Distributed Data Persistency

Apostolos Kokolis, Antonis Psistakis, Benjamin Reidys, Jian Huang, and Josep Torrellas, *University of Illinois at Urbana-Champaign, Champaign, IL, 61801, USA*

Distributed applications such as key-value stores and databases provide fault tolerance by replicating records in the memories of different nodes, and using data consistency protocols to ensure consistency across replicas. In this environment, nonvolatile memory (NVM) offers the ability to attain high-performance data durability. However, it is unclear how to tie NVM memory persistency models to the existing data consistency frameworks. In this article, we propose the concept of distributed data persistency (DDP) model, which is the binding of the memory persistency model with the data consistency model in a distributed system. We reason about the interaction between consistency and persistency by using the concepts of visibility point and durability point. We design low-latency distributed protocols for several DDP models, and investigate the tradeoffs between performance, durability, and intuition provided to the programmer.

Over the past decades, distributed storage systems, such as key-value stores and transactional databases, have become a core component of the cloud infrastructure. To meet ever-increasing performance requirements, these distributed applications typically avoid frequent accesses to slow storage devices, such as solid-state drives (SSDs). Instead, they store the records in main memory and provide fault tolerance by making replicas (i.e., copies) of the records in other nodes' memories.

These replicas are managed by the runtime system using a *data-consistency model*. There are many different *data-consistency* models in use,[1,2] which differ in their strength. Strong consistency models strive to ensure that reading different replicas in different nodes returns the same, up-to-date version of the record. In contrast, weak consistency models permit reads to different replicas to return different, sometimes stale versions. Commercial applications support a variety of models—e.g., Apache's ZooKeeper supports the strong *Linearizable* consistency, while Google's Bigtable provides the weak *Eventual* consistency.

0272-1732 © 2022 IEEE
Digital Object Identifier 10.1109/MM.2022.3162183
Date of publication 24 March 2022; date of current version 30 June 2022.

Not using slow durable storage devices helps deliver high performance during fault-free execution. However, when a fault occurs, distributed applications that use volatile memory to store replicas are subject to data loss or slow data recovery. For example, a Meta key-value store cluster needs hours to recover using remote data replicas, and days to recover using a backend storage.

THE ARRIVAL OF NONVOLATILE MEMORY (NVM) OFFERS A PROMISING APPROACH TO HELP DISTRIBUTED APPLICATIONS ATTAIN BOTH HIGH PERFORMANCE AND DURABLE STORAGE FOR IMPROVED DATA RECOVERY.

The arrival of nonvolatile memory (NVM) offers a promising approach to help distributed applications attain both high performance and durable storage for improved data recovery. NVM can provide data durability in about 100–400 ns. This is faster than a network round trip in data centers with InfiniBand.

To facilitate the use of NVM, researchers have developed a framework of *data-persistency models* for a single machine with hardware-managed cache hierarchies.[3] These models vary in how eagerly they persist writes to

SIDEBAR: DATA-CONSISTENCY AND MEMORY-PERSISTENCY MODELS

Data-Consistency Models

In a distributed computer with replicas of records in different nodes, multiple processes running on different nodes may concurrently read and update the replicas of a given record. The consistency model of a system defines the requirements and guarantees of what data values can processes read. Some popular *data-consistency* models are as follows.

Linearizable consistency or linearizability: As the strongest model, it requires that all writes to all records be seen by all processes in the same order and, in addition, that all reads and writes be ordered by their timestamps.

Causal consistency: Accesses are partially ordered according to the Happens Before (HB) relationship. Specifically, two accesses in the same process are ordered based on the program order. A read from a process that obtains a value written by a write from another process is ordered after the write. Furthermore, this relation is built transitively. In this model, a process can observe a write *w* only after it observes every previous write in *w*'s HB history. Writes do not need to be applied instantly and, therefore, reads can return stale values.

Eventual consistency: Writes are propagated lazily. This model only guarantees that all the replicas will eventually see all the writes. This model provides very weak consistency guarantees, and reads can return unexpected values.

Transactional consistency: Accesses are organized in transactions (Xactions). Although multiple variations exist, this article uses a simple model. The writes in a Xaction only need to be propagated to all the replicas by the end of the Xaction. If the Xaction fails, none of the updates are performed. Moreover, the operations within a Xaction can only see the effects of other Xactions that have completed prior to it.

