Reliable Distributed Storage

Gregory Chockler, IBM Haifa Research Laboratory
Idit Keidar, Technion
Rachid Guerraoui, EPFL
Marko Vukolić, EPFL

A distributed storage service lets clients abstract a single reliable shared storage device using a collection of possibly unreliable computing units. Algorithms that implement this abstraction offer certain tradeoffs and vary according to dimensions such as complexity, the consistency semantics provided, and the types of failures tolerated.

With the advent of storage area network (SAN) and network attached storage (NAS) technologies, as well as the increasing availability of cheap commodity disks, distributed storage systems are becoming increasingly popular. These systems use replication to cope with the loss of data, storing data in multiple basic storage units—disks or servers—called *base objects*. Such systems provide high availability: The stored data should remain available at least whenever any single server or disk fails; sometimes they tolerate more failures.

A distributed storage system's resilience is defined as the number t 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 a 99 percent uptime, storing the data on a single server can provide two nines of availability. If the data is replicated on three servers ($n = 3$, for example), and the solution tolerates one server failure ($t = 1$), service availability approaches four nines: 99.97 percent.

A popular way to overcome disk failures uses a redundant array of inexpensive disks (RAID).[1] In addition to boosting performance with techniques such as striping, RAID systems use redundancy—either mirroring or erasure codes—to prevent loss of data following a disk crash. However, a RAID system generally contains 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 still constitutes a single point of failure.

In contrast, a distributed storage system emulates a robust shared storage object by keeping copies of it in several places, so that data can survive complete site disasters. The systems can achieve this using cheap commodity disks or low-end PCs for storing base objects. Researchers typically focus on abstracting a storage object that supports only basic read and write operations by clients, providing provable guarantees. The study of these objects is fundamental, for they provide the building blocks for more complex storage systems. Moreover, such objects can be used to store files, for example, which makes them interesting in their own right.

CHALLENGES

Asynchrony presents an important challenge developers must face when designing a distributed storage system. Because clients access the storage over connections such as the Internet or a mobile network, access delays can be unpredictable. This makes it impossible to distinguish slow processes from faulty ones and forces clients to take further steps, possibly before accessing all nonfaulty base objects. While a distributed storage algo-

Published by the IEEE Computer Society
0018-9162/08/$25.00 © 2008 IEEE

rithm can use 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 by accessing a collection of base objects and processing their responses. Communication can be intermittent and clients transient. Implementing such storage is, however, nontrivial.

Suppose we implement a read/write object x that must 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 to read the text he must access at least one base object to which Alice wrote. Given our availability requirement, Bob must be able to find such an object even if one base object fails.

The difficulty arises from asynchrony—a client can never know whether a base object has really failed or only appears to have failed because of excessive communication delays. Assume, for example, that Alice writes the text to only one base object and skips a second base object that appears faulty to her even though it is not, as Figure 1 shows.

The base object Alice writes to could eventually fail, removing any record of the text and preventing Bob from completing his read. Clearly, Alice must access at least two base objects to complete the write. To let Alice 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, which becomes possible when the system provides storage as a network service. In these cases, researchers typically talk about arbitrary—sometimes called *Byzantine*, or *malicious*—faults: A client or base object entity incurring an arbitrary fault can deviate from the behavior its implementation prescribes in an unconstrained manner.

