Practical Experience Report: Cassandra+: Trading-Off Consistency, Latency, and Fault-tolerance in Cassandra

Guo-Shu Gau
salous@gmail.com
Tenafe Technology

Kishori Konwar
kishori@csail.mit.edu
Broad Institute, MIT

Juan Mantica†
jp4523456@gmail.com
Audible

Haochen Pan
haochen.pan@bc.edu
Boston College

Darius Russell Kish
russeldk@bc.edu
Boston College

Lewis Tseng
lewis.tseng@bc.edu
Boston College

Zezhi Wang†
herrywangzz@hotmail.com
Brown University

Yingjian Wu†*
yiw079@ucsd.edu
UCSD

ABSTRACT

The exponential growth of data has pushed the industry towards new types of data management systems such as NoSQL database. Our goal is to augment NoSQL, with the focus on its application as a distributed key-value store (KV-store). Existing production-ready systems often provide only probabilistic guarantees on consistency and fault-tolerance, and may violate their correctness properties if a cluster has severe clock drift. Our system Cassandra+ addresses these issues by providing more choices for consistency and fault-tolerance. We build Cassandra+ by implementing theoretical distributed shared memory (DSM) in Cassandra, one of the most popular NoSQLs in the industry. In this paper, we share our experience in adapting and implementing DSM algorithms into a real-world system. We hope our experience and results allow a better understanding of the DSM and the tradeoffs involved.

CCS CONCEPTS

• **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Computing methodologies** → **Distributed algorithms**.

KEYWORDS

Cassandra, Consistency, Fault-tolerance, Implementation, Evaluation, Distributed KV-store

*Authors listed alphabetically. Authors with † worked on this project when they were affiliated with Boston College. Authors from Boston College are supported in part by National Science Foundation award CNS1816487. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.

ICDCN '21, January 5–8, 2021, Nara, Japan

ACM ISBN 978-1-4503-8933-4/21/01…$15.00
https://doi.org/10.1145/3427796.3427816

ACM Reference Format:
Guo-Shu Gau, Kishori Konwar, Juan Mantica†, Haochen Pan, Darius Russell Kish, Lewis Tseng, Zezhi Wang†, and Yingjian Wu†. 2021. Practical Experience Report: Cassandra+: Trading-Off Consistency, Latency, and Fault-tolerance in Cassandra . In *International Conference on Distributed Computing and Networking 2021 (ICDCN '21), January 5–8, 2021, Nara, Japan.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3427796.3427816

1 INTRODUCTION

With the advent of the Cloud and mobile devices, we have witnessed an exponential growth in the popularity of online activities that are generating unprecedented amounts of data on the Internet. NoSQL databases have become one of the most popular storage solutions (such as Amazon Dynamo [14] and Cassandra [2], and RIAK [1]) because of its salient features like availability, scalability, capability of handling unstructured data, etc. In this work, we focus on the usage of NoSQL databases as a distributed key-value store (KV-store), as this simple abstraction can be viewed as a well-studied topic in the distributed computing community better known as *Distributed Shared Memory (DSM)* or *read/write data objects (or simply registers)* in message-passing systems.

KV-stores maintain data in the form of key-value pairs (or KV-pairs) which can be accessed concurrently by write (*PUT*) or read (*GET*) operations. To ensure high availability and horizontal scalability, KV-stores replicate data to a large number of commodity machines, and are often classified by the consistency and fault-tolerance guarantees they provide. For example, Cassandra provides consistency choices ranging from strong to eventual consistency. Depending on the selected consistency level (i.e., the minimum number of Cassandra nodes that must acknowledge an operation), Cassandra tolerates different number of crashed nodes. While these properties are sufficient for a wide range of application scenarios, we observe that it is possible to improve Cassandra by providing more choices for consistency and fault-tolerance guarantees.

Motivation and Goals: Most KV-stores use probabilistic quorum [8] to replicate data. Such a design have a few limitations: (i) Its strong consistency only ensures returning the most recent value, which does *not* guarantee a *total ordering* of operations, i.e., linearizability or atomicity [21]; (ii) Its strong consistency is ensured by quorum intersection and synchronized clocks, which does *not* always hold

if a cluster has severe clock drift or intermittent machine failures; (iii) It only tolerates machine crash failures; and (iv) Its weaker consistency (e.g., probabilistically bounded staleness [8]) is difficult to reason with, and does *not* capture natural causality relations.

