Reliable Distributed Storage

Article *in* Computer · May 2009

DOI: 10.1109/MC.2009.126 · Source: OAI

CITATIONS
62

READS
338

4 authors:

Gregory Chockler
University of Surrey
107 PUBLICATIONS **2,793** CITATIONS
SEE PROFILE

Rachid Guerraoui
Swiss Federal Institute of Technology in Lausanne
844 PUBLICATIONS **22,134** CITATIONS
SEE PROFILE

Idit Keidar
Technion – Israel Institute of Technology
283 PUBLICATIONS **5,550** CITATIONS
SEE PROFILE

Marko Vukolic
EURECOM
54 PUBLICATIONS **3,242** CITATIONS
SEE PROFILE

Reliable Distributed Storage

Gregory Chockler, Rachid Guerraoui, Idit Keidar, Marko Vukolić

Abstract— **Storage is nowadays commonly provided as a *service*, accessed by clients over a network. A *distributed storage service* provides clients with the abstraction of a single reliable shared storage device, using a collection of possibly unreliable computing units. Algorithms that implement this abstraction vary according to many dimensions, including their complexity, the consistency semantics they provide, the number and types of failures they tolerate, as well as the assumptions they make on the underlying architecture. Certain tradeoffs have to be addressed and the choices are tricky.**

Index Terms— **Distributed algorithms, storage, reliability, failures.**

MOTIVATION

Distributed storage systems are becoming increasingly popular with the advent of Storage Area Network (SAN) and Network Attached Storage (NAS) technologies, as well as the increasing availability of cheap commodity disks. These systems cope with the loss of data using replication, whereby data is stored in multiple basic storage units (disks or servers), called *base objects*.

An important goal for such systems is providing *high availability*: the storage should remain available at least whenever any single server (or disk) fails; sometimes more failures are tolerated. The *resilience* of a distributed storage system is defined as the number t out of n base objects (servers or disks) that can fail without forgoing availability and consistency. The resilience level dictates the service's availability. For example, if every server has an uptime of 99%, then by storing the data on a single server, one gets *two nines* of availability. If the data is replicated on 3 servers (i.e., $n = 3$), and the solution tolerates one server failure ($t = 1$), then the service availability is close to *four nines* (99.97%).

A popular way to overcome disk failures is using RAID (Redundant Array of Inexpensive Disks) technology [11]. RAID systems, in addition to boosting performance (e.g., using striping), use redundancy (either mirroring or erasure codes) to prevent the loss of data following a disk crash. However, a RAID system is generally a single box, residing at a single physical location, accessed via a single disk controller, and connected to clients via a single network interface. Hence, it is still a single-point-of-failure.

In contrast, a distributed storage system emulates (i.e., provides the abstraction of) a robust shared storage object by keeping copies of it in several places, in order to have data survive complete site disasters. This can be achieved using cheap commodity disks or low-end PCs for storing base objects. It is typical to focus on the abstraction of a *storage object*, that supports only basic *read* and *write* operations by clients, providing provable guarantees. The study of these objects is fundamental, for these are the building blocks for more complex storage systems. Moreover, such objects can be used, for example, to store files, which makes them interesting in their own right.

CHALLENGES

An important challenge one faces when designing a distributed storage system is *asynchrony*. Since clients access the storage over a network (e.g., the Internet, or a mobile network), access delays may be unpredictable. This makes it impossible to distinguish slow processes from faulty ones and forces clients to take further steps possibly before accessing all non-faulty base objects. Whilst a distributed storage algorithm can make use of common-case synchrony bounds to boost performance when these bounds are satisfied, it should not rely on them for its correctness. If chosen aggressively, such bounds might be violated when the system is overloaded or the network is broken. If chosen conservatively, such bounds might lead to slow reactions to failures.

A distributed storage algorithm implements read and write operations in a distributed fashion; that is, by accessing a collection of base objects and processing their responses. Communication can be intermittent and clients may be transient (i.e., they may come and go). Implementing such a storage is however not trivial. Suppose we are implementing a read/write object x that has to remain available as long as at most one base object crashes. Consider a client Alice performing a write operation, writing "I love Bob" to x. If Bob later performs a read operation on x, then Bob must access at least one base object to which Alice wrote in order to read the text. Due to our availability requirement, Bob must be able to find such an object even if one base object fails. The difficulty is that, due to asynchrony, a client can never know for sure whether a base object has really failed, or only appears to have failed due to excessive communication delays. Assume, for example, that Alice writes the text to only one base object, and skips a second base object that appears to her to be faulty albeit it is not (see Fig. 1(a)). Then the base object Alice does write to may eventually fail, removing any record of the text, thus preventing Bob from completing his read. Clearly, Alice must access at least two base objects in order to complete the write. In order to allow Alice to do so when one base object fails, the system should include at least three base objects (assuming two are correct).

