DMap: A fault-tolerant and scalable distributed data structure

Samuel Benz
Università della Svizzera Italiana (USI)

Fernando Pedone
Università della Svizzera Italiana (USI)

***Abstract*—Major efforts have been spent in recent years to improve the performance, scalability and reliability of distributed systems. In order to hide the complexity of designing distributed applications, many proposals provide efficient high-level communication abstractions (e.g., atomic multicast). These abstractions, however, are often unfamiliar to average application designers and, as a result, implementing distributed applications that tolerate failures and scale performance without sacrificing consistency remains a challenging task. In this paper, we introduce DMap, a reliable and scalable distributed ordered map. DMap fully implements the generic Java SortedMap interface and can be easily used to scale existing Java applications. To substantiate our claim, we have used DMap to turn H2, a centralized database, into a scalable and reliable data management system.**

I. Introduction

Most modern cloud services are distributed systems. Today's on-demand computing resources, common in public cloud environments, provide operators of these systems with the possibility to provision as many servers as needed by the service and to react quickly to changes in application workload. Starting up new servers once increased traffic is detected and shutting down low utilized servers to save costs are common operations. While it is relatively easy to reconfigure stateless components (e.g., application servers), dynamically provisioning stateful components (e.g., storage) is complicated. Consequently, major research efforts have been spent in recent years to improve the performance, scalability and reliability of distributed data stores. The abstractions resulting from these efforts, however, are often unfamiliar to average application designers. As a result, implementing applications that support strong consistency, elastic scalability and efficient recovery remains challenging.

Scalable state machine replication has been shown to be a useful approach to solving the challenges involved in building reliable distributed data stores (e.g., [1]–[3]). State machine replication [4], [5] simplifies the problem of implementing highly available linearizable services by decomposing the *ordering* of requests across replicas from the *execution* of requests at each replica. Requests can be ordered using atomic multicast, a communication abstraction, and service developers can focus on the execution of requests, which is the aspect most closely related to the service itself. State machine replication requires the execution of requests to be deterministic, so that when provided with the same sequence of requests, every replica will evolve through the same sequence of states and produce the same results.

Despite the clear separation of concerns provided by state machine replication (i.e., ordering and execution), designing and implementing a fully functional system from atomic multicast primitives is still complex and unnatural to application designers. In this short paper, we contend that higher-level abstractions, in the form of distributed data structures, can hide this complexity. For example, given a distributed B-tree, services like distributed databases [6] or file systems [7] can be implemented in a distribution transparent manner. We discuss the design and some of the main challenges to implement a distributed ordered map as a ready-to-use data structure.

Existing distributed data structures often rely on transactions or distributed locking to allow concurrent access. Consequently, operations may abort, a behavior that must be handled by the application. We implemented a distributed ordered map (DMap) that does not rely on transactions or locks for concurrency control. Relying on atomic multicast, all partially ordered operations succeed without ever aborting. Additionally, DMap is scalable, fault-tolerant and supports consistent long-running read operations (e.g., to allow background data analytics).

DMap is a distributed sorted key-value store that implements the full Java SortedMap and the ConcurrentMap interfaces. It is generic in that it allows arbitrary Java objects as keys and values. For example, one can define a SortedMap that uses integer objects as keys and string objects as values or a map that uses string objects as keys and holds other complete Java maps as values. DMap also supports user generated objects, as long as they are Java serializable.

DMap achieves scalability through *sharding* (i.e., hash partitioning) and fault tolerance through *replication* (i.e., each shard is replicated by a group of replicas using state machine replication [3]). Moreover, DMap ensures strong consistency in the form of *linearizability*.[1] Consequently, DMap can be used to distribute any local Java application that uses a SortedMap (or Map) by simply replacing the interface implementation.

The paper makes the following contributions. First, we propose a lock-free distributed ordered map with strong multi-shard consistency guarantees that implements the Java SortedMap interface. Second, we detail the implementation of

[1] A concurrent execution is linearizable if there is a sequential way to reorder the client operations such that: (1) it respects the real-time semantics of the objects, as determined in their sequential specs, and (2) it respects the order of non-overlapping operations among all clients [8].

