ClusterOn: Building Highly Configurable and Reusable Clustered Data Services using Simple Data Nodes

Ali Anwar Yue Cheng Hai Huang[†] Ali R. Butt
Virginia Tech [†]*IBM Research – T.J. Watson*

Abstract

The growing variety of data storage and retrieval needs is driving the design and development of an increasing number of distributed storage applications such as key-value stores, distributed file systems, object stores, and databases. We observe that, to a large extent, such applications would implement their own way of handling features of data replication, failover, consistency, cluster topology, leadership election, etc. We found that 45–82% of the code in six popular distributed storage applications can be classified as implementations of such common features. While such implementations allow for deeper optimizations tailored for a specific application, writing new applications to satisfy the ever-changing requirements of new types of data or I/O patterns is challenging, as it is notoriously hard to get all the features right in a distributed setting.

In this paper, we argue that for most modern storage applications, the common feature implementation (i.e., the *distributed* part) can be automated and offloaded, so developers can focus on the core application functions. We are designing a framework, ClusterOn, which aims to take care of the *messy plumbing* of distributed storage applications. The envisioned goal is that a developer simply "drops" a non-distributed application into ClusterOn, which will convert it into a scalable and highly configurable distributed application.

1 Introduction

The big data boom is driving the development of innovative distributed storage systems aimed at meeting the increasing need for storing vast volumes of data. We examined the number of representative storage systems that have been implemented/released by academia in the last decade, and found a steady increase in such systems over recent years. Figure 1 highlights the trend of innovating new solutions for various and changing storage needs.[1]

These new storage systems/applications[2] share a set of features such as replication, fault tolerance, synchronization, coordination, and consistency. This implies that a great portion of these features are overlapped across various such applications. Furthermore, implementing a new application from scratch imposes non-trivial engineering efforts in terms of numbers of lines of code (LoC) or person-year. Table 1 gives the LoC[3] of 6 popular distributed storage applications in three categories. Correspondingly, Figure 2 shows the LoC breakdown. An interesting observation is that all of the six applications have a non-trivial portion of LoC (45.3%–82.2%) that implements common functionalities such as distributed management. While LoC is a major indicator for the engineering effort involved, it is by no means definitive or comprehensive enough. The engineering effort required at different stages of development also includes the fundamental difficulties of bundling management components as well as increased maintenance cost (e.g., bug fixing) as the codebase size increases. It would be highly desirable and efficient if the common features across various storage applications can be *"reused"*.

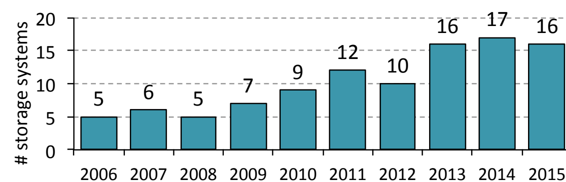


Figure 1: Number of storage systems papers in SOSP, OSDI, ATC, and EuroSys conferences in the last decade (2006–2015).

	Key-value store			Object store		DFS
	Redis	HyperDex	Berkeley DB	Swift	Ceph	HDFS
Total #LoC	41760	20691	159547	43375	187144	112510

Table 1: Total LoC in the 6 studied storage applications.

One may argue that different storage solutions are developed to meet certain needs, and thus are specialized. In this paper, however, we posit that storage management software, not the application developers, should implement all the *messy plumbings* of distributed storage applications. We argue that there is a strong need for such a modularized framework that provides a thin layer to realize the common functionalities seen in distributed storage applications. On one hand, such a framework will significantly reduce the complexities of developing a new storage application. On the other hand, a modularized

[1] We only count papers that implement a full-fledged storage system. USENIX FAST is not included as it solely covers storage. Given the very large number of systems available in the industry, and general lack of details about the inner workings, we only studied systems proposed by academia.

[2] We use "systems/applications" interchangeably throughout.

