Fast key-value stores: An idea whose time has come and gone

Atul Adya, Robert Grandl, Daniel Myers
Google

Henry Qin
Stanford University

Abstract

Remote, in-memory key-value (RInK) stores such as Memcached [6] and Redis [7] are widely used in industry and are an active area of academic research. Coupled with stateless application servers to execute business logic and a database-like system to provide persistent storage, they form a core component of popular data center service architectures. We argue that the time of the RInK store has come and gone: their domain-independent APIs (e.g., PUT/GET) push complexity back to the application, leading to extra (un)marshalling overheads and network hops. Instead, data center services should be built using stateful application servers or custom in-memory stores with domain-specific APIs, which offer higher performance than RInKs at lower cost. Such designs have been avoided because they are challenging to implement without appropriate infrastructure support. Given recent advances in auto-sharding [8, 9], we argue it is time to revisit these decisions. In this paper, we evaluate the potential performance improvements of stateful designs, propose a new abstraction, the linked, in-memory key-value (LInK) store, to enable developers to easily implement stateful services, and discuss areas for future research.

CCS Concepts • **Computer systems organization** → Distributed Systems; Key/Value Stores; Stateful Architectures;

Keywords Distributed Systems, Key-Value Stores, Caches

ACM Reference format:
Atul Adya, Robert Grandl, Daniel Myers and Henry Qin. 2019. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of Workshop on Hot Topics in Operating Systems, Bertinoro, Italy, May 13–15, 2019 (*HotOS '19), 7 pages.
https://doi.org/10.1145/3317550.3321434

1 Introduction

Modern internet-scale services often rely on remote, in-memory, key-value (RInK) stores such as Redis [7] and Memcached [6] (Fig. 1). These stores serve at least two purposes. First, they may provide a cache over a storage system to enable faster retrieval of persistent state. Second, they may store short-lived data, such as per-session state, that does not warrant persistence [25].

These stores enable services to be deployed using *stateless application servers* [16], which maintain only per-request state; all other state resides in persistent storage or in a RInK store. Stateless application servers bring operational simplicity: for example, any request may be handled by any application server, which makes it easy to add and remove application servers, to handle server failures, and to handle skewed workloads.

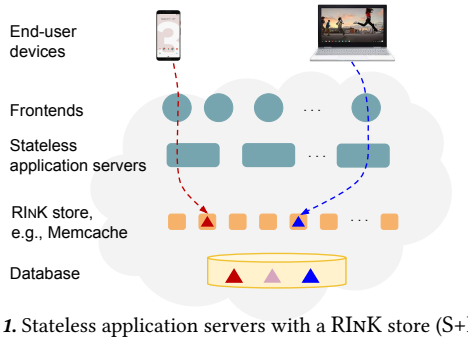


Figure 1. Stateless application servers with a RInK store (S+RInK).

A key property of RInK stores is that they provide a simple and *domain-agnostic* interface (e.g., PUT/GET of string keys and string values, or manipulation of simple data structures such as lists). Because RInK stores handle well-defined and simple operations, implementations have been able to achieve extraordinarily high throughput and low latency [14, 18, 20, 24]. Moreover, the high performance of these stores makes distributed caching challenges such as load balancing relatively less important: even when a workload exhibits skew, simply overprovisioning the RInK store can be viable.

On the other hand, the domain-agnostic interfaces of RInK stores push cost and complexity back to the application servers. For example, they force applications to repeatedly convert their internal data structures between native language representations and strings, which imposes both a CPU cost and a programmer burden. This problem is exacerbated when applications do not use the entirety of every value retrieved from a RInK store, since bytes are needlessly transferred only to be discarded. Finally, the network distance imposes a latency cost, particularly when large values must be transferred or multiple round trips are needed.

