# Rethinking Eventual Consistency

Philip A. Bernstein
Microsoft Research
Redmond, WA 98052, USA
philbe@microsoft.com

Sudipto Das
Microsoft Research
Redmond, WA 98052, USA
sudiptod@microsoft.com

## ABSTRACT

There has been a resurgence of work on replicated, distributed database systems to meet the demands of intermittently-connected clients and of disaster-tolerant databases that span data centers. Many systems weaken the criteria for replica-consistency or isolation, and in some cases add new mechanisms, to improve partition-tolerance, availability, and performance. We present a framework for comparing these criteria and mechanisms, to help architects navigate through this complex design space.

## Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software – *Distributed systems.*

## General Terms: Design

**Keywords:** Eventual consistency, replication.

## 1. INTRODUCTION

Data replication is a widely-used technique for spreading read and write load across servers and improving availability. Research on the topic began in the 1970's. Commercial database management systems (***DBMS***) started supporting replication in the late 1980's. Today, replication functionality is found in file systems, cache managers, queue managers, and cloud storage systems.

Ideally, replication is transparent to the clients. This is captured in two well-known correctness criteria: ***one-copy serializability (1SR)*** [1] and ***linearizability*** [13]. A system is 1SR if it behaves like a serial processor of multi-step *transactions* on a one-copy database. A system is linearizable if it behaves like a serial processor of single-step *operations* on a one-copy database. However, these strict correctness goals are impractical in many situations.

The three basic techniques for synchronizing replicated data are ***primary copy*** [1][25], ***multi-master***, and ***quorum consensus***. Although each can achieve one-copy semantics, there is a cost due to an unavoidable tradeoff between consistency, availability, and partition-tolerance. This tradeoff was first observed in 1977 by Rothnie and Goodman [22], and later popularized as Brewer's **CAP** conjecture [5], which was proved by Gilbert and Lynch [12]. It states that a replicated, distributed data store can have at most two of Consistency of replicas (or *copies*), Availability of writes, and Partition tolerance. That is, a system can provide: (i) consistency of available copies and write availability if there are no partitions; (ii) consistency of available copies during a partition with at most one partition available for writes; or (iii) write availability during a partition but copies in different partitions will be inconsistent.


*SIGMOD'13,* June 22–27, 2013, New York, New York, USA.


Most distributed systems need to cope with network partitions, and write availability is essential for many Internet-facing applications. Therefore, systems have to offer weaker forms of consistency. One popular form is ***eventual consistency***. It states that in an updatable replicated database, eventually all copies of each data item converge to the same value.

The origin of eventual consistency can be traced back to Thomas' majority consensus algorithm [30]. The term was coined by Terry et al. [29] and later popularized by Amazon in their Dynamo system, which supported only eventual consistency [9]. Although eventual consistency enables high availability, it increases application complexity to handle inconsistent data. For instance, in a seat reservation application implemented on an eventually-consistent DBMS, two clients can reserve the last seat. The copies will agree on the result, but the result is incorrect. Complex application logic is needed to avoid the error.

Many consistency criteria have been proposed to improve on eventual consistency, such as causal consistency [16][18], timeline consistency [7], eventually consistent transactions [6], parallel snapshot isolation [26], session consistency [29], and prefix consistency [28]. It is a confusing design space. In this tutorial, we simplify the design space by: (i) characterizing consistency criteria; (ii) describing mechanisms to support these criteria; (iii) organizing them in a taxonomy; and (iv) summarizing their strengths and weaknesses.

Past surveys of replication include ones by Bernstein and Newcomer [1] (Chapter 9), Davidson et al. [8], Kemme and Alonso [14], Saito and Shapiro [23], and Terry [27]. Compared to these, our tutorial presents a different framework for reasoning about replication and covers newly-published techniques. Our framework reasons about tradeoffs between consistency and availability in a partitioned network, to help readers understand which consistency and read-write ordering constraints a system can support while still being available. We focus only on the consistency and isolation of operations; atomicity and durability properties are out of scope.

## 2. RETHINKING CONSISTENCY

There are three classic approaches to replicated data: primary copy, multi-master, and quorum consensus. In all approaches, a ***client update*** arrives at one copy, is processed there, and is forwarded as ***downstream updates*** to the other copies. In primary copy, one copy, the ***primary***, processes all client updates. Multi-master allows all copies to process client updates. In quorum consensus a client update completes only after a quorum of copies has processed it. Quorum consensus can use primary copy or multi-master.

The next subsection discusses techniques to achieve eventual consistency of downstream updates. These techniques constrain only the relative order of writes. Section 2.2 describes other constraints on the order of writes and on the order of reads with respect to writes, which we call ***admissible executions***. We

discuss these admissibility constraints first for single read and write operations and then for multi-operation transactions.

## 2.1 Eventual Consistency

We classify techniques for applying downstream updates to ensure eventual consistency into 3 categories: ***commutative downstream operations***, ***ordered updates***, and ***custom convergent merges***.