Matters become even more complicated if clients or base objects can be *corrupted*; such corruption can happen for various reasons, ranging from hardware defects in disks, through software bugs, to malicious intrusions by hackers (possible when storage is provided as a service over a network). In these cases, it is typical to talk about *arbitrary* (sometimes called Byzantine, or malicious) faults: An entity (client or base object) incurring an arbitrary fault can deviate from the behavior prescribed by its implementation in an arbitrary (unconstrained) manner.

A distributed storage system typically uses access control, so that only legitimate clients access the service. Yet it is desirable for the system to function properly even in the face of password leaks and compromised clients. In this context, it is important to differentiate between clients who are only allowed to read the data (called *readers*), and clients who are allowed to modify it (*writers*). Storage systems usually have many readers but only a few writers (possibly a single writer). Therefore, protection from arbitrary failures of the readers is more important. Moreover, a faulty writer can always write "garbage" into the storage, rendering it useless. Hence, one typically attempts to overcome arbitrary client failures only by readers and not by writers. The latter are assumed to be authenticated and trusted; still, any writer may fail by crashing.

In short, distributed storage algorithms face the challenge of overcoming asynchrony and a range of failures, without deviating significantly from the consistency guarantees and performance of traditional (centralized) storage. Such algorithms vary in several dimensions: in the consistency semantics they provide; in their resilience (number and types of failures tolerated); in their architecture (whether the base objects are simple disks or more complex servers); and in their complexity (e.g., latency). Clearly, there are many tradeoffs: for example, providing stronger consistency or additional resilience impacts complexity.

A Simple Storage Algorithm

The classical algorithm of Attiya, Bar-Noy and Dolev *(ABD)* [3] illustrates the typical modus operandi of distributed storage algorithms. This algorithm overcomes only crash failures, of both clients and base objects. ABD implements a single-writer multi-reader storage abstraction. That is, only one client, Alice for instance, is allowed to write to the storage. Other clients only read. ABD implements *atomic* objects; that is, it gives clients the illusion that accesses to the shared storage are sequential (occurring one client at a time), although in practice many clients are concurrent. In general, ABD tolerates t crash failures out of $n = 2t + 1$ base objects (which is optimal).

The algorithm is invoked by a client wishing to perform a read or write operation, and it proceeds in rounds. In each round, the client sends a message to all base objects and awaits responses. Since t base objects may crash, a client should be able to complete its operation upon communicating with $n - t$ base objects. Due to asynchrony, the client may "skip" a correct albeit slow object when there are actually no failures.

Consider a system with three base objects, of which one may fail ($t = 1, n = 3$). Say Alice attempts to write "I love Bob" to all base objects, but her message to one of them is delayed, and she completes her operation after having written to two. Now Bob performs a read operation, and he also accesses only two base objects. Of these two, at least one was written by Alice. Thus, Bob obtains the text "I love Bob" from at least one base object. However, the second object Bob accesses may be the one Alice skipped, which still holds the old text "I love cheese". How can Bob know which value is the up-to-date one in this case? To this end, Alice generates monotonically increasing *timestamps*, and stores each value along with the appropriate timestamp. For example, the text "I love cheese" is associated with the timestamp 4, and the later text, "I love Bob", with timestamp 7. Thus, Bob returns the text associated with the higher timestamp of the two (see Fig. 1(b)).

More specifically, in ABD, the write(v) operation is implemented as follows: the writer increases its local timestamp ts, and then writes the pair $\langle v, ts \rangle$ to the base objects. Writing is implemented by sending *write-request* messages containing $\langle v, ts \rangle$ to the base objects. Upon receiving such a message, a base object checks if ts is higher than the timestamp stored locally. If it is, the base object updates its local copies to hold v and ts. In all cases, the object replies with an acknowledgment to the writer. When the writer receives acknowledgments from $n - t$ base objects, the write operation completes.