Read-enforced consistency: We introduce this new model, which is slightly weaker than linearizable consistency. A write only needs to update all the replicas at the point when a subsequent read tries to read any of the replicas. Compared to linearizability, this model allows faster completion of write requests at the potential expense of delaying reads.

Memory-Persistency Models

The availability of NVM has led to the creation of multiple memory-persistency models for single-server platforms. We adapt these models to work on a distributed system, building on traditional durability models that distributed systems have used to persist data to SSDs. The memory-persistency models we use are as follows.

Synchronous persistency: We introduce this new model as the adaptation of the strict memory-persistency model from single-server systems[3] to distributed systems. When a replica is updated in volatile memory, it is immediately persisted to NVM. This model is strict, but the time of the persist depends on when the replica is updated, which in turn depends on the *data-consistency* model of the system.

Read-enforced persistency: This model is more relaxed than synchronous. Replicas do not need to be persisted when they are updated. Instead, all the updated replicas need to be persisted before any of them is read. This model guarantees that any value that has been read is also recoverable. However, updates that have not yet been read may be lost in a crash.

Eventual persistency: Persist operations are performed lazily. They happen whenever it is possible, without any concern about the order of persists. In case of a volatile storage failure, an arbitrary number of updates may be lost.

Scope persistency: We introduce this model as an example of the model proposals that persist a set of writes as a group. Every write is augmented with a *Scope ID*, and the application can invoke a *Persist* on a given Scope ID. Writes can be persisted in the background, but the model guarantees that all the writes in a scope are persisted by the time the persist call for that scope terminates. On a failure, the state of all the completed scopes is recovered, and that of those partially executed is discarded.

Strict persistency: Strictest model that dictates that a write should be persisted in the NVM of all the replica nodes by the time the write completes—possibly even before the replicas in the volatile memories of the replica nodes receive the update. On a failure of volatile storage, no update is lost. This model is relatively less interesting because of its high strictness.

NVM. For example, *Strict* persistency requires that a record be persisted as soon as it is updated, while *Epoch* persistency only requires that updated records be persisted at certain program locations.

As we use NVM in distributed applications and systems, we have to carefully manage both the consistency and persistency of the data. Although distributed *data-consistency* has been well studied, it has almost always been used in systems which, at best, use slow storage devices for durability. Hence, it is unclear how to best incorporate the NVM memory persistency models into these *data-consistency* frameworks. In fact, it is unclear how these two classes of models interact with each other, and how their combination impacts data durability, performance, and programmer intuition in applications.

INTEGRATING PERSISTENCY IN DISTRIBUTED SYSTEMS

The "Data-Consistency and Memory-Persistency Models" sidebar describes some *data-consistency* models and memory-persistency models. They include some of the most popular ones, which will be used in this article.

In this work, we introduce the concept of *distributed data-persistency* (DDP) model of a distributed system, which is the binding of the memory-persistency model with the *data-consistency* model. To understand DDP models, consider a distributed computer where each node has a volatile memory hierarchy and some NVM. When an application runs on this platform, the runtime makes copies of records in the volatile memory hierarchy of multiple nodes. Such replication is performed for fault tolerance and performance, and may be later followed by the persistence to NVM.

In such a system, we decouple *data-consistency* models from memory-persistency models by using the concepts of *visibility* and *durability*. The consistency model is concerned with visibility, or *when to propagate* the update of a record to its replicas in the volatile hierarchies of nodes. The persistency model is concerned with durability, or *when to persist* the update to the NVMs of nodes. To be specific:

> The consistency model defines the *visibility point* (VP). The VP of an update with respect to a node is when the update becomes available for consumption at that node. The persistency model defines the *durability point* (DP). The DP of an update is when the update is made durable (in the necessary number of nodes, as required by the recovery system) and, hence, cannot be wiped out by a failure.

TABLE 1. Visiblity and durability points of an update for different consistency and persistency models, respectively.

Consistency	VP of an update
Linearizable	Wrt all R nodes: when the update takes place
Read-enforced	Wrt all R nodes: before the update is read
Transactional	Wrt all R nodes: at the transaction end
Causal	Wrt a R node: after the VPs wrt the same node of all the updates in the happens-before (HB) history
Eventual	Wrt a R node: sometime in the future
Persistency	**DP of an update**
Strict	When the update takes place
Synchronous	At the VP of the update
Read-enforced	Before the update is read
Scope	Before or at the scope end
Eventual	Sometime in the future

Notes: "wrt" and 'R" stand for "with respect to" and "replica."