### *2.1.1 Commutative Downstream Operations*

If all downstream updates commute, then eventual consistency is guaranteed. ***Thomas' write rule*** [30] is the earliest generic mechanism to make downstream operations commutative. Each client write is assigned a timestamp. Each copy $x_c$ of data item $x$ stores the timestamp of the last write applied to $x_c$. A downstream update of $x$ is applied to $x_c$ only if its timestamp is greater than that of $x_c$. That is, the highest timestamp wins at every copy.

Another mechanism for eventual consistency is the use of convergent and commutative replicated data types (***CRDTs***) [24]. For example, an ordinary set is not a CRDT, since its add and remove operations do not commute: for an element $e$, [add($e$), add($e$), remove($e$)] ≢ [add($e$), remove($e$), add($e$)]. Instead, we can use the CRDT *counting set*, which associates a count with each element in a set, where add($e$) increments $e$'s count, remove($e$) decrements it, and element $e$ exists if its count is positive. Write operations on a counting set commute. Shapiro et al. [24] present many other CRDT's over registers, sets, and graphs.

Commutative updates can be applied anytime, anywhere, and in any order. However, they expose a constrained and unfamiliar programming model, they do not address inconsistencies that arise from read-write ordering, and they do not cover all operation types.

### *2.1.2 Ordered Updates*

If all updates are applied in the same order at all copies, then the copies will be eventually consistent. This order can be ***total*** or ***partial***. In primary copy, downstream updates are applied in the same order as their corresponding client updates at the primary. Logging is a second approach to totally-order updates, where the order in which operations are written to a log is the order they are applied to copies. A third approach is consensus algorithms, to reach agreement on the ordering, and group communication abstractions, such as totally-ordered broadcast, where the update order is based on the message order.

An early implementation of ordered updates was done in Bayou [21], a multi-master system with a primary "committing copy." An update can be applied at any copy, but its order at the committing copy determines its official order. After that order is determined, it is reordered if necessary at copies where it previously executed.

A major advantage of total ordering is that it simplifies reasoning. However, each approach to it has disadvantages. In primary copy, the primary is a bottleneck and point-of-failure. Similarly, in logging the tail of the log is a bottleneck and point-of-failure. Consensus algorithms have higher message overhead, more complex implementations and higher update latency.

***Vector clocks*** (or ***version vectors***) are a classic approach to partial ordering [11][20]. Each copy in multi-master assigns a monotonically increasing version number to each client update. A vector clock is an array of version numbers, one per copy. Vector clock $vc_2$ **is later than** vector $vc_1$ if $vc_1[i] \leq vc_2[i]$ for all copies $i$, and $vc_1[j] < vc_2[j]$ for some copy $j$. Vector clocks can be used to partially order updates, trim the prefix of the update logs, and identify the state that a client update depends on [15]. However, major challenges arise in efficient maintenance of these vectors as the number of copies grows and when copies are added or removed.

### *2.1.3 Custom Convergent Merges*

An application-specified procedure can be used to merge updates into a single update that can then be applied to the copies. Such a merge procedure takes two versions of an object and creates a new one. For eventual consistency, the merges must be commutative and associative. Such approaches are used in groupware systems such as collaborative editing [10]. Custom merges enable concurrent execution of conflicting operations without establishing a total order. However, application-specific logic is needed to define such functions and is hard to generalize.

## 2.2 Admissible Executions

Admissible execution criteria constrain read-write, write-read, and write-write order. We classify admissibility criteria into two broad categories: single read and write operations and multi-operation transactions. Admissibility criteria for single operations comprise ***causality constraints*** and ***session constraints***. Admissibility criteria for transactions are called ***isolation constraints***.

### *2.2.1 Single Read and Write Operations*

#### *2.2.1.1 Causality Constraints*

A sequence of operations on each copy is causally consistent if it preserves ***session order*** and ***reads-from order*** [16]. Session order preserves the order of operations from a session across all copies. For instance, suppose user 1 stores a photo and then a link to it. If user 2 reads the link, then causality requires her to see the photo as well. Reads-from order ensures that if a read $r[x]$ in session $V$ reads the value of $x$ written by a write $w[x]$ in session $S$, then $w[x]$ causally precedes every operation in $V$ that follows $r[x]$. Causality is typically enforced by dependency tracking and vector clocks.

A common way to enforce causal consistency is to tag each operation with a vector clock that describes a superset of the earlier operations it depends on and for each copy to maintain a vector clock that describes its state. An operation is executed at a copy only if its dependencies are satisfied.

By itself, causality does not imply that copies are eventually consistent, since there need not exist a causal relationship between updates to two copies of the same data item. Still, most algorithms that enforce causal consistency add a mechanism to ensure eventual consistency [18]. An example is [15], which also offers stronger synchronization options: *forced operations*, which execute in the same order with respect to each other; and *immediate operations*, which execute in the same order with respect to all other operations.