The read operation invokes two rounds: a read round, and a *write-back* round. In the read round, a reader sends a *read-request* message to all base objects. A base object that receives such a request responds with a *read-reply* message including its local copies of v and ts. When the reader receives $n-t$ replies, it selects a value v' and the corresponding timestamp ts', such that ts' is the highest timestamp in the replies. In the write-back round, the reader writes the pair $\langle v', ts' \rangle$ to the base objects, as in the write operation described above.

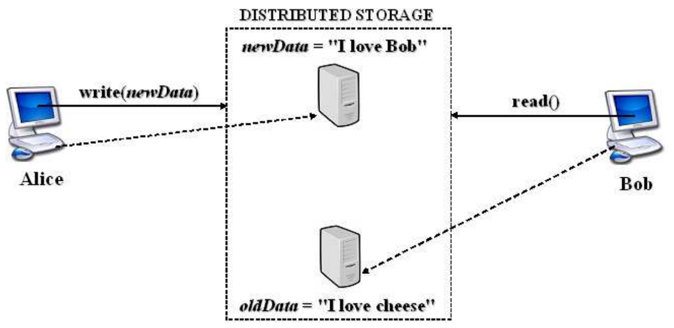
The write-back round is required to ensure atomicity (i.e., that the emulated object is atomic): it guarantees that, once a read returns v', every subsequent reader will read either v' or some later value. Without this round, ABD ensures only weaker semantics, called *regularity* (see *Consistency Semantics* sidebar).

For example, assume Alice begins a write operation, but after she manages to update one base object, her network stalls for a while, and her messages to the remaining base objects are delayed. In the interim, Bob invokes a read operation. Since Alice's operation has been initiated but is incomplete, it can be serialized either before or after Bob's read operation. If Bob encounters the single object Alice updated, then Bob returns the new value, with the highest timestamp. Assume that after Bob completes its operation, another reader, Carol, invokes a read. Carol may skip the single base object that Alice already wrote to. If Bob writes back, then Carol encounters the new value in another base object (since Bob writes to $n-t$), and returns it. But if write-back is not employed, Carol returns the old value. This behavior violates atomicity, because Carol's operation returns an older value than the preceding operation by Bob (see Fig. 1(c)).

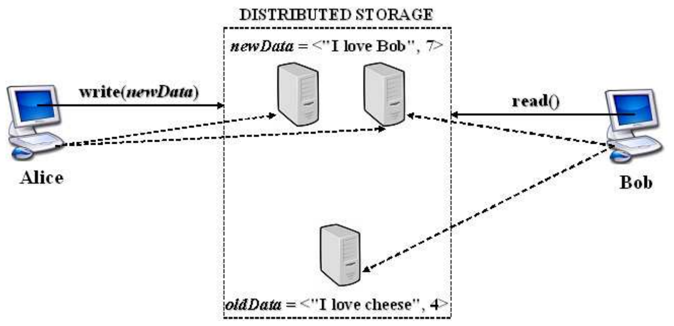
In order to support multiple writers, the write operations can be extended to two rounds. In the first round, a writer collects the latest timestamps from all base objects, selects the highest timestamp, which the writer then increments in the second round. The first round is required to ensure that a new write uses a timestamp higher than every previous write, and is only needed when there are multiple writers. The second round is identical to the original, single-writer, write operation.

Due to the use of monotonically increasing timestamps that may grow indefinitely, ABD's storage requirements are potentially unbounded. However, timestamps typically grow very slowly, and are therefore considered acceptable in practice.

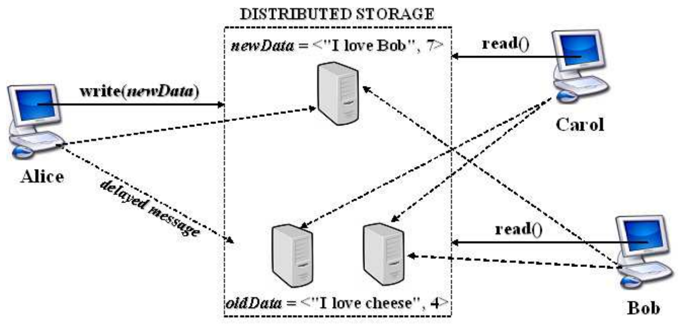
ABD is simple, and yet it achieves many desirable properties: atomicity, unconditional progress to all clients (called *wait-freedom*), and resilience to the maximum possible number of crash failures. However, it does not cope with arbitrary failures.