DMap and highlight the underlying replication and ordering techniques. Finally, we provide a performance assessment of all these components. DMap is available as opensource.[2]

II. System model and assumptions

We assume a distributed system composed of a set of interconnected client and server processes that communicate by message passing. Processes may fail by crashing and subsequently recover, but do not experience arbitrary behavior (i.e., no Byzantine failures). Processes are either *correct* or *faulty*. A correct process is eventually operational "forever" and can reliably exchange messages with other correct processes. In practice, "forever" means long enough for processes to make some progress (e.g., terminate one instance of consensus). The servers are divided into groups of replicas, where in each group there is a majority of non-faulty processes.

Our protocols ensure safety under both asynchronous and synchronous execution periods. The FLP impossibility result [9] states that under asynchronous assumptions consensus cannot be both safe and live. To ensure liveness, we assume the system is *partially synchronous* [10]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous, called the *Global Stabilization Time (GST)* [10], is unknown to the processes. Before GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After GST, such bounds exist but are unknown.

III. The DMap distributed data structure

In this section, we present dynamic atomic multicast, a fundamental abstraction used by DMap (§III-A), detail the main technical challenges in the design of DMap (§III-B) and discuss how DMap handles failures (§III-C).

A. Dynamic Atomic Multicast

Dynamic atomic multicast is a communication abstraction that creates the notion of *message streams*. There are primitives for a server to multicast a message m to a stream S, *multicast*(S, m), and for replicas to deliver a multicast message, *deliver*(m). A group of replicas G can subscribe to and unsubscribe from one or more streams with primitives *subscribe_msg*(G, S) and *unsubscribe_msg*(G, S), respectively. After replicas subscribe to stream S, they will eventually deliver messages multicast to S. Similarly, if replicas unsubscribe from S, they will eventually stop delivering messages multicast to S.

Dynamic atomic multicast guarantees strong reliability and order properties. Let relation $<$ be defined such that $m < m'$ iff there is a server that delivers message m before message m'. Dynamic atomic multicast ensures that (i) if a server delivers m, then all non-faulty servers that subscribe to S deliver m *(reliability)*; (ii) if a non-faulty server multicasts m in S then all non-faulty servers that subscribe to S deliver m *(liveness)*; and (iii) relation $<$ is acyclic *(order)*. The order property implies that if servers p and q deliver messages m and m', then they deliver them in the same order.

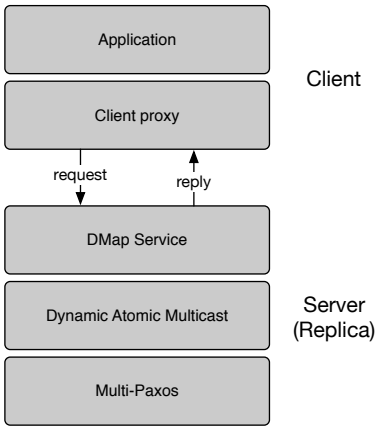

Fig. 1. Architecture overview of DMap.

Dynamic atomic multicast is implemented by combining multiple instances of Multi-Paxos, one Multi-Paxos instance per message stream [11], [12]. If a group G subscribes to multiple streams, then replicas in G combine the various messages multicast to the subscribed streams using a deterministic merge mechanism. For example, one simple merge consists in delivering messages from the subscribed streams in round-robin fashion (i.e., one message from stream S_1, one message from S_2, ..., and again one message from S_1, and so on). Moreover, dynamic atomic multicast accounts for different stream rates by allowing replicas to deterministically skip messages in a stream [11].

B. Basic execution and challenges

In DMap, there is one group of replicas per shard. A *partition map*, stored by all replicas and cached by clients, stores the mapping of shards to replica groups. Under normal operation, each replica group subscribes to two message streams: one stream that is specific to the replica group and one global stream that is common to all replica groups. The group-specific stream is used to order commands within a group of replicas and the global stream is used to order commands across all groups.