Broadly speaking, it helps to think as follows. Consistency models are more or less strict depending on how eagerly they propagate the update of a record to the volatile memory hierarchy of the nodes with replicas. Persistency models are more or less strict depending on how eagerly they persist the update to the NVMs of the nodes with replicas. This separation of concerns provides an intuitive way to plug the framework of persistency models into the framework of consistency models, creating DDP models.

Table 1 shows the VP and DP of an update in the different consistency and persistency models, respectively, of the "Data-Consistency and Memory-Persistency Models" sidebar. The models are listed from more to less strict. In the following, for a given variable, we call "replica nodes" all the nodes that contain a copy of the record. More details are found in our MICRO-2021 paper.[4]

DDP PROTOCOLS FOR MODERN HARDWARE

To understand the tradeoffs in DDP models, we design new distributed protocols for 25 DDP models that pairwise combine the persistency and consistency models of Table 1. We target modern data center architectures, where nodes communicate with low-latency advanced remote direct memory access (RDMA) and use NVM for persistency. In this setting, where a round trip between nodes takes single-digit microseconds, and data persistency can be obtained in a few hundred microseconds, we design protocols that emphasize low latency. Our protocols have no single leader—i.e., a client read or write request can be received and processed at *any node*.[5] Moreover, on reception of a client's write, a node broadcasts

messages to all the other replica nodes, instead of sending a message that sequentially visits all the other replica nodes. Our protocols build on Hermes.[5] Following Hermes' terminology, we call *coordinator* the node that receives the client's read/write request for a record and initiates the transaction; *followers* are all the other nodes with a replica of the record. The protocols are described in our MICRO-2021 paper.[4]

TRADEOFFS BETWEEN DDP MODELS

The different DDP models provide different tradeoffs between durability, performance, intuition provided to the programmer, programmability, and implementability. Durability refers to how capable the system is to retain a consistent state after a failure that causes the loss of some or all the volatile state. Performance depends on three main factors: 1) the speed of reads, 2) the speed of writes, 3) and the volume of traffic generated.

Programmer intuition is determined by what values a read can return. In particular, we consider whether the system supports monotonic reads and/or nonstale reads.[6] A system supports monotonic reads if, given two system-wide reads to the same variable, the later read always provides the same or a later version of the record that the earlier read provided. A system fails to provide nonstale reads if a read that follows a write system-wide fails to provide the value of the write. The most obvious case is when a failure between the write and read causes the loss of the written version. Intuitive systems support both monotonic and nonstale reads.

Programmability refers to the developer's ease of writing the application. For example, if the developer has to include annotations for transactions or scopes, programmability is hurt. Finally, implementability refers to the simplicity of the algorithms in the model. For example, keeping track of the HB histories of writes in the causal consistency model complicates the implementation.

Specific DDP Model Analysis

Table 2 compares 10 representative DDP models: five that bind synchronous persistency, two that bind read-enforced persistency, one that binds eventual persistency, and two that bind scope persistency to consistency models. We consider durability, performance, programmer intuition, programmability, and implementability. In the table, upward, flat, and downward arrows mean high, medium, and low; crosses mean no and tick marks means yes. We represent a DDP model as <consistency model, persistency model>.

Combinations With Synchronous Persistency

Row 1 shows the very strict <Linearizable, Synchronous>. Durability is high because a write does not return until it is persisted in all replica nodes. In terms of performance, writes are not optimized because a client write is not acknowledged until the update is propagated to all replicas and persisted in the replica nodes. Reads are not optimized either because a client read is blocked until an incoming prior write from another node to the same record has updated and persisted all the replicas. For these reasons, even though we can say that the traffic is medium, the overall performance is low. In terms of intuitiveness, this model is highly intuitive because it provides both monotonic reads and nonstale reads. Finally, both programmability and implementability are high.

Row 2 shows <Read-Enforced, Synchronous>, which optimizes writes by allowing the coordinator to acknowledge the client as soon as the coordinator

TABLE 2. Comparing different DDP models.