We argue that these costs are under-appreciated and no longer necessary, given recent improvements in auto-sharding

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321434

systems [8, 9]. Rather than externalizing in-memory state in a RInK, developers should instead build stateful application servers. By a stateful application server, we mean a server that couples application code and long-lived in-memory state *in the same process*. If a stateful application server is not feasible, e.g., because state is shared by multiple applications or languages, developers should instead build a custom in-memory store, which is at a network distance from application servers but exposes a domain-specific API.

Building stateful services requires solving new technical challenges, and the research community should focus on helping developers solve them, rather than on building ever-faster RInK stores. Although they pose challenges, stateful services offer significant performance improvements. For example, ProtoCache (a component of a widely-used Google application) saw a 40% reduction of 99.9% latency when it made this architectural switch, and experiments with a model application and synthetic workloads show potential latency improvements of up to 57%.

This paper makes three contributions. First, we argue that coupling application code and cached in-memory state brings underappreciated performance benefits. Second, we propose a new linked, in-memory key-value (LInK) store abstraction. A LInK store is a key-to-rich-object map linked into application servers or custom in-memory stores that replaces an external RInK store; it implements functionality, such as reconfiguration after a resharding event, whose absence we have found to impose a burden on developers. To conclude, we describe additional areas in which the research community can make contributions.

Finally, we are concerned with improving application performance when accessing in-memory state by eliminating RInK stores. Questions of how to ensure persistence of data, while important, are outside the scope of this paper.

2 Motivation

2.1 Stateless and RInK Architecture

Modern internet-scale services need to be highly available and reliable. One of the most important principles for robustness is simplicity, and service writers aim to architect their systems as simply as possible so that they are easy to implement, debug, maintain, and operate.

As far back as the late 1990s, developers were converging on stateless application servers, where multiple independent processes in a data center handle requests without maintaining any state that outlives a request (the "LAMP stack"). The state needed to handle each request is obtained ad-hoc from a storage system, such as a SQL database or a horizontally partitioned distributed storage system such as HBase [2], Cassandra [1], Bigtable [12], or Spanner [13]. This design maximizes the simplicity of the application server, which contains only business logic; in particular, it contains no code to manage state.

The purely stateless design yields operational benefits because every application server is equivalent. If the load on the service increases, new servers can be added to absorb the additional load. If a server crashes, requests can be directed to the remaining servers. If a server becomes overloaded, load can easily be shed, e.g., by directing calls to the least-loaded server or by using the power-of-two approach [22].

On the other hand, accessing persistent storage on every request is expensive and slow. Services may also have state that is inappropriate to persist but that nonetheless must outlive individual requests (e.g., session state). To handle these demands, developers evolved the stateless architecture to include a layer of RInK stores such as Memcached [6] and Redis [7] (Fig. 1). We call this the *Stateless + RInK* architecture, or S+RInK. The expectation is that the RInK: (1) improves scalability by offloading work from the persistent store, since the RInK handles most requests; (2) lowers the latency, because it does not need to provide durability.

Moreover, in some cases, a RInK store provides a shared cache between multiple different applications, which improves overall efficiency by avoiding duplication of data in a per-application caches.

Finally, RInK stores are more scalable than (most) application servers, since application servers often execute complex logic and RInK stores have a straightforward, well-defined interface. High performance allows RInK stores to be sharded with simple techniques (e.g., load-unaware consistent hashing in Memcacheg). Given a lack of auto-sharding infrastructure, isolating sharding to the RInK store would be attractive.

2.2 Stateless and RInK Architecture Limitations

The S+RInK architecture attempts to provide the best of both worlds: to simultaneously offer both the implementation and operational simplicity of stateless application servers and the performance benefits of servers caching state in RAM. We believe the architecture falls short due to fundamental limitations in the abstraction it offers.

We identify the domain-agnostic API of RInK stores as the flaw that hinders end-to-end application performance. The limited power of such APIs forces applications to incur unnecessary resource costs and increased latency.