[3] We count the LoC on a per-file basis, excluding source code of the client side and the testing framework.

```
1  void Put(Str key, Obj val) {
2    if (this.master):
3      Lock(key)
4      HashTbl.insert(key, val)
5      Unlock(key)
6    Sync(master.slaves)
7  }
8
9  Obj Get(Str key) {
10   if (this.master)
11     Obj val = Quorum(key)
12   Sync(master.slaves)
13   return val
14 }
15
16 void Lock(Str key) {
17   ... // Acquire lock
18 }
19
20 void Unlock(Str key) {
21   ... // Release lock
22 }
23
24 void Sync(Replicas peers) {
25   ... // Update replicas
26 }
27
28 void Quorum(Str key) {
29   ... // Select a node
30 }
```
(a) Vanilla

```
1  void Put(Str key, Obj val) {
2    if (this.master):
3      zk.Lock(key) // zookeeper
4      HashTbl.insert(key, val)
5      zk.Unlock(key) // zookeeper
6    Sync(master.slaves)
7  }
8
9  Obj Get(Str key) {
10   if (this.master)
11     Obj val = Quorum(key)
12   Sync(master.slaves)
13   return val
14 }
15
16 void Sync(Replicas peers) {
17   ... // Update replicas
18 }
19
20 void Quorum(Str key) {
21   ... // Select a node
22 }
```
(b) Zookeeper-based

```
1  #include <vsynclib>
2
3  void Put(Str key, Obj val) {
4    if (this.master):
5      zk.Lock(key) // zookeeper
6      HashTbl.insert(key, val)
7      zk.Unlock(key) // zookeeper
8    Vsync.Sync(master.slaves)
9  }
10
11 Obj Get(Str key) {
12   if (this.master)
13     Obj val = Vsync.Quorum(key)
14   Vsync.Sync(master.slaves)
15   return val
16 }
```
(c) Vsync-based

```
1  void Put(Str key, Obj val) {
2    HashTbl.insert(key, val)
3  }
4
5  Obj Get(Str key) {
6    return HashTbl(key)
7  }
```
(d) ClusterOn-based

Table 2: An example of different approaches to developing a distributed KV store. 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.

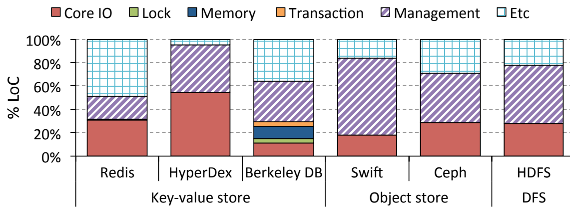


Figure 2: LoC breakdown of the 6 studied storage applications. `Core IO` component includes the core data structure and protocol implementation. `Management` component includes implementations of replication/recovery/failover, consistency models, distributed coordination and metadata service. `Etc` includes functions providing configurations, authentications, statistics monitoring, OS compatibility control, etc. `Management` and `Etc` are the components that can be generalized and implemented in ClusterOn.

framework enables more effective and simpler service differentiation management on the service provider side.

Table 2 shows 4 snippet implementations of the core functions for a simple key-value (KV) store. To implement everything from scratch (`(a) Vanilla`), the application developer has to create his own concurrency control functionality (`Lock()`, `Unlock()`), and consistency and quorum management logic (`Sync()`, `Quorum()`). Using a mature distributed coordination system such as Zookeeper [9] (`(b) Zookeeper-based`) reduces the LoC from 30 to 22, implying reduction of the overall developer efforts. Similarly, the third option (`(c) Vsync-based`) calls a library such as Vsync [3] that provides coordinating actions to support KV storage (DHT storage), etc. By linking and interfacing with the library, the engineering effort is further reduced. One caveat of using a system such as Zookeeper or a library such as Vsync is that, the developers needs to familiarize themselves with these systems or libraries to use them appropriately in their own application code.

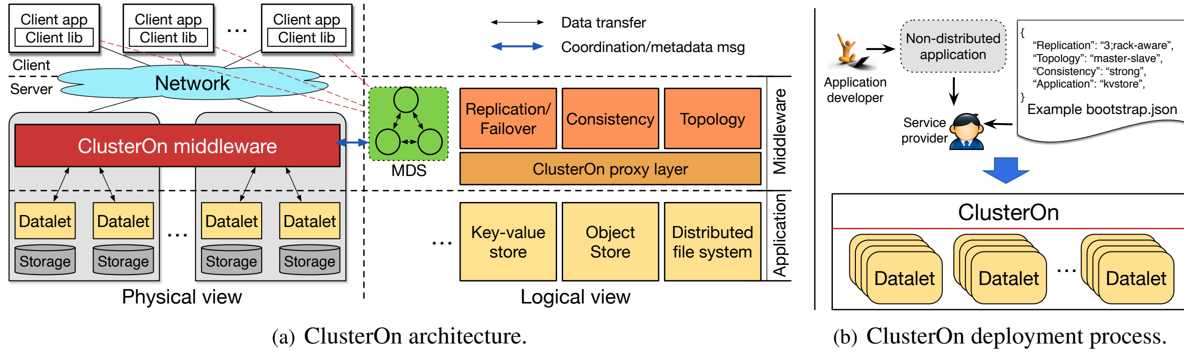
We envision a framework where *developers only need to implement the needed core functionality*, and common features/management etc. is automatically provided, akin to User Defined Functions in the popular MapReduce framework. To this end, we propose ClusterOn, a framework that takes a non-distributed core version (which we call *datalet*) of a storage application, adds common features and management, and finally morphs the code into a scalable distributed application. Table 2(d) (`(d) ClusterOn-based`) gives a simplified example of the ClusterOn usecase. ClusterOn is motivated by the observations we made earlier that modern storage applications share a great portion of functionality. By providing a transparent and modularized distributed framework that provides configurable services such as replication, fault tolerance, and consistency, ClusterOn hides the inherent complexities of developing distributed storage applications.

