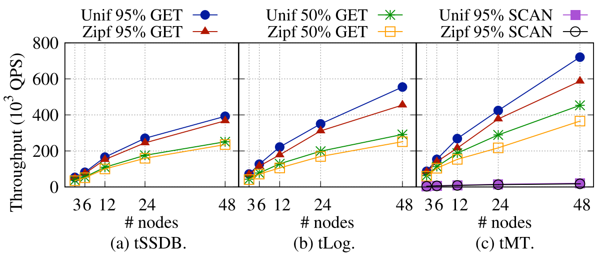


Fig. 9. BESPOKV scales *tSSDB*, *tLog*, and *tMT* with MS+EC.

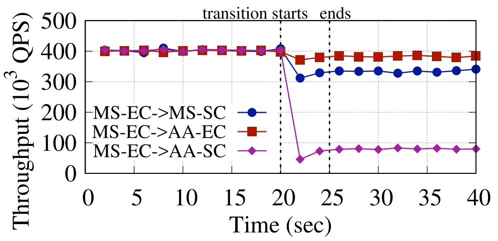


Fig. 11. BESPOKV seamlessly adapts service from MS-EC to MS-SC, AA-EC, and AA-SC.

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 percent `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 [65], [66].

## 9.3 Impact of Varying Replication Factor

Next, we analyze the impact of varying the replication factor on performance. Fig. 10 shows the average throughput of an 8-shard cluster when varying the number of replicas from 1 to 3.

For cluster configurations with EC under read-intensive (uniform and Zipfian 95 percent `Get`) workloads, a larger replication factor results in higher performance. This is because there are more nodes that can serve `Get` requests. The effect is more significant for uniform workload, as the load is more balanced.

On the other hand, for MS+SC, scaling the number of replicas does not improve performance. Performance stops scaling above 2 replicas under read-intensive workloads, and it actually degrades by a factor of 2 to 3 for write-intensive workloads. This is because BESPOKV uses chain replication to support MS+SC, and all `Puts` are going to the head controlets, which need to do more work as the length of the chain increases.

Increasing the number of replicas does increase the performance for AA+SC under read-intensive workloads but with limited improvement. This is due to the high cost of distributed locking. Zipfian workloads severely increase the lock contention at the lock server, leading to the observed performance drop.

## 9.4 Adaptability

We evaluate BESPOKV's adaptability in switching online consistency levels and topology configurations (Section 5). In all the tests we use 3 shards with a Zipfian workload of 95 percent `Get`. As shown in Fig. 11, 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 complete switching in extremely short time compared to existing solutions because BESPOKV does not require data migration or down time.

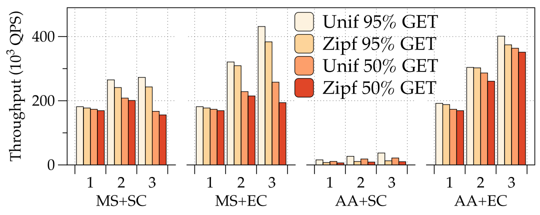


Fig. 10. Varying the number of replicas in BESPOKV.

## 9.5 Extensibility and New Services

As sketched in Section 4, 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 6.2) under MS+SC and a Zipfian workload with a 25:75 percent ratio of SC:EC as the desired consistency. We observed the performance to be between MS+SC and MS +EC as shown in Fig. 7; for example, with 24 nodes, we obtain $\sim 300k$ QPS for 95 percent `Get` and $\sim 270k$ QPS for 50 percent `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 6.3) 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 Figs. 7 and 9; for example, with 24 nodes, we obtain $375k$ QPS for 95 percent `Get` and $200k$ QPS for the 50 percent `Get` workload.

## 9.6 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. We perform each test at three different periods of time to capture interfernce caused by cloud-based multi tenancy.