2.2.1 CPU cost due to (un)marshalling

When reading data from a RInK store, a stateless application server must perform a conversion step to translate that data into an application object. For example, a calendar application might construct a structured representation of events and attendees. This operation takes CPU time. Simply receiving the bytes from the network may also require CPU, although we do not consider this cost here. For example, in ProtoCache prior to its rearchitecture, 27% of latency was due to (un)marshalling. In our experiments (Section 3), (un)marshalling accounts for more than 85% of CPU usage. We also found (un)marshalling a 1KB protocol buffer to cost

over 10us, with all data in the L1 cache. A third-party benchmark [5] shows that other popular serialization formats (e.g., Thrift [27]) are equally slow.

We see (un)marshalling as fundamental to real-world applications. Abstract data types are useful [21], and their internal representation is unlikely to correspond to a wire format. For example, it may include variable-length fields or pointers. Second, as software evolves, fields may be added and removed from the data structure stored in the RInK, which argues different wire and in-memory formats. Finally, if programs written in different languages consume the same cached data, some (un)marshalling is unavoidable.

RInK API introduces significant CPU overhead due to data conversion into domain-specific representations.

2.2.2 Overreads

An overread occurs when an application reads data from a RInK store in excess of what was required, resulting in extra CPU and network costs. This happens when an application must fetch an entire value, even if only a fraction of it is useful for the corresponding computation. For example, consider an application that caches the address books for all online users in a RInK store: for each user, the address book is stored as a single key-value pair. If a request arrives from Alice to read Bob's phone number from her address book, all of Alice's contacts must be fetched and unmarshalled, even though only a small portion of the data is needed.

Decomposing large key-value pairs into multiple key-value pairs can reduce the degree of overread, but at the cost of weaker consistency guarantees: most RInK stores do not support atomic cross-key operations. Additionally, devising a decomposed representation to accelerate rich operations (e.g., find the first free time slot for a user in a calendar application) is not obviously trivial. Finally, the more the schema is optimized to support particular operations, the harder it is to evolve as requirements or workloads change. Overreads can also sometimes be mitigated with richer data models in the store [7], but as we discuss below, such solutions lack generality and flexibility.

As a real-world example, for a common type of operation in ProtoCache, at the 99th percentile, responses are only 2% of the size of the total cached item: avoiding overreads is important. These operations are implemented using a collection of indices, which means that there is no single decomposition of cached items into multiple key-value pairs that would work for all operations.

Finally, our experiments in Section 3 show that if only part of the record is needed (10%), RInK stores incur both extra CPU (46%) and network (85%) costs.

RInK causes unnecessary data processing and transfer.

2.2.3 Network Latency

Even fast data center networks can impose latency costs in the stateless and RInK architecture. If an application requires multiple reads or writes to service a request, these latencies can quickly add up. Beyond the simple RTT, the size of the data to be transferred also matters, which is a particular problem when coupled with overreads. For example, prior to its rearchitecture, ProtoCache incurred an 80 ms latency penalty simply to transfer large records from a remote store, despite a high speed network.

Low latency data center networks do not eliminate data transfer overheads when using a RInK store.

2.3 State-of-the-art RInK Stores

Although many approaches have been proposed to improve RInK implementations, none of them address the challenges described above.

2.3.1 Basic RInK stores

Basic RInK stores offer an interface that is effectively only a PUT/GET API. For example, Memcached [6] is one of the original and best-known RInK stores; it has been broadly deployed, including in Facebook's Tao [11] and Dropbox's Edgestore [4]; a version is also widely used at Google. Beyond PUT/GET, it includes some limited additional operations, such as appending a value to an existing value and incrementing an existing integer value.