Conceptually, DMap's execution model is quite simple. Applications, by means of a client proxy, contact one replica of the DMap service and then submit commands to this replica (see Fig. 1). The client proxy finds a DMap replica by contacting a ZooKeeper[3] service. Upon contacting the replica, the client proxy downloads the current partition map. Communication between the application and the client proxy uses the Apache Thrift RPC framework.[4]

The replica contacted by the client proxy propagates commands for execution to one or more groups of replicas using atomic multicast. Commands that involve a single shard (e.g., *put(K,V)*) are multicast in the message stream specific to the

[2]https://github.com/sambenz/URingPaxos

[3]https://zookeeper.apache.org/

[4]https://thrift.apache.org/

replica group in charge of the involved shard. Commands that involve multiple shards (e.g., *size()*) are multicast in the global message stream. Upon delivering a command, the replicas execute the command and return the result to the replica initially contacted by the client proxy. The client proxy assembles the results and returns them to the application.

Three aspects challenge DMap's simple design:

a) Coping with outdated partition maps: A replica group makes progress at the pace of the fastest majority of its servers. Therefore, a client may contact a replica that is outdated. To ensure that commands reflect the latest commands executed by the system (needed for linearizability), despite outdated client caches and outdated replicas, the partition map is *versioned*. Commands include the partition map version. If a server delivers a command with an outdated partition map version, it notifies the client, which should install the most recent partition map returned by the server. Modifications to the system that lead to changes in the partition map must be multicast to the global stream, which will be delivered by all replicas and ordered with respect to all commands.

b) Consistency and performance of iterators: To ensure the ordered delivery of entries while iterating over sharded data, DMap clients proceed as follows: First, they request a globally consistent in-memory snapshot of all shards. Second, they stream the snapshot in parallel from every shard, a few entries at a time. Third, they deliver to the application the lowest entry of all shards until all entries are delivered. This procedure allows to iterate over a huge amount of data, since only few entries must be kept in memory simultaneously.

The key to implementing such efficient and consistent iterators over hash-partitioned data is the ability to create multi-shard snapshots. Creating such snapshots is complicated since servers do not share a common clock [13]. DMap relies on atomic multicast to create in-memory snapshots at the replicas. Atomic multicast, as described previously, allows to send partially, or in this case totally, ordered commands to be executed at every replica.

c) Consistency of multi-shard commands: In the case of multi-shard commands (e.g., *size()*), replicas must coordinate the execution to ensure linearizability [2]. Without coordination, single-shard and multi-shard commands can interleave in ways that may violate linearizability, despite the commands being consistently ordered. We use the technique proposed in [2] to guarantee linearizable execution of commands. In brief, to avoid undesired command interleaving, upon executing a multi-shard command, a replica must exchange signal messages with at least one replica from every other replication group involved in the command. For example, when executing the *size()* command, each replica determines the local database size and then forwards a signal to the servers in all replica groups involved in the command. Every server waits for such a signal from every involved replication group to finish the execution of the size command.

C. Recovering from failures

DMap can be configured to use stable storage (e.g.. a harddisk) or main memory only (e.g., battery-backed DRAM). With stable storage, each replica periodically checkpoints its state onto stable storage. Upon resuming from a failure, the replica retrieves and installs the last stored checkpoint and recovers the commands missing in this checkpoint (i.e., the commands executed after the replica's last checkpoint) from Paxos.

In main memory mode, a recovering replica will first subscribe to all required multicast groups. Dynamic Atomic Multicast ensures that after subscribing to all streams, the message ordering is guaranteed. Followed by requesting the most recent partition map and a snapshot of the current data on all replicas. To download and install these checkpoints, a recovering replica behaves like a DMap client. The version of the partition map and the snapshot id are the unique values of the Paxos instance in which the commands are decided. Therefore, the recovering replica can skip learned commands before the snapshot id and start applying commands with ids right after the snapshot. To finish recovery, a replica adds itself to the system partition map. After this point, clients will start sending command to the recovered replica.

IV. Experimental evaluation

In this section, we experimentally assess some of the main aspects of DMap: scalable performance (§IV-A), recovery (§IV-B) and repartitioning (§IV-C). All the experiments were performed in a cluster of 10 HP SE1102 servers, equipped with 2x 2.5 GHz Intel Xeon CPUs and 8 GB of main memory. These servers were interconnected through a HP ProCurve 2910 switch with 1 Gbps interfaces. The round trip time between the nodes is 0.1 millisecond (ms). In all the experiments, clients and servers were deployed on separate machines, and servers use in-memory storage. We keep the machines approximately synchronized by running NTP.