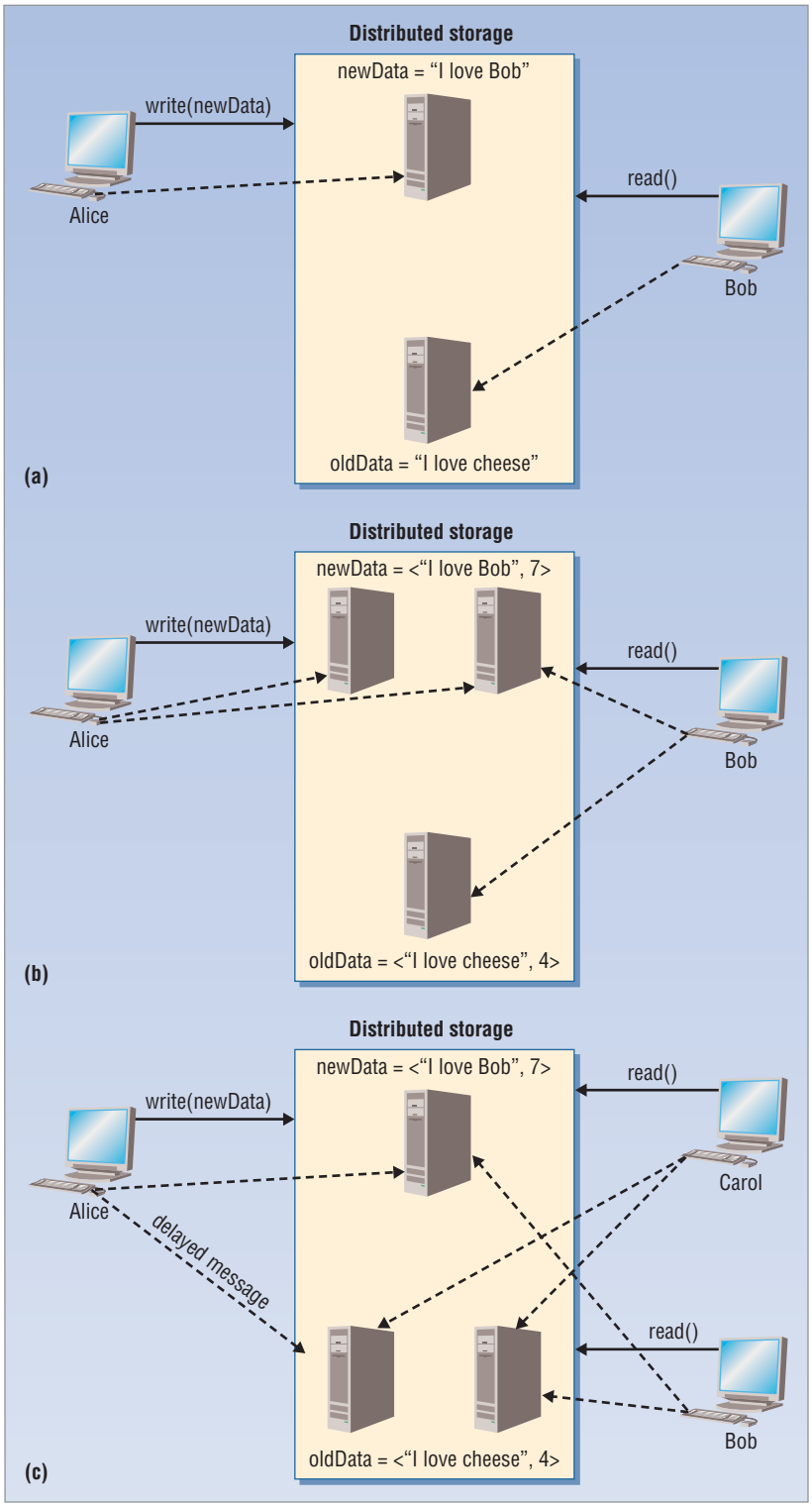

Figure 1. Simple distributed storage algorithm. (a) Bob returns an outdated value as the operation accesses only one base object. (b) With an additional base object, Bob returns the latest written value. (c) A write-back is needed if multiple readers are involved.

Consistency Semantics

Leslie Lamport[1] 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 distributed storage applications, 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 traditional, sequential storage. The latter is captured by the notion of atomicity, which ensures the linearizability[2] of read/write operations, providing the illusion that the storage is accessed sequentially. Regular storage might 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. 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.

References

1. L. Lamport, "On Interprocess Communication," *Distributed Computing*, vol. 1, no. 1, 1986, pp. 77-101.
2. M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Programming Languages and Systems*, vol. 12, no. 3, 1990, pp. 463-492.

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 allowed to read only the data, called *readers*, and clients allowed to modify it, called *writers*. Storage systems usually have many readers but only a few writers, and possibly only one. Therefore, protection from arbitrary reader failures is more important. Moreover, a faulty writer can always write garbage into the storage, rendering it useless. Hence, developers typically attempt to overcome arbitrary client failures only by readers and not by writers. Developers assume the latter to be authenticated and trusted; still, any writer could 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:

- consistency semantics provided,
- resilience (number and types of failures tolerated),
- architecture (whether the base objects are simple disks or more complex servers), and
- complexity (latency, for example).