Fig. 12 shows the throughput. BESPOKV enables new MS +SC ($\sim 500k$ QPS under Zipfian 95 percent `Get`) and AA+EC ($\sim 750k$ QPS under Zipfian 95 percent `Get`) configurations

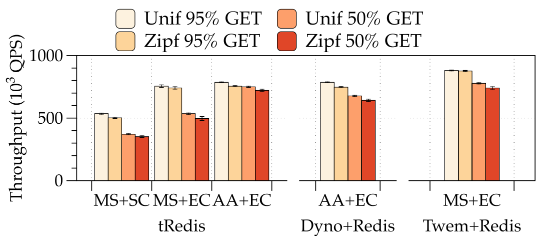


Fig. 12. BESPOKV adds MS+SC and AA+EC for Redis. Comparison with Dynomite (Dyno) and Twemproxy (Twem).

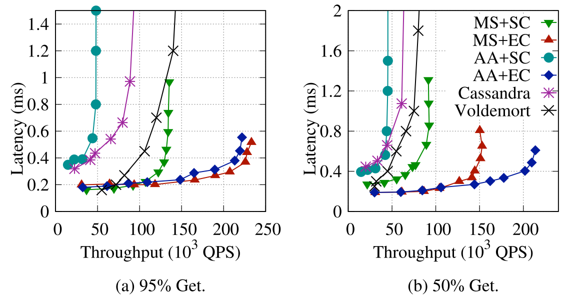


Fig. 13. Average latency versus throughput achieved by various systems under Zipfian workloads.

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.[5] However, we observed the same performance for Dynomite +Redis in supporting AA+EC configuration for Redis as BESPOKV in supporting AA+EC. The small error bars in Fig. 12 show that inherent multi tenancy effect of cloud-based environment is almost negligible.

## 9.7 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 [67]. 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 [68]. 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*.

Fig. 13 shows the latency and throughput for all tested systems/configurations when varying the number of clients

5. Twemproxy itself does not provide any consistency support.

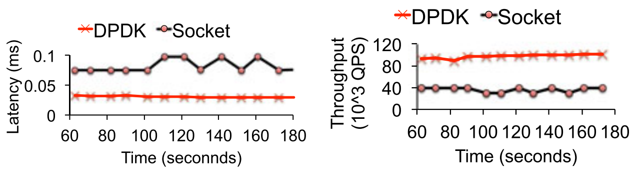


Fig. 14. Latency and throughput improvements by using DPDK.

to increase the throughput in units of kQPS.[6] 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 *tLog* 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 effects the write performance and increases the read latency due to use of extra CPU and disk usage [69]. 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 [69].

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 percent Get workload since both configurations serve Gets from all replicas. AA+EC achieves 47 percent higher throughput than MS+EC under 50 percent 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.

## 9.8 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 versus DPDK based communication. We deployed a single shard on our local testbed (Section 9.1) and measured latency and throughput using YCSB. Each node in our local setup is equipped with Intel ethernet controller X540-AT2. We used Intel's DPDK framework version 17.05. Fig. 14 shows that DPDK reduces latency by up to 65 percent. We also observe 3× improvement in throughput compared to socket based communication. Another interesting finding is that DPDK based communication results in more stable performance.

## 10 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

6. Uniform workloads show similar trend, hence are omitted.

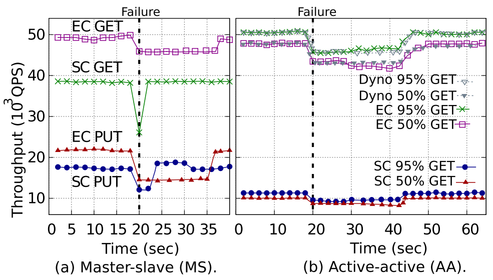


Fig. 15. Throughput timeline on failover. `EC`: eventual consistency; `SC`: strong consistency; `Dyno`: Dynomite.

KV tuples generated with a Zipfian distribution. We intentionally crash a node to emulate a failure, and Fig. 15 shows the resulting throughput change.

*MS topology.* For MS+SC, we bring down the head node under the write-intensive workload (50 percent `Put`, as shown in the bottom half of Fig. 15a) and the tail node for the read-intensive workload (95 percent `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 controlet–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 Fig. 15a 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 2nd-from-last node in the chain the new tail, and updates the topology metadata. Once clients see the update, they reroute the corresponding `Gets` 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 drops throughput by only˜$1/9$.