A. Scalability

1) Throughput and latency: In this experiment 60 clients per shard send *put()* commands to random keys in a closed loop. The overall system runs on approximately 75% of its maximum throughput. The values are strings of 380 bytes each. We use up to three shards. Every shard is served by three replicas running on one server each.

Results. Figure 2 (left) shows the overall throughput of single-shard commands as we increase the number of shards. Throughput is linear in the number of shards, while the system still has the ability to consistently execute multi-shard commands. Figure 2 (right) shows the cumulative distribution function of the latency for all requests. Commands to one shard show a sharp CDF around the average latency. Increasing the number of involved shards also increases the coordination overhead of Dynamic Atomic Multicast: imbalances of client loads are compensated every Δ time interval (in this case 5 ms). Adding shards, and consequently message streams, increases the overhead of the deterministic merge mechanism

(see §III-A). This is visible in the CDF with a bend in the curve around 5 ms.

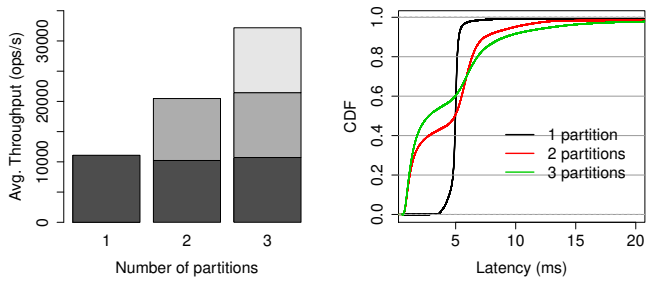

Fig. 2. Throughput (left) and latency (rigth) of DMap (1, 2 and 3 partitions).

2) Performance of iterators: In this experiment, an increasing number of clients create an iterator (snapshot) in DMap with 3 partitions. Clients then iterate over the whole distributed data set. The data set was previously provisioned with 1.2 million key-value pairs.

Results. As seen in Figure 3, the iterators show a better performance than the single command throughput. Initially, creating a snapshot is slow (200 ms), but once a snapshot (iterator) is created, every client can stream the data in parallel from every replica in all partitions. A single iterator achieves 50k entries per second while the number of parallel iterators scales almost linearly up to 50 clients.

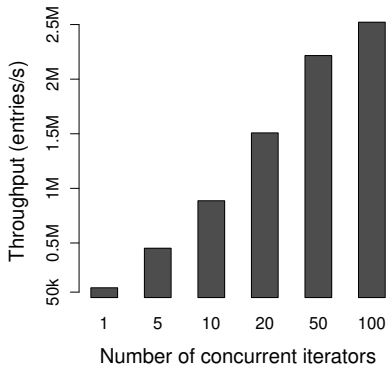

Fig. 3. Performance of retrieving entries of DMap iterator (1 to 100 clients).

3) Yahoo! Cloud System Benchmark (YCSB [14]): We evaluate the performance of DMap using YCSB and compare it with the performance of a single-server configuration that uses the Thrift interface only. Both systems are deployed with 3 partitions and we use 180 clients. The data set was previously provisioned with 1.2 million key-value pairs. All six core YCSB workloads are considered in our evaluation:

- *Workload A (Update heavy workload):* This workload has a mix of 50/50 reads and writes. An application example is a session store recording recent actions.
- *Workload B (Read mostly workload):* This workload has a 95/5 reads/write mix. An application example is photo tagging where adding a tag is an update, but most operations are reads.
- *Workload C (Read only):* This workload is 100% read. An application example is a user profile cache, where profiles are constructed elsewhere (e.g., Hadoop).
- *Workload D (Read latest workload):* In this workload, new records are inserted, and the most recently inserted records are the most popular. An application example is user status updates, where people want to read the latest.
- *Workload E (Short ranges):* In this workload, short ranges of records are queried, instead of individual records. An application example is a threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id).
- *Workload F (Read-modify-write):* In this workload, the client will read a record, modify it, and write back the changes. An application example is a user database, where user records are read and modified by the user or the user activities are recorded.

Results. The YCSB throughput of all workloads is shown in Figure 4. Workloads B, C and D correspond to the baseline performance of single-partition commands. Workloads A and F send update and read-modify-write commands. The way YCSB is implemented in DMap, such commands are composed of a read, followed by a write command. YCSB is a multi-map which allows to update a single entry in the value. DMap must first read the value as map, update the field and put again the whole value.