Clearly, these algorithms pose many tradeoffs—for example, providing stronger consistency or additional resilience affects complexity.

SIMPLE STORAGE ALGORITHM

Hagit Attiya, Amotz Bar-Noy, and Danny Dolev's classic ABD algorithm[2] shows the typical modus operandi of distributed storage algorithms. It overcomes only crash failures of both clients and base objects. ABD implements a single-writer multireader storage abstraction. That is, only one client—Alice, for example—can write to storage. Other clients only read. ABD implements atomic objects, giving clients the illusion that accesses to the shared storage are sequential and occur one client at a time, though in practice many clients perform accesses concurrently. In general, ABD tolerates an optimal t crash failures out of $n = 2t + 1$ base objects.

A client seeking to perform a read or write operation invokes the algorithm, and it proceeds in rounds. In each round, the client sends a message to all base objects and awaits responses. Since t base objects might crash, a client should be able to complete its operation upon communicating with $n - t$ base objects. Due to asynchrony, the client might skip a correct albeit slow object when no actual failures occur.

Consider a system with three base objects, of which one might 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 also accesses only two base objects. Of these two, Alice wrote to at least one. Thus, Bob obtains the text "I love Bob" from at least one base object. However, the second object Bob accesses might be the one Alice skipped, which still holds the old text, "I love cheese." So that Bob knows which value is the up-to-date one, Alice generates monotonically increasing timestamps and stores each value along with the appropriate timestamp.

For example, the text "I love cheese" is associated with 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, as Figure 1b shows.

More specifically, in ABD, the write(v) operation is implemented as follows: the writer increases its local timestamp ts, then writes the pair (v, ts) to the base

objects. Writing is implemented by sending *write-request* messages containing (v, ts) 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 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 $(v'; ts')$ to the base objects, as in the write operation already described.

The write-back round ensures atomicity—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*, as the "Consistency Semantics" sidebar describes.

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 might 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, as Figure 1c shows.

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 and 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.

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

Arbitrary Failures with Authentication

With signatures, the ABD algorithm can be simply transformed to handle arbitrary failures of readers and up to t base objects, provided at least $n = 3t + 1$ base objects.[1] 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 writer did indeed generate and sign the data.

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 produce a bogus value with a higher timestamp than the latest the writer used. Therefore, as in ABD, the reader can safely return the highest time-stamped value it sees. In the second-round write-back 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.

Reference

1. D. Malkhi and M. Reiter, "Byzantine Quorum Systems," *Distributed Computing*, vol. 11, no. 4, 1998, pp. 203-213.

ABD is simple, 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.

COPING WITH ARBITRARY BASE-OBJECT FAILURES

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

Arbitrary client failures are much easier to deal with in the former: aside from the lower resilience, the techniques used differ little from those used in the simple crash failure model, as the "Arbitrary Failures with Authentication" sidebar describes. In both models, $n = 2t + 1$ servers no longer suffice to overcome t arbitrary base object failures, as the "Optimal Resilience" sidebar explains. However, the high overhead for computing unforgeable signatures presents an important drawback of the authenticated model.

Optimal Resilience

A storage implementation is called optimally resilient if it requires the minimal number n of base objects 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 to tolerate t failures.[1] To illustrate the lower bound, consider Figure A and 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 asynchrony delays her message to the third base object. 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. 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, he perceives the first base object as crashed and reads v_0 from the other two base objects, the faulty one and the one Alice skipped. Bob cannot wait for the first base object because it might have crashed. Therefore, Bob returns an outdated value v_0, violating safety.

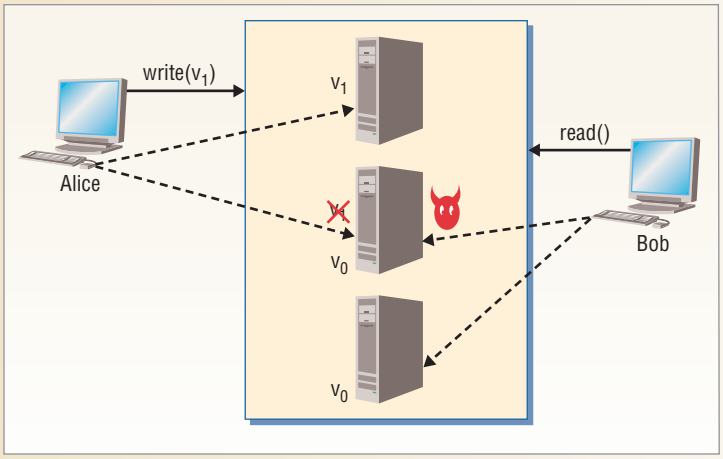

Figure A. Due to asynchrony, Bob perceives the first base object as crashed. He cannot wait for the first base object because it might have crashed. Therefore, Bob returns an outdated value, violating safety.