*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 Fig. 15b, 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 Dynmoite as BESPOKV also uses datalet's callback functions to import and export the data. Please note that users can choose to add more replicas to increase the overall performance so that in case of a single node failure or a topology/consistency switch the performance drop is not significant enough to affect the HPC application.

## 11 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 [70] is a cloud storage system that offers a range of consistency-level SLAs. Some storage systems offer tunable consistency, e.g., ManhattanDB [71]. Flex-KV [72] is another flexible key-value store that can be configured to act as a non-persistent/durable store and operates consistently/inconsistently. Morphus [73] provides support towards reconfigurations for NoSQL stores in an online manner. MOS [74], [75] and hatS [76], [77] provide flexible and elastic resource-level partitioning for serving heterogeneous object store workloads. ClusterOn [78] proposes to offer generic distributed systems management for a range of distributed storage systems. MBal [36], [37] provides fine-grained service-level differentiation via flexible data partitioning. 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 [22] 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 Vsync to further enrich flexibility. EventWave [79] elastically scales inelastic cloud programs. PADS [80] provides policy architecture to build distributed applications. Similarly, mOS [81] 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.

## 12 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. BESPOKV's decoupled control and data plane architecture, configurability, and extensibility enable new solutions for emerging HPC systems and workloads. 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. As examples, we present a novel mechanism to make transitions from MS+EC to MS+SC, and from AA+EC to MS+EC. We also present several use cases to show effectiveness of BESPOKV to support HPC applications. 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

## Acknowledgments

This work was sponsored in part by the US National Science Foundation under the Grants: CNS-1565314, CNS-1405697, CNS-1615411, CCF-1919075, CCF-1919113, and CNS-1814430. This work was performed while Fred Douglis was employed by Dell EMC and gratefully acknowledges Dell EMC's support.

## REFERENCES

[1] J. Kim, S. Lee, and J. S. Vetter, "PapyrusKV: A high-performance parallel key-value store for distributed NVM architectures," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 57.

[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 *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, Art. no. 9.

[3] Z. W. Parchman, F. Aderholdt, and M. G. Venkata, "SharP hash: A high-performing distributed hash for extreme-scale systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 647–648.

[4] S. Eilemann *et al.*, "Key/value-enabled flash memory for complex scientific workflows with on-line analysis and visualization," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 608–617.

[5] MongoDB, Accessed: Aug. 2019. [Online]. Available: https://www.mongodb.com/

[6] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Princ.*, 2007, pp. 205–220.

[7] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, pp. 35–40, 2010.

[8] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.

[9] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," in *Proc. 25th Symp. Operating Syst. Princ.*, 2015, pp. 263–278.

[10] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A distributed, searchable key-value store," in *Proc. ACM SIGCOMM Conf. Appl. Technol., Archit. Protocols Comput. Commun.*, 2012, pp. 25–36.

[11] Social Artisan, Accessed: Aug. 2019. [Online]. Available: http://socialartisan.co.uk/

[12] Behance, Accessed: Aug. 2019. [Online]. Available: https://www.behance.net/

[13] The Migration Process, Accessed: Aug. 2019. [Online]. Available: https://academy.datastax.com/planet-cassandra//mongodb-to-cassandra-migration/#data_model

[14] Why flowdock migrated from cassandra to mongodb, Accessed: Aug. 2019. [Online]. Available: http://blog.flowdock.com/2010/07/26/flowdock-migrated-from-cassandra-to-mongodb/

[15] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.

[16] Twitter's Twemproxy, Accessed: Aug. 2019. [Online]. Available: https://github.com/twitter/twemproxy

[17] Memcached, Accessed: Aug. 2019. [Online]. Available: https://memcached.org/

[18] Redis, Accessed: Aug. 2019. [Online]. Available: http://redis.io/

[19] Netflix's Dynomite, Accessed: Aug. 2019. [Online]. Available: https://github.com/Netflix/dynomite

[20] LevelDB, Accessed: Aug. 2019. [Online]. Available: https://github.com/google/leveldb

[21] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 183–196.