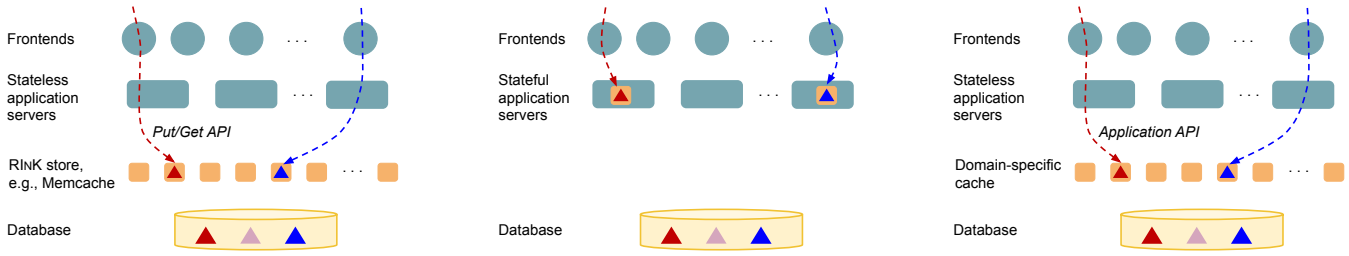
Recently, the academic community has focused on building more performant basic RInK stores, as well as optimizing Memcached to death [10, 15, 26, 28]. FaRM [14] attempts to improve the performance of stateless applications by reducing the cost of fetching remote data using RDMA. NetCache [18] addresses load skew for in-memory stores with an extra layer of caching for hot keys directly in the network switch. KV-Direct [20] leverages programmable NICs to bypass the remote CPU and achieve high throughput access to remote memory while offering richer semantics than traditional RDMA.

These systems focus on improving the performance of the PUT/GET operations rather than addressing the problems of (un)marshalling costs or overreads.

2.3.2 Extended RInK stores

Extended RInK stores offer a richer domain-agnostic API than the basic RInK stores.

Gribble et al. [17] proposed building a small set of cluster-scale distributed data structures. As considered here, this is similar to Redis [7], which also provides data structures that enable finer-grained access to stored data but requires the application to model its objects using one of the available data structures. Redis can also be extended using modules of arbitrary C code, and Splinter [29] allows applications to ship small fragments of code to the store. These extensions are in the spirit of the custom in-memory stores for which we advocate. We argue for taking this approach even further, embedding the store into the application server when possible, and for providing a richer set of features, such as dynamic load balancing and replication.



(a) Stateless servers with RInK *(b)* Stateful servers *(c)* Stateless servers with domain-specific caches

Figure 2. Different in-memory architectures. In each case, an auto-sharding system (not shown) could be incorporated.

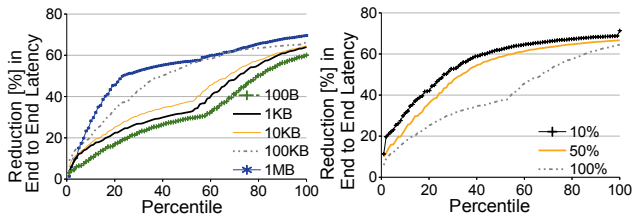
3 Eliminating RInK stores

We argue that RInK stores (Fig. 2a) should not be used when implementing scalable data center services. In this section, we describe how to eliminate RInK stores from existing architectures by presenting two standard architectures for stateful services. In Section 4, we describe how to make these architectures easier to implement.

3.1 Stateful application servers

Stateful application servers couple full application logic with a cache of in-memory state linked into the same process (Fig. 2b). This architecture effectively merges the RInK with the application server; it is feasible when a RInK is only accessed by a single application and all requests access a single key. For example, a contacts application server might cache user address books and directly accept HTTP requests to render them in a web UI.

This architecture eliminates the problems with RInK discussed in Section 2. Since the cache stores application objects, there is no (un)marshalling overhead or network latency. Similarly, overreads are eliminated because the application can directly access only the required portions of its objects; for example, for small modifications to a large object, an expensive read-modify-write operation can be replaced by a cheap local modify operation.



(a) *(b)*

Figure 3. CDF of improvement in end to end latency using stateful application servers in place of a S+RInK for (a) different object sizes (with no overreads) and (b) various overread percentages (for $10KB$ objects).