(a) Bob returns an outdated value as it accesses only one base object.



(b) With an additional base object, Bob is able to return the latest written value.



(c) A write-back is needed if multiple readers are involved.

Fig. 1. Illustrating a simple distributed storage algorithm.

Coping with Arbitrary Base-Object Failures

There are two principal models in which arbitrary failures are considered, differing in the cryptographic mechanisms employed. The first, called the *authenticated* model, employs unforgeable digital signatures. The second model, called *unauthenticated*, makes no use of signatures, and assumes only that the immediate message source may be verified. Arbitrary client failures are much

Consistency Semantics

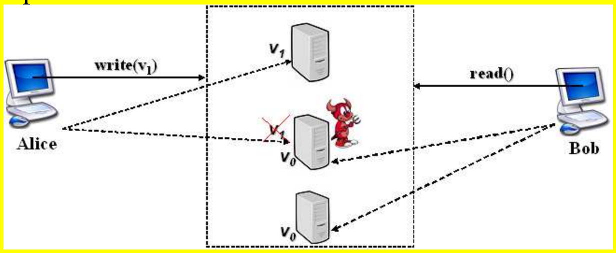
Lamport [8] defines three universally accepted consistency guarantees for a read/write storage abstraction: *safe*, *regular* and *atomic*.

Safe storage ensures that a read that is not concurrent with any write returns the last value written. Unfortunately, *safety* is insufficient for most applications of distributed storage, since a read concurrent with some write may return an arbitrary value. *Regular* storage strengthens safety by ensuring that read always returns a value that was actually written, and is not older than the value written by the last preceding write.

Although regular storage provides sufficient guarantees for many distributed storage applications, it still fails to match the guarantees of the traditional, sequential storage. The latter is captured by the notion of *atomicity*, which ensures the *linearizability* [7] of read/write operations, providing the illusion that the storage is accessed sequentially. Regular storage may fail to achieve such a level of consistency when two reads overlap the same write. This drawback of regular storage is known as *new-old read inversion* (Figure 1(c) depicts an example). Atomic storage overcomes this drawback, by ensuring that a read does not return an older value than returned by a preceding read (in addition to regularity).

Optimal Resilience

A storage implementation is called *optimally resilient* if it requires the minimal number n of base objects in order to tolerate t base object failures (in the given failure model). In case of arbitrary failures, at least $n \geq 3t + 1$ base objects are required in order to tolerate t failures [10]. To illustrate the lower bound, consider the following example for the case that $t = 1$ and $n = 3$:



(1) A shared object is initialized to v_0.

(2) Alice invokes write(v_1), which reaches two of the base objects, but her message to the third base object is delayed due to asynchrony. Alice falsely perceives the third object to be crashed, and the write completes without waiting for this object.

(3) The second base object incurs an arbitrary failure by "losing memory", and reverting to v_0. (Note that this is possible even in the authenticated model, since v_0 was once a valid value). This leaves only the first base object with information about v_1.

(4) Bob invokes a read. Due to asynchrony, Bob perceives the first base object as crashed, and reads v_0 from the other two base objects (the faulty one and the one that Alice skipped). Bob cannot wait for the first base object, since this might have crashed. Therefore, Bob returns an outdated value v_0, violating safety.

easier to deal with in the former: the techniques used are not very different from those used in the simple crash-failure model (see *Arbitrary Failures with Authentication* sidebar), aside for the lower resilience. Namely, in both models, $n = 2t+1$ servers no longer suffice to overcome t arbitrary base object failures (see *Optimal Resilience* sidebar). However, an important drawback of the authenticated model is in the high overhead for computing unforgeable signatures.

In the unauthenticated model, where signatures are unavailable, in order for a read to return a value v, v must appear in at least $t + 1$ responses. This makes achieving optimal resilience (i.e., $n = 3t+1$) quite tricky. Consider the following scenario with $n = 4, t = 1$. Alice invokes write("I love Bob"), which completes after accessing three of the base objects; the fourth appears to be faulty. But of the three base objects that respond, one is really faulty, whereas the one that has not responded is simply slow. In this case, only two correct base objects have stored "I love Bob". Next, Bob invokes a read operation. Bob receives "I love Bob" from one of these, "I love cheese" from the out-of-date object, and "I hate Bob" from the faulty object. In order to ensure progress, Bob does not await the fourth object, which appears as faulty albeit it is not. In this situation, Bob has no way of knowing which of the three values to return. Three recent algorithms address this challenge using different techniques, each making different assumption on the underlying storage.