[22] Vsync, Accessed: Aug. 2019. [Online]. Available: https://vsync.codeplex.com/

[23] Google Cloud Platform, Accessed: Aug. 2019. [Online]. Available: https://cloud.google.com/compute/

[24] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2013, pp. 77–88.

[25] J. Mars, L. Tang, and R. Hundt, "Heterogeneity in "Homogeneous" warehouse-scale computers: A performance opportunity," *IEEE Comput. Archit. Lett.*, vol. 10, no. 2, pp. 29–32, Jul.–Dec. 2011.

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2010, Art. no. 11.

[27] Distributed locks with Redis, Accessed: Aug. 2019. [Online]. Available: http://redis.io/topics/distlock

[28] ZLog, Accessed: Aug. 2019. [Online]. Available: https://github.com/noahdesu/zlog

[29] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, "CORFU: A shared log design for flash clusters," in *Proc. 9th USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 1–14.

[30] M. A. Sevilla *et al.*, "Malacology: A programmable storage system," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 175–190.

[31] SSDB, Accessed: Aug. 2019. [Online]. Available: https://github.com/ideawu/ssdb

[32] Google Protocol Buffers, Accessed: Aug. 2019. [Online]. Available: https://developers.google.com/protocol-buffers/

[33] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 13.

[34] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect: Provable load balancing for randomly partitioned cluster services," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, pp. 1–12.

[35] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with SwitchKV," in *Proc. 13th Usenix Conf. Netw. Syst. Des. Implementation*, 2016, pp. 31–44.

[36] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–16.

[37] Y. Cheng, A. Gupta, A. Povzner, and A. R. Butt, "High performance in-memory caching through flexible fine-grained services," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 56.

[38] X. Jin *et al.*, "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 121–136.

[39] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proc. 6th Conf. Symp. Operating Syst. Des. Implementation*, 2004, Art. no. 7.

[40] S. Almeida, J. A. Leitão, and L. Rodrigues, "ChainReaction: A causal+ consistent datastore based on chain replication," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 85–98.

[41] J. Terrace and M. J. Freedman, "Object storage on CRAQ: High-throughput chain replication for read-mostly workloads," in *Proc. Conf. USENIX Annu. Tech. Conf.*, 2009, Art. no. 11.

[42] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.

[43] Cassandra Configuration, Accessed: Aug. 2019. [Online]. Available: http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html

[44] How Dynomite handles the data conflict, Accessed: Aug. 2019. [Online]. Available: https://github.com/Netflix/dynomite/issues/274

[45] R. Van Renesse, D. Dumitriu, V. Gough, and C. Thomas, "Efficient reconciliation and flow control for anti-entropy protocols," in *Proc. ACM Workshop Large-Scale Distrib. Syst. Middleware*, 2008, Art. no. 6.

[46] Polyglot persistence, Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Polyglot_persistence

[47] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 149–160, 2001.

[48] B. Xie *et al.*, "Characterizing output bottlenecks in a supercomputer," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, Art. no. 8.

[49] A. K. Paul *et al.*, "I/O load balancing for big data HPC applications," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 233–242.

[50] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, Art. no. 69.

[51] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "BurstMem: A high-performance burst buffer system for scientific applications," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 71–79.

[52] 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 *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 404–409.

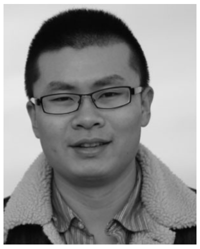
[53] X. Shi, M. Li, W. Liu, H. Jin, C. Yu, and Y. Chen, "SSDUP: A traffic-aware SSD burst buffer for HPC systems," in *Proc. Int. Conf. Supercomput.*, 2017, Art. no. 27.

[54] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. Panda, "A 1 PB/s file system to checkpoint three million MPI tasks," in *Proc. 22nd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2013, pp. 143–154.