These issues indeed have been studied thoroughly in the theory literature, and evaluated rigorously under theoretical models. However, we are *not* aware of any practical systems that addressed these issues *without* using expensive coordination protocols such as consensus or leader/master. This observation motivates us to make the following main contributions:

- We have developed our system Cassandra+ based on Cassandra. More precisely, we replace Cassandra's replication protocol by five DSM algorithms – ABD [7], MW (Multi-Writer) [24], SBQ (Small Byzantine Quorum) [22], CM (Causal Memory) [4]), and BSR (Byzantine Safe Register) [15]. These algorithms are lock-free in the sense that they do not rely on any kind of consensus, commit, or lock protocols.
- Cassandra+ABD provides atomicity, which is one of the strongest and most widely researched consistency in the theory community. It provides a total order with real-time constraints, which intuitively ensures an illusion that each operation is carried out sequentially, as if it is executed on a single machine. Hence, atomicity is simpler to reason with, compared to other consistency models that do not have such a total ordering constraint.
- Cassandra+CM provides causal consistency, which ensures that clients observe causality or happens-before relation [20].
- Cassandra+MW explores various forms of strong consistency that provide pair-wise total ordering.
- Cassandra+SBQ provides safeness consistency under semi-Byzantine fault model, where a faulty server can behave arbitrarily, but the timestamp cannot be tampered.
- Cassandra+BSR provides safeness consistency under Byzantine servers and crash-prone clients. BSR assumes that timestamp is corruptible, and thus tolerates a typical Byzantine model.
- The DSM algorithms in Cassandra+ use some types of logical timestamps; hence, they are correct even if clocks drift severely.
- Cassandra users can use our implementation *without* knowing the details of the algorithm. That is, users can use the same Cassandra API regardless of which replication algorithm is used.

We implement these algorithms in Java and seamlessly port our implementation with other components in Cassandra [2]. We chose to develop our framework inside Cassandra mainly because of its popularity and active community.

Practitioner Experience: Our main technical contributions are adapting and implementing prior theoretical DSM algorithms into the framework of Cassandra so that they are compatible with other components in Cassandra. In this paper and accompanied technical report, we detail our experience in working with Cassandra, e.g., how to decompose its functionalities, where to implement the storage algorithms, and which data structure to be used, etc. Our hope is that our experience will lower the barrier of implementing other theoretical DSM algorithms in similar NoSQL systems in the future. To our knowledge, Cassandra adopts a general architecture that shares similarity with many other NoSQL systems. Additionally, the five DSM algorithms we chose cover a wide spectrum of properties, e.g., strong/weak consistency, crash-/Byzantine-fault tolerance, quorum/non-quorum-based. Hence, we hope that the lessons we share in this paper would achieve two interleaving goals: (i) for theoreticians to see how their algorithms fit with production-ready systems; and (ii) for practitioners to integrate theoretical DSM algorithms with NoSQL systems. Our implementation of Cassandra+ can be found at https://github.com/Dariusrussellkish/cassandra

In the literature of fault-tolerance and distributed computing, numerous DSM algorithms have been proposed, e.g., [4, 7, 22, 24], along with a wide spectrum of consistency and fault-tolerance guarantees. Unfortunately, to our knowledge, very few of the proposed algorithms have been integrated into real-world systems, let alone thoroughly evaluated against production systems. Moreover, common metrics of interest are message and round complexity in the theory community. However, several prior works [5, 9, 13] have demonstrated that such theoretical metrics do not indicate the practical performance. As a consequence, it is very difficult for practitioners to pick the most appropriate DSM algorithms or even tune the performance of the implemented algorithms. Our extensive evaluation results shed light on this aspect.

We evaluate our implementation extensively using Google Cloud Platform (GCP).All the servers and clients are in the same datacenter. We collect both latency and throughput using the YCSB (Yahoo! Cloud Service Benchmark) workload generator [12]. In Table 1 below, we present a quick glance on a particular set of our execution. For Cassandra, we use the configuration providing weakest (eventual) consistency as the basis, namely Cass-One.

Related Work: There is a long history of research on consistency models, e.g., [3, 16, 19] and fault-tolerance such as atomicity, sequential consistency, and regularity, e.g., [7, 21, 25]. In the theory literature, numerous DSM algorithms have been proposed [4, 7, 22, 24, 25]. However, to the best of our knowledge, there is very few study on comparing the practical performance of these algorithms in a real-world environment. Closest works that we notice are the ones evaluating Byzantine Quorum Systems (BQS), e.g., [5, 9, 13]. While they provided careful analysis and implementation, the results are quite outdated (the newest one is more than 13 years ago). Plus, we implement the DSM algorithms in a real-world system that have a potential to be production-ready, whereas prior works proposed and built their own evaluation framework that lacks integration with other components like failure detection and database engine. Finally, with the advancement of open-source technology and cloud platform, we are able to deploy and test our systems in a real-world platform, a privilege that prior works lacked.