The first such algorithm is SBQ-L (Small Byzantine Quorums with Listeners) due to Martin, Alvisi and Dahlin [10]. SBQ-L implements multi-writer multi-reader atomic storage, tolerating arbitrary failures of base objects. It uses full-fledged servers that can actively push information to clients. It provides atomicity and optimal-resilience. The basic algorithm can be extended to overcome client failures by having the servers broadcast

Arbitrary failures with authentication

With signatures, the ABD algorithm (Fig. 1) can be simply transformed to handle arbitrary failures of readers and up to t base objects, provided at least $n = 3t+1$ base objects [9]. The writer, before sending value v and a timestamp ts, signs these with its private key. As in ABD, a write returns upon receiving replies from $n - t$ base objects. All readers possess the corresponding public key, with which they can verify that the data was indeed generated and signed by the writer. Readers collect $n-t$ responses from base objects, of which at least one is correct and up-to-date. Thanks to the use of digital signatures, the faulty base object cannot "cook up" a bogus value with a higher timestamp than the latest used by the writer. Therefore, as in ABD, the reader can safely return the highest-timestamped value it sees. In the write-back (second round) of a read operation, readers communicate to base objects the value with the highest timestamp, along with the signature of the writer that base objects verify (to overcome arbitrary reader failures).

updates among them.

SBQ-L addresses the optimal resilience challenge outlined above using two main ideas: First, before a value v is returned by a read operation, v must be confirmed by a at least $n - t$ different base objects. Since a write operation may skip at most t servers, and at most t may be faulty, a value reported $n - t \geq t + 1$ times is always received from at least one correct and up-to-date base object. This high confirmation level also eliminates the need for a write-back phase as in ABD, since once v appears in $n - t$ base objects it cannot be missed by later reads.

At first glance, it may seem impossible to obtain $n - t$ confirmations of the same value, because a write operation must sometimes complete without receiving an acknowledgment from all the correct base objects (as explained above). However, observe that even in this case, the write operation does *send* write requests to all base objects before returning, even if it does not await all acknowledgments. Since all writers are assumed to be correct, some process on the writer's machine can remain active after the write operation returns. SBQ-L uses such a process to ensure that every write request eventually does reach all base objects. The remaining difficulty is that a read operation that samples the base objects before the latest written value reaches all of them, may find them in different states, so the reader cannot be sure to find a value with $n - t$ confirmations.

This is addressed by SBQ-L's second main idea— a *Listeners* pattern, whereby base objects act as servers that *push* data to listening clients. If a read by Bob cannot obtain $n - t$ confirmations of the same value after one read round, then the base objects add Bob to their *Listeners* list. Base objects send all the updates they receive to all the readers in the *Listeners* list. Eventually, every update is propagated to all $n - t$ of the correct base objects, which in turn, forward the updates to the pending readers (*Listeners*), allowing read operations to complete.

One drawback of SBQ-L is that in the writer synchronization phase of a write operation, writers increment the highest timestamp they receive from potentially faulty base objects. Hence, the resulting timestamp might be arbitrarily large and the adversary may exhaust the value space for timestamps. This issue was addressed by Bazzi and Ding [4], who provide an elegant solution to this problem using *Non-skipping timestamps*, whereby writers select the $t + 1^{st}$ highest timestamp instead of simply the highest one. However, this solution sacrifices the optimal resilience of SBQ-L, employing $n = 4t + 1$ base objects.

Tolerating Client Failures

Recall that SBQ-L provides optimal resilience by obtaining $n - t = 2t + 1$ confirmations of a value returned in a read operation. In order to achieve so many confirmations, SBQ-L relies on the fact that every written value is eventually propagated to all correct base objects, either by the writer (which is supposed never to fail), or by active propagation among the base objects. However, in a setting where the writer may fail and base objects are passive disks, there is no way to ensure that the written value always propagates to all correct base objects. Consider a scenario with $n = 4, t = 1$, where Alice writes "I love Bob" to three base objects, two of them correct and one faulty, and then completes the write operation, since she perceives the fourth base object as faulty. Alice's machine then crashes, before ensuring that the update reaches the fourth base object. If the base objects are passive, there is no active process that can propagate the update to the final base object. If now Bob initiates a read operation, Bob should return the new value (to ensure safety), and yet it cannot get more than $2 = t + 1$ confirmations for this value.