To quantify these benefits, we ran experiments comparing S+RInK to stateful application servers. We used 5 clients, 5 servers of each type, and a deployment model as in Fig. 2a. Each workload ran for 2 hours and had 80% read and 20% write operations with a total throughput ranging from 6KB/s to 250MB/s. We measured the reduction in resource consumption (CPU and bytes transferred) and end-to-end latency of the two architectures. We performed experiments for different object sizes and overread percentages.

Figs. 3a, 3b, and Table 1 show our results. The stateful approach is superior to S+RInK in terms of per request/response latency and resource utilization:

- Latency is 29% to 57% better (at the median), with relative improvement increasing with object size (Fig. 3a);
- Decreased overreads lead to lower latency and resource utilization (Fig. 3b, Table 1);

	Object Size					Overread Percentages		
	$100B$	$1KB$	$10KB$	$100KB$	$1MB$	90%	50%	0%
CPU	43%	43%	44%	41%	48%	46%	45%	44%
Network	46%	51%	50%	50%	51%	85%	69%	50%

Table 1. Stateful application servers vs S+RInK: Average percent reduction of resources for different object sizes (with no overreads) and various overread percentages (for $10KB$ objects).

3.2 Custom in-memory stores

Custom in-memory stores are a separate, in-memory cache with a domain-specific interface (Fig. 2c). This architecture incurs an additional network hop relative to a stateful application server, which means that it only alleviates, rather than eliminates, the problems with a RInK store. For example, it can still reduce unmarshalling overheads and overreads using a domain-specific API. On the other hand, in exchange for the network hop, it allows a single cache to be shared across multiple applications and languages.

For example, a calendar service might cache user calendars and expose a *find-free-slots(user, date)* API to find the free timeslots for a particular user on a given day, which would be less expensive than requiring an application to fetch the entirety of a user's calendar.

There are several reasons that developers often believe a RInK store to be essential. We briefly enumerate some of them and describe how a custom in-memory store can replace a RInK while providing equivalent or better performance.

Fanout: Stateless application servers often read many keys from a RInK while handling a single logical operation. A custom in-memory store can also support fanout, but a domain-specific API can reduce overreads, e.g., using aggregation or

```
class Linklet<V> {
  // Converts between objects and strings.
  class Marshaller {
    virtual string Marshall(const V& v) = 0;
    virtual V Unmarshall(const std::string& s) = 0;
  };
  Linklet(std::unique_ptr<Marshaller> m);

  // Caller owns returned value if non-null.
  std::unique_ptr<V> Get(const string& key);

  // Resharding may move object after call returns.
  void Commit(const string& key,
              std::unique_ptr<V> value);
};
```

Figure 4. LINKLET API to access in-memory rich objects.

filtering. For example, in order to schedule a meeting with several participants, a calendar application might issue many *find-free-slots* RPCs to fetch candidate meeting times for each participant.

Sharing: A RINK store is often used to provide a cache shared by multiple different applications, either to enable cross-application integration (such as showing a calendar in an email client) or to avoid duplicating data in separate caches. A custom in-memory store can also fill this role, again with possibly better performance.

Resource Disaggregation: Service owners may prefer isolating CPU- and memory-heavy parts of their workload in different processes (i.e., an application server and a RINK) so that they can be provisioned separately. To the extent that a RINK store enables more efficient provisioning, so too can a custom in-memory store.

In summary, a stateful application server or a stateless application server with a domain-specific cache will *always* offer equal or better latency, efficiency, and scalability than a RINK based approach, since these architectures can trivially mimic it. In particular, these architectures reduce or eliminate (un)marshalling overheads, overreads, and network delays. However, RINK stores are popular in part because they are easy to adopt. To make it easy for service writers to achieve the benefits of stateful architectures, we propose a new store abstraction, as described next.

4 LINK store: Raising the Abstraction

This section presents the LINK store abstraction, first motivating it by describing requirements unmet by an auto-sharder.

4.1 Auto-Sharder: Necessary but Insufficient

An *auto-sharding* system [8, 9] is a necessary building block for stateful application servers: without the ability to react to server failures and to changes in workload (e.g., hot keys), deploying stateful application servers at scale is not practical. However, our experience with dozens of internal customers of Slicer [9] at Google over five years suggests that an auto-sharding system that simply assigns application keys to servers leaves important problems unsolved.