Workload E shows the performance of small scans. Retrieving a scan in DMap creates an iterator and loops over a small amount of values. Since the cost in DMap is creating iterators (snapshots) and not looping over iterators, the overall performance in case E is only 290 scans per second.

In all workloads, except E, the single-server Thrift implementation is faster. This is obvious, since all partitions run independent from each other (consistent scans are not possible) and there is no latency overhead of atomic multicast.

B. Recovery

As in the previous experiments, we use 180 clients to generate load on three shards, each one replicated on three replicas. The data set was previously provisioned with 1.2 million key-value pairs. After 20s we kill one replica in one replica group. At 40s we restart the killed replica, which immediate starts to recover.

Results. Figure 5 shows the system throughput over time while recovery is active. Instant (1) indicates the killing of one replica. At this point, performance drops to almost zero, since all commands to the failed replica time out. Additionally, all clients have to update their locally cached partition map. The partition map gets updated because the killed replica is removed. At (2), the replica starts recovering. Instant (3) indicates the end of recovery. The recovered replica updates the partition map with the information that it is operational.

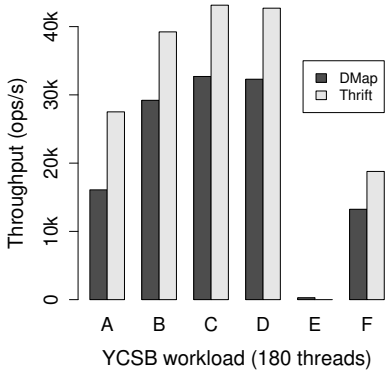

Fig. 4. Yahoo! Cloud Serving Benchmark for A:update heavy, B:read mostly, C:read only, D:read latest, E:short ranges, F:read-mod-write workloads.

Clients will install a new partition map, but compared to (1), no Thrift connections are invalidated. State transfer while recovering is very fast, since it uses the iterators described in §III-B.

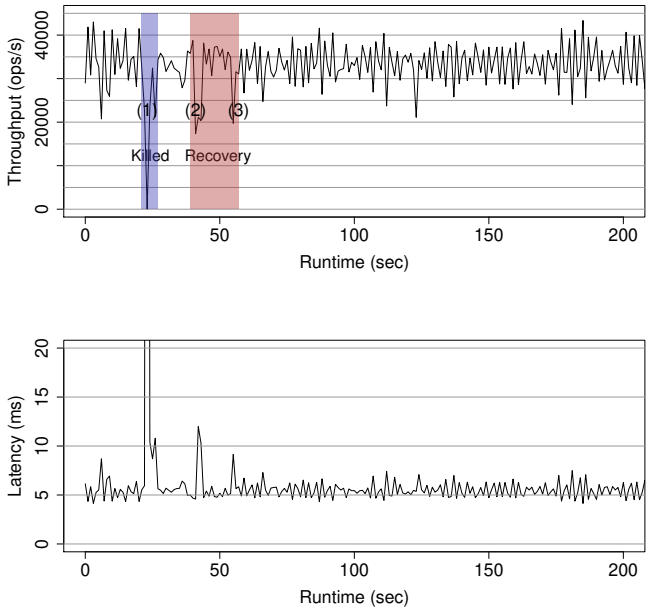

Fig. 5. Impact of recovery on throughput under full system load.

C. Repartitioning

In this experiment we start with two shards (P1 and P3) and after 20s we dynamically add a third shard (P2). We use 180 clients to generate load and the data set was previously provisioned with 1.2 million key-value pairs. The new set of 3 replicas first recover the state from the currently available shard (not shown in this experiment), reconfigure all involved atomic multicast message streams and later update the system partition map.

Results. Subscribing to and unsubscribing from message streams have no visible impact, as seen in Figure 6. The overall throughput drops during re-partitioning for a short period to 50% (half of the clients are re-assigned to the new shard). After re-partitioning, the overall throughput increases. Before the repartitioning, P1 was responsible for 2/3 of the hash space and therefore overloaded. After repartitioning, every shard is responsible for 1/3 of the keys, which explains why the average latency decreases.