In general, algorithms that achieve optimal resilience with passive base objects and tolerate client failures must allow read operations to return after obtaining as few as $t + 1$ confirmations of the returned value. This is one of the main principles employed by Abraham, Chockler, Keidar, and Malkhi (ACKM) [2], who present an optimally resilient single-writer multi-reader algorithm tolerating client failures. Readers are prevented from modifying the state of base objects, and hence an unbounded number of arbitrary reader failures are tolerated. The algorithm stores base objects on passive disks, which support only basic read and write operations. Two variants are presented: one that implements safe storage and ensures wait-freedom, and a second that implements regular storage with a weaker liveness condition, *Finite-Write termination (FW-termination)*. This condition slightly weakens wait-freedom in that a read operation must complete only in executions in which a finite number of write operations is invoked (hence the name). All write operations are ensured to complete.

The write operation takes two rounds in ACKM (with a single writer), and two timestamp-value pairs are stored in each base object— pw (for pre-write), and w (for write). In the first write round, the writer pre-writes the timestamp-value pair, i.e., writes to the base objects' pw field. In the second round, it writes the same timestamp-value pair in the w fields. In each round, the writer awaits $n - t = 2t + 1$ acknowledgments from base objects.

To illustrate, consider Alice writing "I love Bob", and successfully updating two of the three correct base objects plus one faulty object. Once the write is complete, "I love Bob" is stored with some timestamp, e.g., 7, in the pw and w fields of two correct base objects. If Bob now invokes a read round that accesses only $n - t = 3$ base objects, he may encounter only one base object holding ⟨"I love Bob",7⟩ in both the pw and w fields, while one correct base object returns an old value ⟨"I love cheese",4⟩, and a faulty base object returns a fallacious value, ⟨"I hate Bob",8⟩ in both the pw and w fields. This is clearly not sufficient for returning "I love Bob" (at least $t + 1 = 2$ confirmations are required in order to prevent faulty base objects from forging values).

On the other hand, Bob cannot wait for the fourth object to respond, because Bob cannot distinguish this situation from the case that all responses are from correct base objects, and Alice has begun writing "I hate Bob" with timestamp 8 *after* the first base object has already responded to Bob, but before the third did. Since Bob can neither return a value nor wait for more values, it must invoke another read round to gather more information. This is exactly what ACKM does in such situations to ensure regular semantics. If "I hate Bob" was indeed written by Alice, then in the new read round, two correct base objects should already hold ⟨"I hate Bob",8⟩ at least in their pw fields, since otherwise Alice would not have updated the w field of the third object. If an additional base object reports this value (within its pw or w field), then Bob regrettably returns "I hate Bob". On the other hand, if the first two base objects continue to return values with smaller timestamps than 8 (as in the first round), then Bob can know that the third object is lying, and may safely return "I love Bob".

Unfortunately, Bob cannot always return after two rounds, because there is a third possibility: the second and third base objects can return two different newer values (with timestamps exceeding 8). In this case, Bob cannot get the two needed confirmations for any of the values. This scenario can repeat indefinitely if Alice is constantly writing new values, much faster than Bob can read them.

If only safety is required, a read operation can return in a constant number of rounds, at most $t+1$. Basically, if the reader cannot obtain sufficiently many confirmations for any value within $t + 1$ rounds, it can detect concurrency, in which case it can return any value (by safety). If regularity is required, then Bob is guaranteed to obtain sufficiently many confirmations once Alice stops writing, ensuring FW-termination.

Optimizing Latency

Precluding readers from writing allows ACKM to support an unbounded (and unknown) number of readers and tolerate their arbitrary failures. ACKM pays however a price for this: read operations (of the safe storage) require $t + 1$ rounds in the worst-case. It is thus natural question to ask if this latency can be improved if readers are allowed to write. Guerraoui and Vukolić [5] address this question by presenting an optimally resilient regular storage algorithm, GV, in which both reads and writes complete in at most two rounds.