Consistency Model	Persistency Model	Durability	Performance				Programmer Intuition			Other	
			Wr Opt?	Rd Opt?	Traffic	Overall	Monot. Rds?	Non Stale Rds?	Overall	Program-mability?	Implemen-tability?
1. Linearizable	Synchronous	⇑	✘	✘	⇔	⇓	✔	✔	⇑	⇑	⇑
2. Read-Enfor.		⇔	✔	✘	⇔	⇔	✔	✘	⇔	⇑	⇑
3. Transactional		⇑	✔	✔	⇑	⇑	✔	✔	⇑	⇓	⇓
4. Causal		⇔	✔	✔	⇑	⇑	✔	✘	⇔	⇑	⇓
5. Eventual		⇓	✔	✔	⇓	⇑	✘	✘	⇓	⇑	⇑
6. Linearizable	Read-Enfor.	⇔	✔	✘	⇑	⇔	✔	✘	⇔	⇑	⇑
7. Causal		⇔	✔	✘	⇑	⇑	✔	✘	⇔	⇑	⇓
8. Linearizable	Eventual	⇓	✔	✔	⇓	⇑	✘	✘	⇓	⇑	⇑
9. Linearizable	Scope	⇑	✔	✔	⇑	⇑	✘	✘	⇑	⇓	⇓
10. Transactional		⇑	✔	✔	⇑	⇑	✘	✘	⇑	⇓⇓	⇓⇓

Notes: "opt" means optimized.

sends the update messages to replica nodes. Reads, however, are not optimized and still need to wait until an incoming prior write from another node to the same record is propagated to all the replicas and persisted everywhere. In this model, durability is medium because, if a failure occurs after a write has been acknowledged to the client but before the write has persisted, the write may be lost. Since writes are optimized but reads are not, and the traffic is medium, overall performance is medium. Monotonic reads are guaranteed but not nonstale reads, due to the failure just described: as the system recovers from the failure, a subsequent read will not return the value produced by the lost write. Hence, intuitiveness is medium. Programmability and implementability are high.

Row 3 shows <Transactional, Synchronous>, which is similar to <Linearizable, Synchronous> except that it operates at transaction level. It has high durability—completed transactions are never lost. It optimizes writes through overlapping them inside a transaction, and reads by not stalling them. As a result, although traffic is high due to transaction begin/end messages, its performance is high. It provides both monotonic reads and nonstale reads and, hence, intuitiveness is high. However, programmability is low due to the need to annotate code with transactions, and implementability is low due to the need to implement transactions and their conflict detection and resolution.

Row 4 shows <Causal, Synchronous>, which optimizes both writes and reads. Neither of them stalls: writes are acknowledged to the client as soon as the coordinator sends the update messages, and reads return the latest version in persistent memory. Since the write optimization may result in a write to be lost in a failure, durability is medium. Both reads and writes are fast but the traffic is high because each write carries its HB history. Still, performance is high. Monotonic reads are guaranteed because, even if updates arrive at a follower out of order, the system buffers them and performs them in the order based on their HB history. However, nonstale reads are unsupported because writes can be lost to failures. Hence, intuition is medium. Programmability is high but implementability is low because of the need to buffer and enforce the HB histories.

Row 5 shows <Eventual, Synchronous>. As it provides practically no guarantees on when writes update replicas and persist, its durability is low. It has optimized reads and writes, and low traffic. Hence, performance is high. However, since neither monotonic reads nor nonstale reads are supported, intuitiveness is low. Programmability and implementability are high.

Relaxing Persistency

As we relax persistency and go from <Linearizable, Synchronous> to <Linearizable, Read-Enforced> in row 6, we optimize writes by acknowledging the client after the replicas are updated but not yet persisted. Reads are not optimized due to read-enforced persistency. Writes can be lost in a failure. Consequently, durability decreases to medium. Performance, however, increases to medium. Furthermore, since nonstale reads are not guaranteed in a failure, intuitiveness decreases to medium.

Row 7 shows <Causal, Read-Enforced>, which mostly optimizes writes over <Linearizable, Synchronous>. Because of this change, and despite its high traffic due to HB history transfers, its performance is high. However, since writes can be lost, durability is medium and nonstale reads are not supported. As a result, intuitiveness is medium. Implementability is low because of the need to keep HB histories.