2 CASSANDRA VS. CASSANDRA+

Cassandra and Replication: Cassandra implements a quorum-based storage protocol called Probabilistic Quorum System (PQS) [8]. PQS was inspired by a long history of study on quorum-based systems [17, 18, 25]. It was first implemented in Dynamo [14], and later adopted in open-source systems like Cassandra [2] and RIAK [1]. After these systems become popular in industry, the PQS framework

	Fault Tolerance	Consistency	Read/Write RTT	Avg Read Lat. Ratio	Avg Write Lat. Ratio	Throughput Ratio
Cass-One	Crash	Eventual	0/0	1	1	1
ABD-OPT	Crash	Atomicity	1.5/2	1.17-1.43	2.47-3.48	0.67-0.79
MW	Crash	A family of regularity	1/2	1.27-1.49	2.59-3.59	0.64-0.73
CM	Crash	Causal Consistency	0/0	1.08-1.18	1.16-1.28	0.86-0.92
SBQ	Semi-Byz.	Safeness	1/2	1.35-1.64	1.88-2.65	0.61-0.72
BSR	Byzantine	Safeness	1/2	1.3-1.45	4.55-8.11	0.43-0.63

Figure 1: Cassandra+ Performance

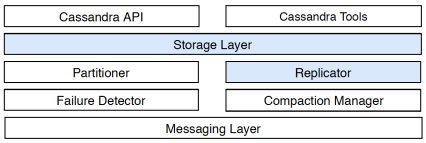

Figure 2: Architecture of Cassandra and Cassandra+

was formalized by Bailis et al. [8]. Several research groups have proposed solutions to augment Cassandra, e.g., [23].

Cassandra is designed to be distributed, scalable, and highly available. To achieve this, a Cassandra cluster contains a set of servers (or replicas) that jointly act as a single instance to the users. To ensure fault-tolerance and availability, each KV-pair is assigned and replicated to multiple servers. The data is partitioned into ranges based on its key, and each range of KV-pairs are dynamically assigned to servers using consistent hashing [2, 14].

PQS requires each client proxy (or coordinate) to obtain responses from read and write quorums. The size of read and write quorums is denoted by r and w, respectively. Strong consistency is satisfied if $r + w > n$, where n is the number of servers. The inequality implies quorum intersection, which guarantees at least one common server that interacted with the clients of each pair of read and write operations. Here, Cassandra's tunable parameter "consistency level" specifies r and w. When $r + w \leq n$, Cassandra ensures eventual consistency, which can be analyzed using the probabilistically bounded staleness framework [8]. Strong consistency in Cassandra (or PQS) only ensures that a read will return the most recent written value; however, it does *not* provide a notion of *total ordering*. Moreover, if time is not perfectly synchronized inside a Cassandra cluster, then clients might read a stale value if the clocks are not synchronized.

Cassandra Architecture: Cassandra is a fairly complicated system which has many components whose high-level architecture is presented in Figure 2. We mainly modify code in storage and replicator layers. Storage layer specifies how each node handles storage-related requests, e.g., fetching local data and writing a data into memory or disk. Replicator layer specifies the replication strategy. In our implementation, we make sure our code is compatible with other components.

DSM in Cassandra+: Most DSM algorithms are designed to tolerate network asynchrony, clock drift, and node failures. As the first step, we target those algorithms that are *always available*, i.e., the algorithms satisfy liveness if up to a certain fraction of nodes become faulty. In particular, all algorithms implemented in Cassandra+ do *not* require any form of lock, consensus or leader/master. All the algorithms we evaluate adopt some form of quorum-based design. Such a design aligns well with the PQS used by Cassandra, and this is the main reason that we can integrate them with Cassandra's structure seamlessly.