At the heart of GV lies the idea of a *high resolution timestamp*. This is essentially a two-dimensional matrix of timestamps, with an entry for every reader and base object. While reading the latest values from base objects, readers (e.g., Bob and Carol) write their own read-timestamps (incremented once per every read round) to base objects. Writers (e.g., Alice) uses the local copies of Bob's and Carol's timestamps, stored within base objects, to provide their write timestamp with a much *higher resolution*. In the first round of a write, Alice (1) stores the value v along with her own (low resolution) timestamp in the base objects, and (2) gathers copies of Bob and Carol's timestamps from base objects and concatenates these to her own timestamp, which results in a final high-resolution timestamp, $HRts$. Then, in the second round of the write, Alice writes v along with $HRts$. However, to achieve two round read latency, GV trades in storage complexity, by requiring base objects to store an entire history of the shared variable.

The read latency optimization of GV is visible in the corner-case where the system experiences arbitrary failures, asynchrony and read/write concurrency. In a more common case, where the system behaves synchronously and there is no read/write concurrency, ACKM provides optimal latency of a single round.

To extend this desirable performance in the common case from regular (ACKM) to atomic storage, the general *refined quorum system (RQS)* [6] framework of Guerraoui and Vukolić can be used. This framework defines necessary and sufficient intersection properties of quorums that need to be accessed in atomic storage implementations in order to achieve optimal best-case latencies of read/write operations. Given an available set of base objects and an adversary structure (RQS distinguishes crash from arbitrary failures and is not bound to the threshold failure model), RQS outputs the set of quorums such that, if any such quorum is accessed, read/write operations can complete in a single round. For example, in the case with $3t + 1$ base objects (optimal resilience), a single round latency can be achieved only if all base objects are accessed. This explains why it is difficult to combine low latency with optimal resilience in atomic storage implementations (e.g., in SBQ-L), in contrast to implementations that employ more base objects (e.g., $4t + 1$ or more).

Summary

Building storage systems in a distributed fashion is very appealing; disks are cheap, and the availability of data can be significantly increased. Distributed storage algorithms can be tuned to provide a high degree of consistency, availability and resilience, while at the same time inducing a very small overhead compared to a centralized unreliable solution.

Not surprisingly, and as we emphasized in the paper, combining desirable storage properties incurs various tradeoffs. Besides, it is worth mentioning that practical distributed storage systems face many other challenges, including survivability, interoperability, load balancing and scalability (see e.g., [1]).

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating systems principles*, pages 59–74, 2005.

[2] I. Abraham, G. V. Chockler, I. Keidar, and D. Malkhi. Byzantine disk Paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[4] R. Bazzi and Y. Ding. Non-skipping timestamps for Byzantine data storage systems. In *Proceedings of the 18th International Symposium on Distributed Computing*, volume 3274/2004 of *Lecture Notes in Computer Science*, pages 405–419, Oct 2004.

[5] Rachid Guerraoui and Marko Vukolić. How fast can a very robust read be? In *Proceedings of the 25th annual ACM symposium on Principles of distributed computing*, pages 248–257, 2006.

[6] Rachid Guerraoui and Marko Vukolić. Refined quorum systems. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, pages 119–128, 2007.

[7] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[8] L. Lamport. On interprocess communication. *Distributed computing*, 1(1):77–101, May 1986.

[9] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[10] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Symposium on Distributed Computing*, volume 2508/2002 of *Lecture Notes in Computer Science*, pages 311–325, Oct 2002.

[11] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD Record*, 17(3):109–116, 1988.

Gregory Chockler *is a research staff member in the Distributed Middleware group at the IBM Haifa Research Laboratory. He holds Ph.D., M.Sc., and B.Sc. degrees from the Hebrew University of Jerusalem. He was a postdoctoral associate with the Theory of Distributed Systems group, MIT/CSAIL and an adjunct lecturer at Hebrew University. Contact him at chockler@il.ibm.com.*

Rachid Guerraoui *is a professor of computer science at EPFL. He has a Ph.D. in computer science from the University of Orsay and has been affiliated with HP Labs and MIT. He is coauthor of Introduction to Reliable Distributed Programming (Springer-Verlag, 2006). Contact him at rachid.guerraoui@epfl.ch.*

Idit Keidar *is a professor at the department of Electrical Engineering at the Technion, and a recipient of the national Alon Fellowship for new faculty members. She holds Ph.D., M.Sc., and B.Sc. degrees from the Hebrew University of Jerusalem. She was a postdoctoral research associate at MIT's laboratory for Computer Science. Contact her at idish@ee.technion.ac.il.*

Marko Vukolić *is a Ph.D. student in computer science at EPFL. He received a dipl.ing. degree in electrical engineering from the University of Belgrade. Contact him at marko.vukolic@gmail.com.*