Further relaxing persistency to <Linearizable, Eventual> in row 8 creates a system with both read and write optimization but neither monotonic nor nonstale reads. The result is low durability, high performance, and low intuitiveness.

Finally, we consider scope persistency. In <Linearizable, Scope> (row 9) and <Transactional, Scope> (row 10), we have systems with high durability: in a volatile storage failure, the state of all the completed scopes is recovered, and that of those partially executed is discarded. Within a scope, writes are optimized because they do not serialize their persists, and so are reads, which can read before the scope persists. As a result, despite the higher traffic caused by scope-persist messages, performance is high. Neither monotonic reads nor nonstale reads are guaranteed: on a failure, a group of writes may be lost because the scope did not persist. However, intuitiveness is still high because either the whole scope survives or no part of it does. Finally, both programmability and implementability are low due to the need to mark and support scopes. Further, both properties are worse if scopes are combined with transactions (row 10).

PERFORMANCE EVALUATION

We extensively compare the performance of our 25 DDP models for multiple key-value store applications. We model a platform with five servers of 20 cores each. Figure 1 presents the average throughput of the different DDP models. Each bar represents the combination of a consistency model (*x*-axis) with a persistency model (legend). All bars are normalized to the case of <Linearizable, Synchronous>. We find that models with linearizable consistency have the

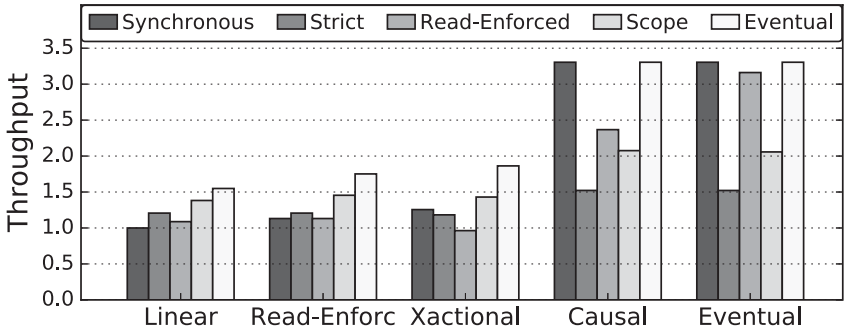

FIGURE 1. Throughput of the different DDP models.

lowest throughput, while those with causal and eventual consistency have the highest (often 2×–3× higher). Those with transactional consistency fail to deliver high throughput when conflicts are high.

Typically, throughput is inversely correlated with mean read and write latencies. Models with causal and eventual consistency have low read and write latencies. The exception is the combinations with strict persistency. The latter stalls writes until the updates are persisted everywhere. Models with transactional consistency have high write latencies, both because of conflicts—a request will not be satisfied until the transaction restarts and completes—and because writes bunch up at transaction end. This is especially the case for strong persistency models, such as strict and synchronous. On the other hand, some models with read-enforced consistency have high read latency. This is because, by enabling more write overlapping than linearizable consistency, they induce more NVM pressure, causing reads to stall longer for writes to persist.

IMPLICATIONS FOR APPLICATIONS

Application developers choose the consistency model according to their needs. The evaluation in our MICRO-2021 paper[4] suggests which persistency models should go with which consistency models and, therefore, which DDP models to use. We now summarize the main insights.

Latency-sensitive applications that can tolerate a certain level of data staleness, such as web browsing and social networking, often use eventual consistency.[7] In this case, using synchronous persistency is a good choice in terms of performance, programmability, and implementability (see Table 2).

For consistency-sensitive applications that require bounded staleness but can accept modest latencies, such as certain web search services,[1] stricter consistency models, such as read-enforced consistency, are good choices. In this case, combining them with scope or eventual persistency results in high throughput and low tail latency. Some of these applications aggregate data from thousands of anonymous users and, therefore, the loss of a certain amount of recent data is acceptable.

MODELS COMBINING STRONG CONSISTENCY WITH WEAK PERSISTENCY, OR WEAK CONSISTENCY WITH STRONG PERSISTENCY ARE TYPICALLY BEST.

Applications that need both reasonable consistency guarantees and high performance, such as photo sharing and news readers,[8] often use causal consistency. In this case, we suggest to use synchronous persistency. Our evaluation shows that causal consistency delivers some of the best performance of all cases in combination with multiple persistency models. Therefore, developers can select the appropriate persistency model based on the durability requirements of their application.