This paper makes the following contributions. We quantitatively analyze various distributed storage application and show that they have repetitive source code that implements common features found across these applications. As a novel solution, we present the design of ClusterOn, which provides these features to reduce the engineering effort required for development of new distributed applications.

2 Related Work

Products related to Mesos [1] and Kubernetes [2] ecosystem such as Chronos[4] and Marathon[5] let applications

[4] https://github.com/mesos/chronos
[5] https://github.com/mesosphere/marathon



(a) ClusterOn architecture.

(b) ClusterOn deployment process.

Figure 3: ClusterOn architecture. MDS: metadata server. Figure 3(b) shows an example about how to use ClusterOn for service deployment. Service provider deploys the non-distributed application developed by application developer in ClusterOn based on a JSON formatted bootstrap configuration file.

scale horizontally by running more instances (Chronos is a job scheduler that supports complex job topologies whereas Marathon handles hardware or software failures to ensures that an app is "always on"). ClusterOn differs in that it provides rich replication and consistency support, which can handle more complex parallelism patterns. Zookeeper [9] has been extensively used for distributed coordination. EventWave [5] elastically scales inelastic programs in clouds. We expect ClusterOn to have a *more profound* effect than the above systems. Rather, ClusterOn focuses on simplifying the development of storage systems by providing general support of distributed management. Application developers simply offload components such as replication, consistency, and coordination to ClusterOn. Furthermore, ClusterOn is completely transparent in that it does not require developers to familiarize themselves with any APIs, hence, further reducing the development complexities.

A large body of research has been focused on improving the performance of some kind of data-intensive applications by enhancing the design of underlying data storage and coordination software that provide services such as replication, consistency, and fault tolerance [4, 10, 12, 13]. However, unlike ClusterOn, such works do not provide a generic framework for supporting a vast variety of storage applications.

Vsync [3] is a library for building scalable high-assurance services and can be used for replication of data in the cloud. Unlike ClusterOn, to use Vsync, application developers have to learn and work with the Vsync API to derive its full benefits (as shown in Table 2). Coign [8] automatically partitions a program into a distributed setup by performing static code analysis and binary-level partitioning. In contrast, ClusterOn is non-intrusive and more scalable, as it seamlessly scales out a non-distributed application instead of partitioning it.

3 Design Goals

Minimize overhead The key to realizing ClusterOn is to minimize the framework overhead. ClusterOn leverages a middleware-/proxy-based architecture design to perform traffic forwarding and distributed management. While simplifying the application development and lowering down the maintenance cost, ClusterOn should not incur too much overhead, which in turn offsets the benefits it brings. This leads to the need of utilizing effective optimization techniques such as DPDK[6] and RDMA[7], which we leave as future work.

More effective service differentiation By modularizing key components and synthesizing them as a holistic middleware, ClusterOn, from the storage service provider's perspective, should be able to provide more effective service-level differentiation such as performance QoS and tunable consistency level automating.

Reusable distributed storage platform Novel optimizations and techniques [7, 12, 13] have been proposed for existing storage applications. As these applications do not share source code, it requires non-trivial engineering efforts to port these improvements from one application to another. We envision ClusterOn will be flexible and extensible enough to support such user-defined features as pluggable modules, so that a vast variety of applications can benefit.

4 ClusterOn Design

Figure 3(a) shows the architecture of ClusterOn. It consists of three major components—(1) application layer, (2) middleware, and (3) a metadata server.

The ***application layer*** includes a homogeneous cluster of *datalets*. A datalet is a single instance of the application (an example datalet is shown in Table 2(d)). These datalets are the building blocks of constructing larger distributed applications, although they are completely unaware that they are running in a distributed setting. Datalets are designed to run on a single node, and they are only responsible for performing the core functions of an application. For example, a KV store datalet, in the simplest form, only needs to implement a `Get` and a `Put` interface. When running a single datalet is no longer

[6] http://dpdk.org.