In particular, an auto-sharder is concerned with partitioning *keys*, but applications must handle *values*. For example, when the assignment of a key changes from one server to another, an auto-sharder does not move the associated value. The server newly assigned the key must recover the value, by reading it from a storage system which impacts tail latency. If the value was not present in persistent storage (e.g., session state), then it is lost, impacting the user experience.

Additionally, applications that require both replication of keys across multiple servers (e.g., for load or availability reasons) and consistency of the associated values (either strong or eventual) must build such functionality themselves: the auto-sharder only handles assignments of keys, not the manipulation of their values. Finally, applications may need to keep cached state fresh with respect to some underlying data store; an auto-sharder does not help here either.

To address these application needs, we propose a new abstraction, which by analogy with RINK we call a LINK store, for linked in-memory key-value store. We have built a prototype, and one production team is currently implementing an application using it. In the rest of this section, we describe the LINK store.

4.2 LINK Store

A LINK store is a high level abstraction over an auto-sharder that provides a distributed, in-memory, key-value map with rich application objects as values, rather than strings or simple data structures. As a high level abstraction, it provides richer functionality than an auto-sharder. In particular, we consider the following features desirable:

- *Data Consistency.* A LINK store may provide consistency of data across multiple replicas, which improves handling of hot keys (which can be served from multiple replicas). Data consistency might be strong or eventual; in contrast, an auto-sharder on its own provides *no* data consistency.
- *High availability.* A LINK store may provide high availability of data, e.g., by replication. This decreases the likelihood of state loss after server failures, although a LINK store does not provide persistence guarantees.
- *Resharding support.* A LINK store transparently responds to changes from the auto-sharder by relocating values to the servers newly assigned the keys. This prevents resharding events from causing state loss, improving application performance.
- *State loss notifications.* Servers may fail, causing state to be lost (since, for performance, persistence of state is not guaranteed). To allow applications to detect state loss, a LINK store informs applications when state may have been lost after resharding.
- *Freshness.* A LINK store may automatically detect changes in an underlying data store and invalidate its entries, improving data freshness.

4.3 API and Architecture

The architecture of a LInK store is shown in Fig. 5. It relies on an auto-sharder to direct requests from application clients to application servers, using a router library linked into clients. Application servers link a new library, called the Linklet, which is the LInK store implementation; it encapsulates all server-side interactions with the auto-sharder. When the auto-sharder reshards, Linklet instances running on different application servers exchange application state over RPC.

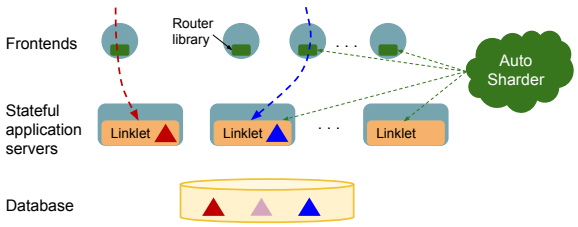


Figure 5. A stateful architecture using a LInK store.

The Linklet exposes a new API that provides references to mutable application objects. We show the essential properties of this API in Fig. 4. A full API would include additional surface area, e.g., to notify applications of state loss, which is beyond the scope of this paper.

An important aspect of this API is that the stored value is a native application object (template parameter V). To enable the Linklet to transfer values in response to resharding events, the application must provide the code to (un)marshall its objects. Developers must write (un)marshalling code when using a RInK store, so this API does not add an additional burden. Unlike as with a RInK store, the marshaller is not on the critical path of every request.

5 Open Problems and Opportunities

In this section, we analyze stateful architectures from two points of view. First, we consider challenges that remain in providing first-class support for stateful architectures using LInK stores, arguing that these are both areas for research contributions and also not intractable. Second, we mention applications that LInK stores can enable.