Applications that require transactional guarantees, such as Google's globally distributed Spanner database[9] use a form of transactional consistency. This consistency model can deliver high throughput, but its performance suffers if transaction conflicts are frequent. In this case, using read-enforced persistency is not a good choice, since reads end up suffering long stalls. Other persistency models, such as scope or eventual should be used.

Many systems use hybrid consistency models—e.g., linearizable or read-enforced consistency in a local cluster, and eventual consistency across the

entire distributed system in a data center.[10] In this case, our results suggest using scope or eventual persistency for the local cluster, and synchronous persistency across the system.

Generally, we find that causal consistency combined with either synchronous or eventual persistency is highly competitive, and robust to increases in number of clients, network latencies, and write traffic. Beyond this, models combining strong consistency with weak persistency, or weak consistency with strong persistency are typically best.

Irrespective of the DDP model, a recovery algorithm is invoked on a crash. The complexity of the recovery is higher in the weaker models than in the stricter ones. For example, strict models, such as linearizable plus synchronous have a simple recovery process because all nodes have the same persistent view of the data. On the other hand, weaker DDP models, such as those with eventual consistency or persistency may need an advanced recovery algorithm.

FUTURE RESEARCH DIRECTIONS AND OPPORTUNITIES

This work can be extended in multiple ways to spur research in computer architecture and distributed systems. This section describes some research directions and opportunities enabled by our work.

Extending the Framework

Future work can extend our framework of consistency and persistency models in many ways. One possible extension is to propose new memory-persistency models for distributed systems. Another extension is to combine the persistency models described with the many other existing *data-consistency* models.

Another possible research line is to develop better protocols for these DDP models. In our MICRO-2021 paper,[4] we built our protocols based on Hermes.[5] However, other protocol designs are possible, which offer different performance and complexity tradeoffs. Finally, another avenue worth exploring is using new implementations of the DDP protocols. Our implementations use RDMAs. One can also use remote procedure calls or services that manage the replication and recovery in distributed systems, such as Zookeeper.[11]

Opportunities for Computer Architecture Research

This article shows that the arrival of NVM together with advances in high-performance networks impacts remote communication and persistency. Additional architectural advances will introduce new tradeoffs in this area.

Specifically, current RDMA functionality is limited. For example, an RDMA request to persist data is acknowledged before it can provide any guarantee that the data have been successfully persisted in NVM. As a result, researchers have proposed RDMA extensions that are needed to facilitate NVMs.[12] Such proposals include flushing data from volatile storage to NVM, as well as performing a write directly to NVM and only acknowledging this write when it has fully persisted in NVM.

AN IMPORTANT CONTRIBUTION OF THIS WORK IS TO UNDERSTAND HOW THE VISIBILITY POINT AND THE DURABILITY POINT OF AN UPDATE INTERACT WITH EACH OTHER IN DIFFERENT DDP MODELS.

The introduction of programmable network interface cards (SmartNICs) provides an opportunity to offload operations and decision making from the central processing unit (CPU) to the SmartNICs—freeing up the CPU for compute tasks. DDP protocols can take advantage of advanced hardware support in the SmartNICs. For example, DDP protocols often require a node to perform fast checks of record metadata before sending responses back to other nodes. Such checks could be handled in the SmartNIC. This functionality requires enhancing the RDMA with new primitives and the NIC with new hardware.

Some DDP protocols, such as those that utilize transactions can use fairly sophisticated support in the SmartNIC. For example, they can identify potential conflicts between the different transactions. Further research is needed to understand and design the SmartNIC hardware requirements.

Development of Applications

Our work provides a tradeoff analysis that can be used by application developers to select the appropriate DDP model for their applications. Applications like databases or third party services like Zookeeper need to implement the DDP models and provide a way for the user to select one of them.

Advanced APIs may be developed that allow for a more effective use of these models. Programs may be able to select the DDP model under which to treat individual or groups of client requests. For instance, the API can indicate that a write should be persisted before replying to the client and that no other client

should be able to read this value until the persist operation has completed.