There are three main reasons behind our choice of the algorithms – ABD [7], MW (Multi-Writer) [24], CM (Causal Memory) [4], SBQ (Small Byzantine Quorum) [22], and BSR (Byzantine Safe Register) [15]: (i) These algorithms together cover a wide spectrum of fault-tolerance and consistency guarantees; and (ii) ABD and CM are well-known in the theory community; however, we have never seen a thorough performance comparison in a real-world setting; and (iii) The internal data structure of these algorithms are quite different. For example, ABD and MW do not need extra data structure; CM uses a priority queue; SBQ uses a hashmap; and BSR uses both a hashmap and priority queue. Key properties and summary of each algorithm are described below:

- **ABD** [7] is a well-known crash-tolerant replication algorithm, named after the authors Attiya, Bar-Noy and Dolev. ABD ensures *atomicity* of read/write operations. The original version of ABD in [7] is for a single-writer case. In this paper, we follow the presentation of the multi-writer version in [6]. We also implement different versions of ABD as explained later in Section 3.
- **MW (Multi-Writer)** [24] consists of a family of crash-tolerant algorithms ensuring various definitions of *multi-writer regularity*, which is weaker than atomicity. MW algorithms can be viewed as a decomposed version of ABD. The authors of [24] decomposed ABD into multiple building blocks, and showed that different combination of blocks achieve a version of multi-writer regularity.
- **CM (Causal Memory)** [4] is a crash-tolerant DSM algorithm ensuring causal consistency [20]. It intuitively means that the effect (e.g., response to a read) must be ordered after the cause (e.g., invocation of the read). In other words, causality or happens-before relation [20] is always guaranteed.
- **SBQ (Small Byzantine Quorum)** [22] ensures safeness [21] under semi-Byzantine servers and crash-prone clients. It only tolerates semi-Byzantine servers, because the (logical) timestamp is assumed to be *incorruptible*. The underlying communication channel is assumed to be reliable.
- **BSR (Byzantine Safe Register)** [15] ensures safeness [21] under Byzantine servers and crash-prone clients. BSR satisfies safeness even under the assumption that timestamp from faulty servers can be corrupted.

The first two algorithms assumes majority correctness, i.e., at least $\lfloor \frac{n}{2} \rfloor + 1$ servers do not crash, and CM is correct as long as there is

one live server. SBQ assumes $\lfloor \frac{2n}{3} \rfloor + 1$ are correct, whereas BSR assumes $\lfloor \frac{3n}{4} \rfloor + 1$ are correct.

Refer to Table 1 for a quick reference to the properties of the algorithms under testing and a summary of *one* particular set of our evaluations. RTT measures the round-trip time required between the coordinator and other servers (replicas), which is a common metric in the theory literature. Note that in the DSM literature, users/clients co-locate with servers; hence, RTT does *not* include the communication time between user and coordinator in our framework. In Table 1, Cass-One denotes the case when PQS is configured with $r = w = 1$, which provides only eventual consistency. This is why CM and Cass-One require 0 RTT. They can immediately return after the coordinator retrieves its local data, i.e., coordinator does not need to wait for acknowledgment from other servers before responding to the client.

For latency and throughput ratio, we use Cass-One as the reference. For ABD entries, we report numbers for ABD-OPT, an optimized version of ABD [7]. For MW entries, we report numbers for MW-All-Off, the most lightweight one in MW algorithm family [24]. The numbers in the table corresponding to the plots in Figure **??**, which has write ratio 0.1 and data size $32B$, and evaluates performance against varying number of writer threads per YCSB client. Please see evaluation details in Section 3. Table 1 together with the results shown in Section 3 should provide a quick reference on the practical tradeoff among performance, fault-tolerance and consistency guarantees.

3 CASSANDRA+ AND EVALUATION

In this section, we share our experience in hopes of lowering the barrier of implementing other theoretical algorithms in Cassandra or other similar systems in the future. In order to obtain a meaningful and fair evaluation, we have three main goals in our implementation: (i) We make a minimum modification to the original Cassandra codebase. For example, we try not to introduce new complicate data structure. We constraint ourselves of using either Cassandra's internal data structure or simple data structures natively supported by Java, e.g., singleton monotonic map. (ii) For each DSM algorithm implementation, we add our code into the same files and follow a similar style into Cassandra codebase. (iii) Cassandra users can use our implementation *without* knowing the details of the algorithm. That is, users can use the same Cassandra API regardless of which replication protocol is used. As a benefit, we can directly use YCSB [12] to evaluate each of our implementation.