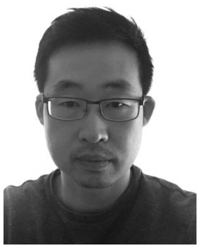
[55] S. Patil and G. Gibson, "Scale and concurrency of GIGA+: File system directories with millions of files," in *Proc. 9th USENIX Conf. File Stroage Technol.*, 2011, Art. no. 13.

[56] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 237–248.
[57] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "DeltaFS: Exascale file systems scale better without dedicated servers," in *Proc. 10th Parallel Data Storage Workshop*, 2015, pp. 1–6.
[58] R. H. Castain, D. G. Solt, J. Hursey, and A. Bouteiller, "PMIx: Process management for exascale environments," in *Proc. 24th Eur. MPI Users' Group Meet.*, 2017, pp. 14:1–14:10.
[59] libmc, Accessed: Aug. 2019. [Online]. Available: https://github.com/douban/libmc
[60] ZooKeeper Recipes and Solutions, Accessed: Aug. 2019. [Online]. Available: https://zookeeper.apache.org/doc/r3.1.2/recipes.html
[61] Embedded Masstree, Accessed: Aug. 2019. [Online]. Available: https://github.com/rmind/masstree
[62] Kubernetes: Production-Grade Container Orchestration, Accessed: Aug. 2019. [Online]. Available: https://kubernetes.io/
[63] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[64] SeaweedFS, Accessed: Aug. 2019. [Online]. Available: https://github.com/chrislusf/seaweedfs
[65] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 29.
[66] N. Zhao *et al.*, "Chameleon: An adaptive wear balancer for flash clusters," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 1163–1172.
[67] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project Voldemort," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, Art. no. 18.
[68] N. Carvalho *et al.*, "Finding consistency in an inconsistent world: Towards deep semantic understanding of scale-out distributed databases," in *Proc. 8th USENIX Conf. Hot Topics Storage File Syst.*, 2016, pp. 66–70.
[69] Why not Cassandra, Accessed: Aug. 2019. [Online]. Available: http://www.dynomitedb.com/docs/dynomite/v0.5.6/faq/
[70] 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 *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 309–324.
[71] Manhattan, our real-time, multi-tenant distributed database for Twitter scale, Accessed: Aug. 2019. [Online]. Available: http://goo.gl/7EThfo
[72] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini, "Flex-KV: Enabling high-performance and flexible KV systems," in *Proc. Workshop Manage. Big Data Syst.*, 2012, pp. 19–24.
[73] M. Ghosh, W. Wang, G. Holla, and I. Gupta, "Morphus: Supporting online reconfigurations in sharded NoSQL systems," *IEEE Trans. Emerg. Topics Comput.*, vol. 5, no. 4, pp. 466–479, Fourth Quarter 2017.
[74] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, "Taming the cloud object storage with MOS," in *Proc. 10th Parallel Data Storage Workshop*, 2015, pp. 7–12.
[75] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, "MOS: Workload-aware elasticity for cloud object stores," in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2016, pp. 177–188.
[76] K. Krish, A. Anwar, and A. R. Butt, "hatS: A heterogeneity-aware tiered storage for hadoop," in *Proc. 14th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2014, pp. 502–511.
[77] K. Krish, A. Anwar, and A. R. Butt, "[phi] Sched: A heterogeneity-aware hadoop workflow scheduler," in *Proc. IEEE 22nd Int. Symp. Modelling Anal. Simul. Comput. Telecommun. Syst.*, 2014, pp. 255–264.
[78] A. Anwar, Y. Cheng, H. Huang, and A. R. Butt, "ClusterOn: Building highly configurable and reusable clustered data services using simple data nodes," in *Proc. 8th USENIX Conf. Hot Topics Storage File Syst.*, 2016, pp. 51–55.
[79] 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 *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–16.
[80] N. M. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm, "PADS: A policy architecture for distributed storage systems," in *Proc. 6th USENIX Symp. Netw. Syst. Des. Implementation*, 2009, pp. 59–73.
[81] M. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mOS: A reusable networking stack for flow monitoring middleboxes," in *Proc. 14th USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 113–129.