In addition, the detailed description of the DP for each DDP model enables the introduction of more efficient distributed recovery mechanisms for distributed systems. Such mechanisms can take advantage of the DP to identify the latest update that can be recovered from a record after a crash, minimizing the traffic and coordination needed between the nodes of the system.

CONCLUDING REMARKS

This article has introduced a set of DDP models and low-latency protocols to support them. These models and protocols are widely applicable and not tied to a specific programming framework or distributed system environment. This flexibility makes the DDP models and protocols amenable to be incorporated into a wide variety of services and distributed systems.

An important contribution of this work is to understand how the visibility point and the durability point of an update interact with each other in different DDP models. Such insights can help developers of applications and runtime systems to build more efficient distributed software—i.e., software that attains the desired combination of performance, consistency, and durability properties—and to reason about the correctness of their distributed software.

REFERENCES

1. P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Comput. Surv.*, vol. 49, no. 1, Jun. 2016, Art. no. 19, doi: 10.1145/292696.
2. A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, Apr. 2014, pp. 401–414. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevic
3. S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proc. 41st Annu. Int. Symp. Comput. Archit.*, 2014, pp. 265–276. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665712
4. A. Kokolis, A. Psistakis, B. Reidys, J. Huang, and J. Torrellas, "Distributed data persistency," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2021, pp. 71–85.
5. A. Katsarakis *et al.*, "Hermes: A fast, fault-tolerant and linearizable replication protocol," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2020, pp. 201–217, doi: 10.1145/3373376.3378496.
6. A. Ganesan, R. Alagappan, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Strong and efficient consistency with consistency-aware durability," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 323–337. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/ganesan
7. M. K. Aguilera and D. B. Terry, "The many faces of consistency," *IEEE Data Eng. Bull.*, vol. 39, pp. 3–13, 2016.
8. S. A. Mehdi *et al.*, "I can't believe it's not causal! scalable causal consistency with no slowdown cascades," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, Mar. 2017, pp. 453–468. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi
9. J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 261–264.
10. H. Lu *et al.*, "Existential consistency: Measuring and understanding consistency at facebook," in *Proc. 25th Symp. Operating Syst. Principles*, 2015, pp. 295–310.
11. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Conf. USENIX Ann. Tech. Conf.*, 2010, p. 11.
12. SNIA, "NVM PM remote access for high availability (Technical White Paper)," Tech. Rep., May 2019. [Online]. Available: https://www.snia.org/sites/default/files/technical_work/Whitepapers/NVM-PM-Remote-Access-for-High-Availability.pdf

APOSTOLOS KOKOLIS is a Ph.D. candidate with the University of Illinois at Urbana-Champaign, Champaign, IL, 61801, USA. He builds systems for high performance that integrate nonvolatile memory technologies. His research interests include computer architecture, memory systems, and distributed systems. Contact him at kokolis2@illinois.edu.

ANTONIS PSISTAKIS is a Ph.D. candidate with the University of Illinois at Urbana-Champaign, Champaign, IL, 61801, USA. His research interests include parallel and distributed systems, and computer architecture. Psistakis received an M.Sc. degree in computer science and engineering from the University of Crete, Heraklion, Greece. Contact him at psistaki@illinois.edu.

BENJAMIN REIDYS is a Ph.D. student with the University of Illinois at Urbana-Champaign, Champaign, IL, 61801, USA. His research interests include memory and storage systems with an emphasis on datacenter scale. Reidys received a B.Sc. degree in computer science and mathematics from Virginia Tech, Blacksburg, VA, USA. He is a Graduate Student Member of IEEE. Contact him at breidys2@illinois.edu.

JIAN HUANG is an assistant professor with the University of Illinois at Urbana-Champaign, Champaign, IL, 61801, USA. His research interests include computer systems, systems architecture, memory and storage systems, distributed systems, systems security, and especially the intersections of them. Huang received a Ph.D. degree in computer science from Georgia Tech, Atlanta, GA, USA. He is a Member of IEEE. Contact him at jianh@illinois.edu.

JOSEP TORRELLAS is the Saburo Muroga professor of computer science with the University of Illinois at Urbana-Champaign, Champaign, IL, 61801, USA. His research interests include computer architectures for parallelism, energy efficiency, programmability, and security. Torrellas received a Ph.D. degree from Stanford University, Stanford, CA, USA. Contact him at torrella@illinois.edu.