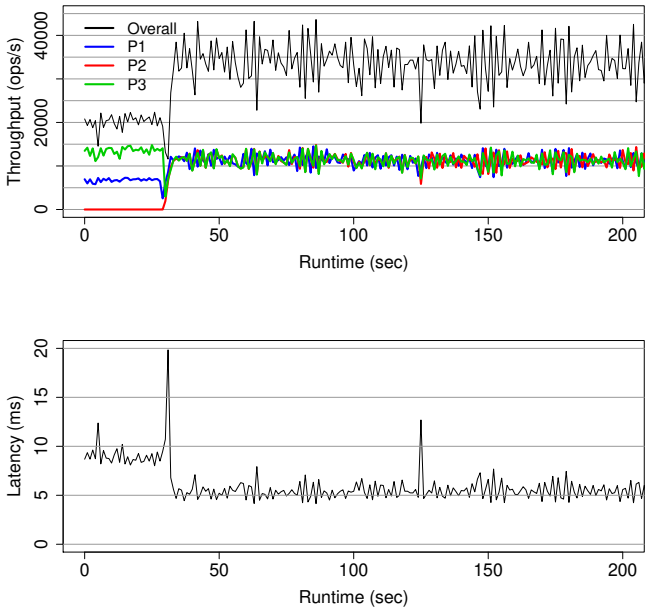

Fig. 6. Impact or repartitioning on performance.

V. Case study: H2 database on DMap

To demonstrate the usefulness of DMap, we replaced the local storage engine of the H2[5] database (MVStore) by DMap. H2 has a modular design, which encapsulates the SQL query processor from the storage layer. By replacing the local storage engine with DMap, multiple independent H2 query processor instances can run simultaneous on the same distributed data.

The core of H2 is MVStore. MVStore allows to create multiple independent sorted maps. The whole database relies on this storage abstraction. All database schema information, primary and secondary indexes, even the undo log are persisted in this layer. Therefore, replacing the MVMap used by MVStore with DMap distributes the whole database.

We needed less than 500 lines of source code to achieve our goal and run multiple H2 instances on top of DMap. Moreover, the modular design of H2 and the expressive interface of DMap allows us to use all special database operations, such

[5]http://www.h2database.com

as creating or altering tables, creating indexes or transactions, without further modifications. The new system supports distributed transactions, based on a distributed undo log, and online database schema altering (e.g., creating tables) which are immediately visible to all query processors. However, since some query optimizations rely on data local to the query processors, such operations would require additional work to distribute the required information.

By adding one additional Java class to H2, we could not only distribute the whole database, but due to the properties of DMap, we could implement a scalable (sharded) and fault-tolerant (replicated) system.

We evaluate the performance of H2 on DMap using the TPC-C benchmark. Our experiments show that H2 on DMap is largely outperformed by the unmodified H2. Figure 7 shows all operations the database executes on DMap during the execution of TPC-C. Single-partition commands run in parallel and can be scaled by adding new replica sets. The all-partition commands must be executed by every replica and do not scale. The create range commands are due to select queries of a range. H2 executes more than 100 DMap operations per second. But, the TPC-C throughput of the New-Order transaction is only about 8 transactions per second. This can be explained by the large number of operations needed to translate an SQL statement into DMap operations (Table I). On average, every SQL statement requires 10 DMap operations to update the undo log twice and setting the locked and final value to a table. This is a direct consequence of the fact that H2 was implemented with a local storage in mind. For example, H2 is not optimized to use a transaction cache. Such a cache could possibly reduce the number of DMap operations.