[7] http://www.mellanox.com/page/products_dyn?product_family=79.

sufficient to handle the load, we can instantiate more of them. However, for them to work coherently with one another, the middleware layer is needed to perform the required coordination.

The ***middleware layer*** has 2 main responsibilities: (1) manages cluster topology, and (2) depending on cluster topology, coordinates network traffic amongst the datalets. For scalability reasons, the middleware layer needs to have a distributed set of entities (we call them *proxies*) to perform these 2 functions well on behalf of a distributed set of datalets. In the simplest design, one could have a one-to-one mapping between a proxy and a datalet that are symbiotically co-located with all traffic going into and out of the datalet proxies. Proxies will communicate with one another depending on the relationship between the datalets that they proxy within the cluster topology. For example, in a master-slave topology, the master's proxy will forward all write requests to the slave's so the data can be replicated appropriately. In most cases, a single proxy can handle N instances of datalets (where $N \geq 1$, and N is configurable depending on the processing capacity of both the proxy and datalet).

Different applications use different protocols for their communication. ClusterOn's proxies parse the application messages to make decisions such as routing. To support new applications, there are two basic options: (1) the HTTP-based REST protocol; and (2) Google's protocol buffer. Due to the cost that these two solutions may incur, performance-sensitive applications can select to use a simple yet generic text/binary-based protocol, which is also supported by ClusterOn.

The ***meta-data server*** is used by the middleware layer as a persistent store for keeping critical meta-data about the cluster, e.g., topology information, current load on datalets, so the network traffic can be appropriately routed, data can be recovered from a failure, or datalets autoscaled when appropriate. Clients to the cluster can also consult the meta-data server for directory services. This allows clients to more efficiently direct their request to nodes within the cluster across a wide variety of topologies. However, this is optional, as proxies can always redirect requests within themselves to route to the destination datalet or datalets.

4.1 Middleware Components

A major challenge in designing ClusterOn is to make sure it can cover various types of distributed applications, e.g., KV stores, object stores, distributed file systems, databases, etc. Though these applications are distributed and share similar modules, they are very different in nature. Hence, there is a need for a systematic approach to classify these applications into different categories and make sure that ClusterOn supports all those categories. ClusterOn realizes this by classifying and supporting these applications on the basis of their underlying replication schemes, cluster topology, and consistency model.

Replication module Different applications adopt different replication schemes, e.g., SPORE [7] replicates data at granularity of KV pairs and Redis at node level. File systems such as GFS [6] and HDFS replicate block-level data chunks within and across racks to provide better fault tolerance. ClusterOn's replication module supports a 2-dimensional configuration space: (1) replication granularity, e.g., key/shard/node-level, and (2) replication locality, e.g., server/rack/datacenter-local. ClusterOn provides a generic module that allows flexible selection of different replication schemes, thus covering a wide variety of storage application use cases.

Cluster topology manager Cluster topology manager specifies the logical relationship among holders (datalets) or different replicas. Generally, most distributed storage applications can be divided in three types of cluster topologies, namely, (1) master-slave, (2) multi-master (or active-active), (3) peer-to-peer (P2P), or some combinations of these. ClusterOn's cluster topology manager supports the above three topologies. Master-slave mode provides chain replication support [11] by guaranteeing that only one datalet is acting as the master and all others as slaves while keeping this fact oblivious to these datalets. Similarly, in the case of active-active topology, multiple datalets can be concurrently write to and read from, and this can be coordinated by the corresponding proxies using distributed locks. In case of P2P topology, ClusterOn controls the management logic that drives the inter-node communication among the peers. As consistent hashing is commonly used in a P2P topology, ClusterOn can easily calculate who are the immediate neighbors of a datalet for data propagation and recovery purposes.

Consistency module The role of consistency module is to support three consistency levels that are commonly adopted by the modern distributed storage applications, namely, (1) strong consistency, (2) eventual consistency, and (3) no consistency. Consistency module is built on top of Zookeeper and provides support to acquire lock at granularity of KV pair, block, object, file, or node level. In the case of strong consistency, each incoming request to access data is executed after acquiring a lock on the data. Eventual consistency is supported by first acquiring lock only on the primary copy of the data and updating the secondary copies later.