This turns out to be a difficult task, at least for the first implementation, because we are constrained to using only Cassandra's internal tools and data structures for communication and coordination. We spent enormous amount of time to decompose and understand Cassandra's internal logic because the codebase is not well documented and often provide poor comments. The version of Cassandra (version 3.11.2) that we used to develop contains around 57,000 LoC in Java.[1] We have implemented 9 algorithms in total (including variations of ABD and MW algorithms). Our implementation consists of around 2,800 LoC in total.

[1]The most recent version 3.11.6 was released in February 2020: https://cassandra.apache.org/download/

Adaption and Implementation. Cassandra adopts the column-family data model, which provides a richer functionalities than a simple read/write data object abstraction for DSM. At a high level, Cassandra's data structure can be viewed as a table, where each row corresponds to data sharing the same key. Hence, in our implementation, each row is used to store a single KV-pair. In addition to the key and value, we also use extra columns to store metadata used by each DSM algorithm such as logical timestamp or vector timestamp. We will discuss in more details in our technical report. In our implementation, we do *not* modify how Cassandra handles user requests, and let its original mechanism handle load balancing and cache/memory management. Therefore, our implementation supports Cassandra's original read/write API. Since the DSM algorithms under testing do not support transactions, those transaction-related APIs are not supported.

All the DSM algorithms that we implemented have separate algorithms for reader/writer client and server separately. Server code is mainly implemented in *MutationVerbHandler.java* and *ReadCommandVerbHandler.java*, whereas client code is mainly implemented in *StorageProxy.java*. There are some minor logics implemented in files dispersed at other places. These minor changes can be tracked on our Github repository. To ensure correctness, we do *not* modify the algorithms themselves. Our efforts lie in translating pseudocode to efficient and correct Java code. While most DSM algorithms are not complicated in terms of algorithm design, the key challenge is using Cassandra's internal tools *without* introducing new communication or non-native data structures. Our approach not only provides a fair comparison with unmodified Cassandra, but also makes our implementation compatible with other components inside Cassandra.

Evaluation: We use the Google Cloud Platform (GCP) as our testing platform. Each virtual machine (VM) is equipped with 4 virtual CPUs, 16 GB memory, and hosting Ubuntu 14.04 LTS. We present our evaluations in the local cluster with 3-server setup (LAN). The average RTT between any two VMs is around 0.2 ms. We use the Yahoo! Cloud Serving Benchmark (YCSB) [12] for evaluation. We have three additional VMs inside the same datacenter running an YCSB instance and each YCSB instance performs 0.3 million read/update operations on 60,000 KV pairs.

For each data point reported below, we make 5 experiment runs, and collect both the average and 95th percentile data for latency, and average for throughput. We study the impact of these performance measures in terms of (i) the number of readers and writers, (ii) the (data) value size, and (iii) ratio of read to write frequencies. Our experiments were confined to the following implementations: (a) unmodified Cassandra with $r = w = 1$ (Cass-One), (b) plain always two-round ABD; (c) optimized ABD (ABD-OPT); (d) Causal memory (CM); (e) unmodified Cassandra with quorum, i.e., $r = w = 2$ (Cass-Quorum); (f) Small Byzantine Quorum (SBQ);(g) MW without any building blocks (MW-All-Off); (h) MW without write-back (MW-No-WB); (i) MW without local cache (MW-No-LC); (j) ABD-MT (ABD-Machine Time); and (k) BSR (Byzantine Safe Register).

Across all the experiments, Cass-One should always have superior performance because a read or write from one single server is the simplest of distributed operation one can perform. Hence, we use Cass-One as the baseline. In the first set of the evaluations,