Reference

1. J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine Storage," *Proc. 16th Int'l Symp. Distributed Computing*, LNCS 2508, Springer-Verlag, 2002, pp. 311-325.

In the unauthenticated model, where signatures are unavailable, for a read to return a value v, v must appear in at least $t + 1$ responses. This makes achieving optimal resilience (such as $n = 3t + 1$) 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 really is 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. He 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.

To ensure progress, Bob does not await the fourth object, which appears faulty but is not. In this situation, Bob cannot know which of the three values to return. Three recent algorithms address this challenge using different techniques, each making a different assumption about the underlying storage.

SBQ-L algorithm

The first such algorithm is Small Byzantine Quorums with Listeners.[3] SBQ-L implements multiwriter/multireader atomic storage, tolerating arbitrary base object failures. It uses full-fledged servers that can actively push information to clients and provides atomicity and optimal resilience. The basic algorithm can be extended to overcome client failures by having the servers broadcast updates among them.

SBQ-L addresses the optimal resilience challenge using two main ideas. First, before a read operation returns value v at least $n - t$ different base objects must confirm it. Since a write operation can skip at most t servers, and at most t might 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 ABD's write-back phase, since once v appears in $n - t$ base objects, later reads cannot access it.

At first glance, it might 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. However, even in this case, the write operation sends 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 might find them in different states, so the reader cannot be sure to find a value with $n - t$ confirmations.

SBQ-L's second main idea addresses this with 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, 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 might exhaust the value space for timestamps. Rida A. Bazzi and Yin Ding[4] addressed this issue, providing an elegant solution using nonskipping timestamps, whereby writers select the t + first-highest timestamp instead of simply the highest one. However, this solution sacrifices the optimal resilience of SBQ-L, employing $n = 4t + 1$ base objects.

Because readers are prevented from modifying the base objects' state, the ACKM algorithm can tolerate an unbounded number of arbitrary reader failures.

ACKM algorithm

Recall that SBQ-L provides optimal resilience by obtaining $n - t = 2t + 1$ confirmations of a value returned in a read operation. To achieve so many confirmations, SBQ-L relies on every written value eventually being propagated to all correct base objects, either by the writer (which supposedly never fails) or by active propagation among the base objects.

However, in a setting where the writer might fail and passive disks are base objects, 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, then completes the write operation because 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 Bob now initiates a read operation, he 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 Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi's algorithm (ACKM),[5] an optimally resilient single-writer multireader algorithm that tolerates client failures. Because readers are prevented from modifying the base objects' state, the algorithm can tolerate an unbounded number of arbitrary reader failures. The algorithm stores base objects on passive disks, which support only basic read-and-write operations.

ACKM has two variants: one that implements safe storage and ensures wait freedom, and a second that implements regular storage with a weaker liveness condition, finite-write 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. All write operations are ensured to complete.

With a single writer, the write operation takes two rounds in ACKM, and stores two timestamp-value pairs in each base object, *pw* (for prewrite) and *w* (for write). In the first-write round, the writer prewrites the timestamp-value pair by writing 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—7, for example—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 might 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 to prevent faulty base objects from forging values.

On the other hand, Bob cannot wait for the fourth object to respond because he 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, the algorithm must invoke another read round to gather more information.

This is exactly what ACKM does in such situations to ensure regular semantics. If Alice did indeed write

"I hate Bob," 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, 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, Bob can know that the third object is lying, and can safely return "I love Bob."

Unfortunately, Bob cannot always return after two rounds, because a third possibility exists: 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 constantly writes new values much faster than Bob can read them.

If the process requires only safety, a read operation can return in a constant number of rounds, at most $t + 1$. Basically, if the reader cannot obtain sufficient confirmations for any value within $t + 1$ rounds, it can detect concurrency, in which case it can return any value by safety. If it requires regularity, Bob is guaranteed sufficient confirmations once Alice stops writing, ensuring finite-write termination.