5.1 Open Problems

Load balancing algorithms: Stateful architectures require load balancing algorithms that operate across diverse applications and workloads. Slicer has one such algorithm, but remaining open problems include: balancing multiple metrics simultaneously, modeling the cost of resharding the application, and rapidly identifying and reacting to abrupt changes in load without causing oscillations. We believe applying control theory would be novel and effective.

Replication: Replication is an important technique to achieve high availability. In RInK stores, replication can be applied to the stored data and isolated from the application logic. In LInK stores, close coupling of application code and data makes the problem more complicated. State machine replication [19, 23] using logical operations requires deterministic application code (which is difficult to guarantee), whereas using physical operations require marshalling objects, imposing a cost that the LInK store sought to avoid. Determining how to address these conflicting goals is an open problem.

Minimizing application footprint: The LInK implementation proposed above relies on linking a significant amount of functionality into the application itself, which has two drawbacks. First, it makes supporting multiple languages harder; second, it makes fixing LInK implementation bugs more difficult, since developers must release new binaries. Determining how much of the LInK architecture could be extracted into a separate, RPC-distance control service is an open question.

5.2 Opportunities

Faster Serverless: *Serverless computing* offers developers the abstraction of a function that executes in response to events (e.g., AWS Lambda [3]). LInK stores could enable high-performance implementations of functions that retain state across invocations.

Context and Personalization: Applications dependent on per-user state (e.g., dialog-based applications like Google Home, Amazon Alexa, etc) need conversational context to answer queries. Not all of this state must be persisted. For example, if server failures and state loss are rare, it might make sense to keep state such as the last-asked question in a LInK store, while keeping longer-term state (such as a per-user voice recognition model) in persistent storage.

6 Conclusion

Industry-standard architectures evolved following decisions made 15-20 years ago, prior to the advent of high-quality, general-purpose auto-sharding systems. We have argued that too much effort has been invested in building fast key-value stores to improve the performance of this architecture, and that industry should move to architectures based on stateful application servers or custom in-memory stores.

Stateful architectures offer higher performance by avoiding unnecessary network and (un)marshalling costs, at the expense of higher demands on infrastructure software. To address these demands, we have proposed the LInK store and described areas for future research.

Acknowledgments

We thank Mark Goodman for converting ProtoCache to a stateful architecture and Bart Locanthi for helping productionize our LInK store prototype. We would also like to thank Eric Brewer, Jeremy Elson, Sanjay Ghemawat, John Ousterhout, Seo Jin Park, and Mendel Rosenblum for their valuable feedback that helped improve the paper.

References

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

[2] Apache HBase. http://hbase.apache.org.

[3] AWS Lambda. https://aws.amazon.com/lambda/.

[4] Edgestore. https://blogs.dropbox.com/tech/2016/08/reintroducing-edgestore.

[5] JVM Serializers. https://github.com/eishay/jvm-serializers/wiki.

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

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

[8] Ringpop. https://ringpop.readthedocs.io/en/latest.

[9] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *OSDI*, 2016.

[10] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *OSDI*, 2014.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *ATC*, 2013.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[14] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *NSDI*, 2014.

[15] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, 2013.

[16] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *SOSP*, 1997.

[17] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *OSDI*, 2000.

[18] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *SOSP*, 2017.

[19] L. Lamport. The part-time parliament. In *TOCS*, 1998.

[20] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: high-performance in-memory key-value store with programmable nic. In *SOSP*, 2017.

[21] B. Liskov. The power of abstraction - (invited lecture abstract). In *DISC*, 2010.

[22] M. Mitzenmacher. The power of two choices in randomized load balancing. In *TPDC*, 2001.

[23] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *ATC*, 2014.

[24] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The ramcloud storage system. In *TOCS*, 2015.

[25] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, 2010.

[26] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: core-aware thread management. In *OSDI*, 2018.

[27] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 2007.

[28] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *ATC*, 2012.

[29] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia. Splinter: Practical private queries on public data. In *NSDI*, 2017.