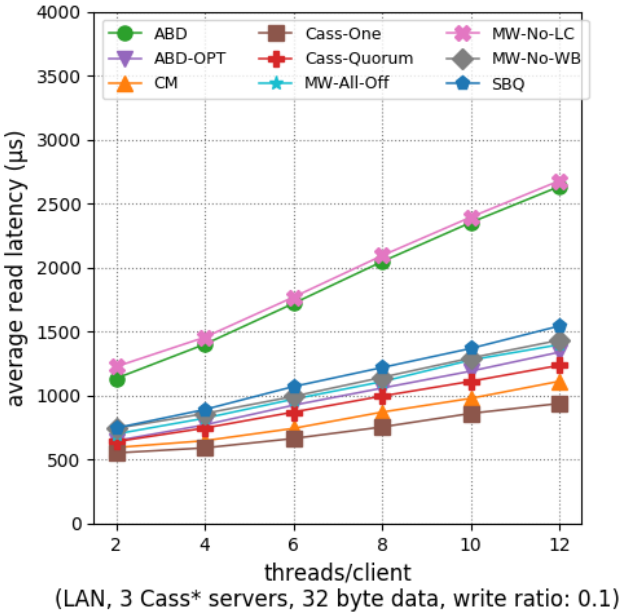

Figure 3: Average read latency

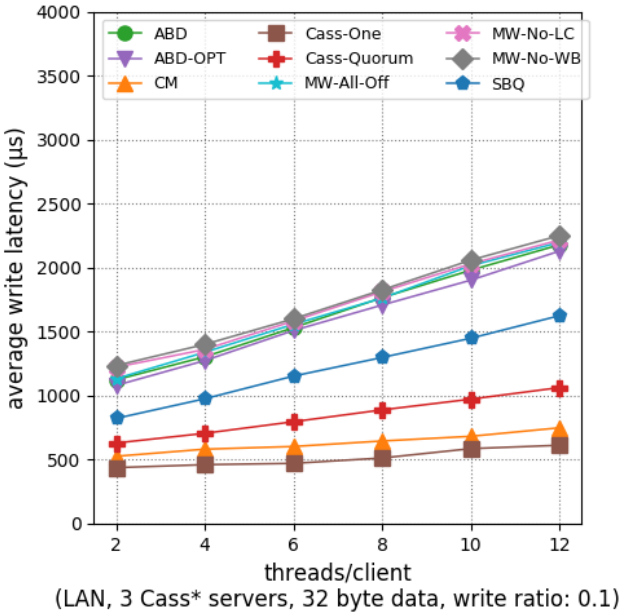

Figure 4: Average write latency

we plot average read latency (in μs), and average write latency (in μs) against various number of threads per YCSB instance in Figures 3 and 4, respectively The value size is 32 Bytes and the write ratio is 0.1 – we pick these two parameters according to a recent study [11] in Facebook workload. The numbers in Table 1 are based on this set of experiments. Our technical report presents more evaluation results, including performance against value size, WAN case, 5-server cluster, etc. We also report throughput, 95th percentile, and read/write latency together with limitations and potential reasons.

4 SUMMARY

We strengthen the Cassandra framework with five family of DSM algorith that provide a wide range of consistency guarantees for practical applications ranging from atomicity to causal consistency. We call this strengthened system Cassandra+. Most importantly, we share our experience and lessons in this paper. We hope to stimulate further effort in understanding the tradeoff among performance, fault-tolerance, and consistency, and implementing and adapting theoretical algorithms in real-world systems.

REFERENCES

[1] Basho Riak. http://basho.com/riak/.
[2] Cassandra. http://cassandra.apache.org/.
[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.
[4] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 1995.
[5] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *Digest of Papers: FTCS-26*, pages 26–35, 1996.
[6] J. Aspnes. Notes on theory of distributed systems. *CoRR*, abs/2001.04235, 2020.
[7] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.
[8] P. Bailis et al. Probabilistically bounded staleness for practical partial quorums. *VLDB Endowment*, 5(8):776–787, 2012.
[9] O. Bakr and I. Keidar. Evaluating the running time of a communication round over the internet. In *PODC 2002*.
[10] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.
[11] N. Bronson et al. TAO: facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*.
[12] B. F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *Proc. ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010.
[13] W. S. Dantas, A. N. Bessani, J. da Silva Fraga, and M. Correia. Evaluating byzantine quorum systems. In *26th IEEE SRDS 2007*.
[14] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Proc. ACM SIGOPS SOSP*, pages 205–220, 2007.
[15] K. Konwar et al. Semi-fast Byzantine-tolerant shared register without reliable broadcast. In *ICDCS*, 2020.
[16] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, May 1990.
[17] D. K. Gifford. Weighted voting for replicated data. In *SOSP 1979*.
[18] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, 2003.
[19] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *SIGARCH Comput. Archit. News*, Apr. 1992.
[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
[21] L. Lamport. On interprocess communication. *Distributed Computing*, 1986.
[22] J. Martin, L. Alvisi, and M. Dahlin. Small byzantine quorum systems. In *2002 International Conference on Dependable Systems and Networks*, 2002.
[23] M. R. Rahman et al. Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores. *ACM Trans. Auton. Adapt. Syst.*, 2017.
[24] C. Shao, J. L. Welch, E. Pierce, and H. Lee. Multiwriter consistency conditions for shared memory registers. *SIAM Journal on Computing*, 40(1):28–62, 2011.
[25] M. Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.