GV algorithm

Precluding readers from writing lets ACKM support an unbounded and unknown number of readers as well as tolerate their arbitrary failures. ACKM pays a price for this, however: Read operations of the safe storage require $t + 1$ rounds in the worst case. It is thus natural to ask if allowing readers to write can improve this latency. In the optimally resilient Rachid Guerraoui and Marko Vukolić (GV) storage algorithm, both reads and writes complete in at most two rounds.[6]

The idea of a high-resolution timestamp lies at GV's heart. 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 write their own read timestamps, incremented once per every read round, to base objects. Writers use 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 first stores the value *v* along with her own low-resolution timestamp in the base objects. Then she gathers copies of Bob's 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 write's second round, 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.

GV's read latency optimization 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 system can use GV's general refined quorum system (RQS)[7] framework. This framework defines the necessary and sufficient intersection properties of quorums that need to be accessed in atomic storage implementations 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), the system can achieve a single-round latency only if it accesses all base objects. This explains why combining low latency with optimal resilience in atomic storage implementations such as SBQ-L is difficult, in contrast to implementations that employ more base objects, such as $4t + 1$ or more.

Building distributed storage systems is appealing: Disks are cheap and the system can significantly increase data availability. Distributed storage algorithms can be tuned to provide high consistency, availability, and resilience, while at the same time inducing a small overhead compared to a centralized unreliable solution.

Not surprisingly, combining desirable storage properties incurs various tradeoffs. In addition, practical distributed storage systems face many other challenges, including survivability, interoperability, load balancing, and scalability.[8]

Acknowledgment

Idit Keidar's research is partially supported by Google.

References

1. D.A. Patterson, G. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *ACM SIGMOD Record*, vol. 17, no. 3, 1988, pp. 109-116.
2. H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," *J. ACM*, vol. 42, no. 1, 1995, pp. 124-142.
3. J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine Storage," *Proc. 16th Int'l Symp. Distributed Computing*, LNCS 2508, Springer-Verlag, 2002, pp. 311-325.

4. R. Bazzi and Y. Ding, "Non-Skipping Timestamps for Byzantine Data Storage Systems," *Proc. 18th Int'l Symp. Distributed Computing*, LNCS 3274, Springer-Verlag, 2004, pp. 405-419.
5. I. Abraham et al., "Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory," *Distributed Computing*, vol. 18, no. 5, 2006, pp. 387-408.
6. R. Guerraoui and M. Vukolić, "How Fast Can a Very Robust Read Be?" *Proc. 25th Ann. ACM Symp. Principles of Distributed Computing*, ACM Press, 2006, pp. 248-257.
7. R. Guerraoui and M. Vukolić, "Refined Quorum Systems," *Proc. 26th Ann. ACM Symp. Principles of Distributed Computing*, ACM Press, 2007, pp. 119-128.
8. M. Abd-El-Malek et al., "Fault-Scalable Byzantine Fault-Tolerant Services," *Proc. 20th ACM Symp. Operating Systems Principles*, ACM Press, 2005, pp. 59-74.

Gregory Chockler *is a research staff member in the Distributed Middleware group at the IBM Haifa Research Laboratory. His research interests include all areas of distributed computing, spanning both theory and practice. Chockler received a PhD from the Hebrew University of Jerusalem, where he was an adjunct lecturer. Contact him at chockler@il.ibm.com.*

Idit Keidar, *a professor at the Department of Electrical Engineering at Technion, is a recipient of the national Alon Fellowship for new faculty members. Her research interests include distributed computing, fault tolerance, and concurrency. Keidar received a PhD from the Hebrew University of Jerusalem. Contact her at idish@ee.technion.ac.il.*

Rachid Guerraoui *is a professor of computer science at EPFL and coauthor of* Introduction to Reliable Distributed Programming. *His research interests include distributed algorithms, languages, and systems. Guerraoui received a PhD in computer science from the University of Orsay. Contact him at rachid.guerraoui@epfl.ch.*

Marko Vukolić *is a PhD student in computer science at EPFL. His research interests include distributed computing, security, and concurrency. Vukolić received a dipl. ing. in electrical engineering from the University of Belgrade. Contact him at marko.vukolic@gmail.com.*