5 Preliminary Evaluation

We are in the process of prototyping ClusterOn using C++, and are adding support for different kinds of storage applications (object stores and distributed file systems). In this section, we present our preliminary evaluation of a basic multi-threaded version of ClusterOn used

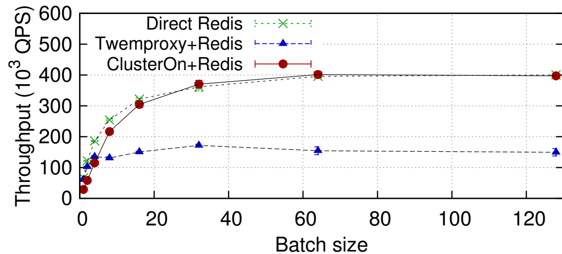


Figure 4: ClusterOn overhead with Redis based datalet. We configure Redis as a memory cache. We run 100% GET workload with 10 million key-value tuples (16 B keys and 32 B values), as the batch size is increased until it eventually saturates Redis. A single thread is run in ClusterOn to offer a fair comparison with the single-threaded twemproxy.

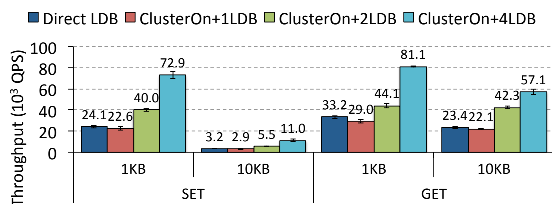


Figure 5: Scaling up LevelDB (LDB) based datalet using ClusterOn. We interface ClusterOn with LevelDB by implementing a simple wrapper around LevelDB and a text protocol supporting the basic GET and SET commands. We run 100% GET and SET workloads with 1 million key-value tuples (value size of 1 KB and 10 KB). LevelDB datalets are scaled from 1 instance to 4, which persist data on an SSD. ClusterOn is configured to run 4 threads so it does not become the bottleneck. `Direct LDB` means direct access to LevelDB instances via network.

to realize a distributed version of two different applications: an in-memory cache Redis—to quantify the overhead; and an embedded NoSQL database LevelDB—to test the basic functionality of scaling up a non-distributed datalet storage application. We perform experiments on a relatively small setup. We run ClusterOn co-located with storage applications as datalets on a single 32 core machine with 64 GB of DRAM and a SATA SSD. We run benchmark clients on a different machine with the same hardware specifications connected via a 10 GbE network. Each data point is the average of three runs.

Figure 4 quantifies the overhead of ClusterOn by comparing it with a baseline case where clients directly talk with the Redis server, and twemproxy that is a state-of-the-art Redis proxy. We observe that when batching 64 and 128 requests, ClusterOn outperforms twemproxy by 1.66×. Note that this is a comparison of data forwarding feature only, and does not include batch splitting and hashing etc., which are comparable under both ClusterOn and twemproxy. More importantly, ClusterOn's throughput approaches that of the `Direct Redis` setup with negligible overhead.

In our next test we leverage ClusterOn to scale up a LevelDB-based datalet application. As shown in Figure 5, ClusterOn is able to linearly scale up the LevelDB performance (the 10 KB GET workload hits the SATA SSD's effective random read bandwidth, hence the sub-linear throughput growth).

The results are promising and show the potential of ClusterOn in providing a lightweight middleware for serving high performance storage applications.

6 Conclusion

We have shown that distributed storage applications share a great portion of common functionalities which can be generalized and abstracted. We propose ClusterOn, a modularized framework that provides a thin layer to realize these functionalities so that the development engineering effort can be significantly reduced. Preliminary results show that ClusterOn incurs negligible overhead and demonstrate its ability to scale up an embedded NoSQL database. Adding distributed management support is the focus of our ongoing research.

Acknowledgments We thank our shepherd, Indrajit Roy, and reviewers for their feedback. This work was supported in part by NSF grants CNS-1405697 and CNS-1422788.

References

[1] Apache Mesos. http://mesos.apache.org/.

[2] Kubernates. http://kubernetes.io/.

[3] Vsync. https://vsync.codeplex.com/.

[4] Bonvin, N., Papaioannou, T. G., and Aberer, K. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *ACM SOCC'10*.

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

[6] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *ACM SOSP'03*.

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

[8] Hunt, G. C., Scott, M. L., et al. The coign automatic distributed partitioning system. In *USENIX OSDI'99*.

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

[10] Porto, D., Leitao, J., Li, C., Clement, A., Kate, A., Junqueira, F., and Rodrigues, R. Visigoth fault tolerance. In *ACM EuroSys'15*.

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

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

[13] Zhang, Y., Power, R., Zhou, S., Sovran, Y., Aguilera, M. K., and Li, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *ACM SOSP'13*.