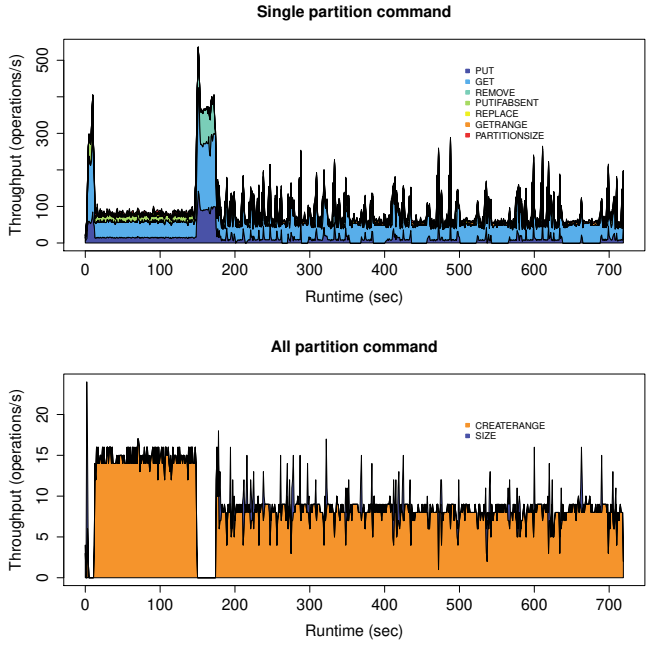

Fig. 7. H2 operations on DMap while performing the TPC-C benchmark.

TABLE I
FROM H2 SQL QUERIES TO DMAP OPERATIONS.

H2 :	"insert into test values (1,'String')"
DMap:	4*GET, PUT, PUTIFABSENT, 2*GET, PUT, REMOVE
H2 :	"select * from test where id=1"
DMap:	GET
H2 :	"update test set value name='XYZ' where id=1"
DMap:	4*GET, PUT, REPLACE, 3*GET, PUT, REPLACE, 2*GET, PUT, REMOVE, 2*GET, PUT, REMOVE
H2 :	"select * from test"
DMap:	SIZE, CREATERANGE
H2 :	"delete from test where id=1"
DMap:	4*GET, PUT, REPLACE, 2*GET, 2*REMOVE

VI. RELATED WORK

In this section, we review related work on distributed data structures, atomic multicast, and recovery.

Distributed data structures. There exists a variety of systems that implement distributed data structures. An overview is shown in Table II. These systems provide different interfaces, consistency guarantees or are built for specific optimized use-cases. To the best of our knowledge, no system implements a generic Java interface and provides scalable, consistent range queries.

One of the first distributed data structures similar to our ordered map was a B-tree algorithm based on a B-link tree proposed in [15]. However, the tree was designed for distributed memory architectures and not high latency networks. Even in the modern literature, not many distributed tree structures exist. SD-RTree [16] is a scalable distributed R-tree designed for networks. This data structure is based on a binary tree and optimized for spatial objects. The first distributed B-tree that tries to address similar requirements to the ones described in this paper is presented in [17]. The concurrency control is based on transactions and not locking, which was common in B-trees for distributed memory. Minuet, a scalable distributed multiversion B-tree [18] addresses the problem of long-running data analytics workloads in the context of short-living transactions. Minuet is based on Sinfonia [19] but provides an optimistic concurrency control mechanism to scale parallel inserts and updates. Further, it implements consistent snapshots and copy-on-write tree branches. Recent work on distributed data structures also proposes to use skip lists to implement efficient range queries for dictionaries [20]. Compared to the work presented in this paper, it uses a hardware level message passing interface (MPI).

Several other publications propose B-trees to build distributed systems. Boxwood [7] uses a distributed B-tree to implement a file system. The tree operations are coordinated by a distributed lock service. Hyder [21] implements an index structure based on a binary tree on a shared flash log. Similar to Hyder, Tango [22] generalizes distributed data structures on append only logs. Both use the log for transaction control and append a new version of the changed index to the log.

HyperDex [23] and Yesquel [6] are the most related to the work proposed here. HyperDex implements a partitioned

TABLE II
OVERVIEW OF EXISTING DISTRIBUTED DATA STRUCTURES.

System	Generic Java Iface	Type	Consistency	Partitioned
DMap	SortedMap	SortedMap	strong	yes
Yesquel	no	SortedMap	strong	yes
HBase	no	SortedMap	strong	yes
Cassandra	no	Map	weak	yes
Redisson	ConcurrentMap	Map	weak	yes
	SortedSet	SortedSet	weak	yes
Hazlecast	ConcurrentMap	Map	weak	yes
Dynamo	no	Map (w/ Scan)	weak	yes
Hyperdex	no	Map (w/ Scan)	strong	yes
SimpleDB	no	Map (w/ Scan)	consistent reads	yes
Ignite	JCache	Map	strong	replicated or partitioned
Atomix	no	Map	strong	no

key-value store which allows efficient search functions and secondary indexes based on a novel multi-dimensional hash function. Yesquel implements a distributed B-tree and proposes several optimizations to use the tree for a distributed SQL database. The architecture and concurrency control used in Yesquel are similar to Sinfonias's mini-transactions. Both systems implement a rich API. However, compared to DMap, their interfaces are not compatible to existing well-known Java interfaces.