**Ali Anwar** received the PhD degree in computer science from Virginia Tech, Blacksburg, Virginia. He is currently a research staff member with IBM Almaden Research Center. In his earlier years, he worked as a tools developer (GNU GDB) at Mentor Graphics. His research interests include distributed computing systems, cloud storage management, file and storage systems, AI platforms, and intersection of systems and machine learning.



**Yue Cheng** received the PhD degree in computer science from Virginia Tech, Blacksburg, Virginia, in 2017. He is currently an assistant professor of computer science with George Mason University. His research interests include distributed systems, cloud computing, and high-performance computing.



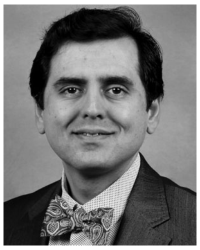
**Hai Huang** received the BSE degree in computer science and engineering (CSE) from the Ohio State University, Columbus, Ohio, in 2000, and the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, Michigan, in 2006. He is a currently research staff with the Cloud Computing Department, IBM Research. His research interests include cloud computing, operating systems, distributed systems management, software testing, and anomaly detection.

**Jingoo Han** is currently working toward the PhD degree with the Department of Computer Science, Virginia Tech, Blacksburg, Virginia. In his earlier years, he has worked as a senior software engineer with Samsung Electronics. His research interests include distributed systems, deep learning, and high-performance computing.

**Hyogi Sim** received the BS degree in civil engineering and the MS degree in computer engineering from Hanyang University, Seoul, South Korea, and the MS degree in computer science from Virginia Tech, Blacksburg, Virginia, in 2014 and is currently working toward the PhD degree at Virginia Tech, Blacksburg, Virginia. He joined Oak Ridge National Laboratory in 2015, as a post-masters associate. During this appointment, he conducted research and development on active storage systems and scientific data management for HPC systems. He is currently an HPC systems engineer with Oak Ridge National Laboratory. His primary role is to design and develop a checkpoint-restart storage system for the exascale computing project. His areas of interest include storage systems and distributed systems.

**Dongyoon Lee** received the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, Michigan. He is currently an assistant professor of computer science with Stony Brook University, where he works on computer systems, software reliability, program analysis, concurrency, security, computer architecture, and software engineering. From 2014 to 2019, he was an assistant professor of computer science with Virginia Tech. He has won a Distinguished Paper Award at FSE 2018 and a Best Paper Award at ASPLOS 2011. His SC 2016 paper was nominated as a Best Student Paper Finalist. He has received a Virginia Tech ICTAS Junior Faculty Award in 2017, a Google Research Award in 2015, a ProQuest Distinguished Dissertation Award in 2013, and a VMWare Graduate Fellowship in 2011. His co-authored papers won the best student paper finalist at SC 2016, and the best paper at ASPLOS 2011.

**Fred Douglis** (Fellow, IEEE) received the PhD degree in computer science from the U.C. Berkeley, Berkeley, California. He is a chief research scientist with Perspecta Labs since January 2018, where he works on applied research in the areas of blockchain, network optimization, and security. He was previously with companies including Matsushita, AT&T, IBM, and (Dell) EMC. His research interests included storage, distributed systems, web tools and performance, and mobile computing. He is a member of the IEEE Computer Society Board of Governors.



**Ali R. Butt** is currently a professor of computer science with Virginia Tech. He is a recipient of an NSF CAREER, IBM Faculty Awards, NetApp Faculty Fellowships, and a VT COE Faculty Fellowship. He has served on the editorial board of the *IEEE Transactions on Cloud Computing*, *ACM Transactions on Storage*, and *IEEE Transactions on Parallel and Distributed Systems*. He is an alumni of the NAE FOE and NAS AA symposia (2010 organizer). His research interests include scalable distributed computing systems, cloud computing, edge computing, file and storage systems, and Internet-of-Things. At Virginia Tech, he leads the Distributed Systems & Storage Laboratory (DSSL).