Distributed databases. The idea of running multiple independent query processors on a distributed data store is not new. MoSql [24] implements a distributed storage engine for the MySQL database. Compared to H2/DMap, it uses deferred update replication to certify concurrent transaction before commit. Yesquel [6] also replaces the local B-tree implementation of SQlite with their distributed balanced tree. F1 [25] is a distributed SQL database which drives the Google ad-words business. The storage engine used by F1 is Spanner [3].

VII. CONCLUSIONS

Implementing a distributed data store that supports strong consistency, performance scalability, and fault tolerance is a challenging task. Some works have responded to this challenge with powerful communication primitives (i.e., atomic multicast), which encapsulate much of the complexity involved. Application developers, however, are often unfamiliar with such primitives.

This paper describes the design of DMap, a distributed data structure with important characteristics. DMap is scalable, supports dynamic re-partitioning, and fast recovery. It builds on atomic multicast to ensure that both single- and multi-shard commands are strongly consistent. More important, DMap is easy to use. It fully implements the generic Java SortedMap interface and is therefore designed to distribute and scale any existing Java application.

We detailed the system architecture of DMap and explained the underlying ordering and recovery mechanisms. Further, we evaluated the performance and demonstrated how system engineers can benefit from distributed data structures.

While DMap delivers scalable performance at low latency, our case-study of H2 on top of DMap shows some limitations of the approach. H2 assumes a non-distributed data store, and therefore misses optimizations that would be important in a distributed setting (e.g., transaction cache). This is something that cannot be solved efficiently in DMap.

VIII. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments and their suggestions to improve the paper. This work was supported in part by the Swiss National Science Foundation under grant number 146714.

REFERENCES

[1] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato, "Building global and scalable systems with Atomic Multicast," in *Middleware*, 2014.
[2] C. E. Bezerra, F. Pedone, and R. van Renesse, "Scalable State-Machine Replication," in *DSN*, 2014.
[3] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's Globally-Distributed Database," in *OSDI*, 2012.
[4] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[5] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.
[6] M. K. Aguilera, J. B. Leners, and M. Walfish, "Yesquel: scalable SQL storage for Web applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 245–262.
[7] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the Foundation for Storage Infrastructure." in *OSDI*, vol. 4, 2004, pp. 8–8.
[8] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.
[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty processor," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.
[10] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.
[11] S. Benz and F. Pedone, "Elastic Paxos: A Dynamic Atomic Multicast Protocol," in *ICDCS*, 2017.
[12] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.
[13] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, Feb. 1985.
[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.

[15] T. Johnson and A. Colbrook, "A distributed data-balanced dictionary based on the b-link tree," in *Parallel Processing Symposium, 1992. Proceedings., Sixth International*. IEEE, 1992, pp. 319–324.

[16] C. Du Mouza, W. Litwin, and P. Rigaux, "Sd-rtree: A scalable distributed rtree," in *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 2007, pp. 296–305.

[17] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed b-tree," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 598–609, 2008.

[18] B. Sowell, W. Golab, and M. A. Shah, "Minuet: a scalable distributed multiversion B-tree," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 884–895, 2012.

[19] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *ACM SIGOPS OSR*, vol. 41. ACM, 2007, pp. 159–174.

[20] S. Alam, H. Kamal, and A. Wagner, "A scalable distributed skip list for range queries," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014, pp. 315–318.

[21] P. A. Bernstein, C. W. Reid, and S. Das, "Hyder-A Transactional Record Manager for Shared Flash." in *CIDR*, vol. 11, 2011, pp. 9–20.

[22] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed data structures over a shared log," in *SOSP*, 2013.

[23] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A distributed, searchable key-value store," *Acm sigcomm computer communication review*, vol. 42, no. 4, pp. 25–36, 2012.

[24] A. Tomic, D. Sciascia, and F. Pedone, "MoSQL: An Elastic Storage Engine for MySQL," in *SAC*, 2013.

[25] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner *et al.*, "F1: A distributed SQL database that scales," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1068–1079, 2013.