#### *2.2.1.2 Session Constraints*

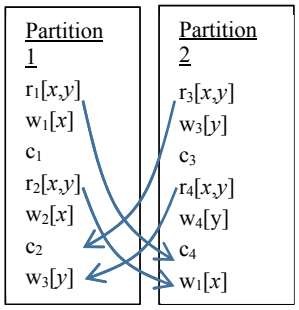
Session constraints encompass consistency in the context of a single client session. Examples include ***read-your-writes***, ***monotonic reads***, ***monotonic writes***, ***consistent prefix***, and ***bounded staleness*** [27][29]. Read-your-writes requires a read to see all preceding writes in the same session. Monotonic reads means that successive reads of a data item in a session do not return older versions. Monotonic writes means that writes from a session are applied in the same order on all copies. Consistent prefix requires that a copy's state only reflects writes that represent a prefix of the entire write history. Bounded staleness ensures that a read does not return a version of an item older than a specified threshold.

Session constraints can be implemented by tracking the writes that precede each read and write in a session. Such explicit tracking is expensive. A more practical approach is for the client session to maintain a version vector for the last item read and one for the last item written. This is a more compact representation of the session constraints but is conservative and hence might cause operations to be delayed even though they are safe to run.

### *2.2.2 Multi-operation Transactions*

Isolation constraints characterize correct interleavings of reads and writes. Examples include 1SR, parallel snapshot isolation, and revision diagrams. 1SR requires that every history is equivalent to a serial history where each transaction writes *all* copies of its write set [1]. Primary copy with two-phase locking is commonly used to support 1SR; the mechanism's complexity is in detecting and recovering from failures.

Parallel snapshot isolation (***PSI***) [26] is a variant of snapshot isolation (***SI***) that allows executions of concurrent programs executing non-conflicting writes that run SI. A PSI history may not have an equivalent serial SI history. For example, the PSI execution in Figure 1 is not equivalent to any SI execution. One benefit of PSI is that transactions only need access to secondary copies of the read-only set. This results in better availability during a partition and better performance during normal operation with geo-replication. However, it is weaker than both 1SR and SI.




- Partition 1 has *x*'s primary
- Partition 2 has *y*'s primary
- Partitions 1 and 2 are each snapshot isolated
- But the result is not equivalent to $T_1T_2T_3T_4$ or $T_3T_4T_1T_2$, where the second two transactions read the snapshot produced by the first two

**Figure 1 Parallel snapshot isolated execution that is not SI**

Revision diagrams [6] present a checkin-checkout model typical of version control systems where there is a mainline. A fork defines a new private snapshot in a branch that can process writes locally. A join of a branch to its parent causes all updates on the branch to be applied to the parent (and eventually to the mainline). Operations are pure reads and writes, and writes never fail. Similar to CRDTs, this model presents an unfamiliar programming experience.

Li et al. [17] propose RedBlue consistency which divides transactions into two classes, blue and red. Each blue one commutes with all other transactions and can run in different orders on different copies. Red ones must run in the same order on all copies. Li et al. show how to transform a red transaction into a blue one, by splitting it into a read-only transaction that generates a request to run a blue transaction. For example, replace a red transaction that calculates interest at each copy by a read-only transaction that calculates interest at one copy, which generates a blue one that adds that interest to every copy.

Lloyd et al. [19] support multi-operation read-only transactions and write-only transactions with a guarantee of causal consistency. They leverage these restrictions over read-write 1SR transactions to improve latency and partition-tolerance.

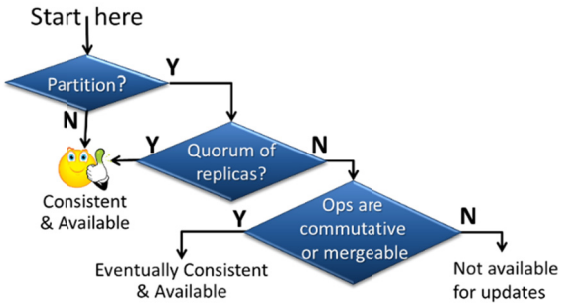



**Figure 2 A framework to reason about weak consistency**

## 3. ANALYZING THE CAP TRADEOFFS

Figure 2 summarizes our framework to reason about weak consistency guarantees. Start at the diamond labeled "Partition." If there is no partition, then there are well known ways to ensure the system is available and consistent. The consistency level chosen trades off an application's isolation and performance requirements.

If there is a partition, a quorum of copies based on primary copy or quorum consensus can offer consistent and available operation, since it knows it has the one and only quorum. Every other partition knows it is a *minority partition*, that is, it does not have a quorum, and hence in general should not allow writes. However, writes can be allowed in special cases. Whether or not writes are allowed, a reconfiguration step might be required after a partition, e.g., to establish that a quorum is accessible, to choose a primary copy, and to catch up stale copies.

What if a partition P does not contain a quorum, or if multi-master is being used (where effectively each copy is a partition)? Then it is still possible to be consistent and available by exploiting the semantics of operations. One approach is to allow only updates that commute and have each partition log them locally during a network partition. When the network partition is healed, each server applies the logs that were captured during the network partition. If updates do not commute, then one can use application-specific merging logic to ensure copies converge, as in [8][9][10]. This allows multi-master operation during a partition when a total order on updates is not enforceable.

Table 1 summarizes the feasible constraints and operations that a system can support during a partition while being available and eventually consistent. A "✓" means the system can remain available while enforcing the constraint, and an "**X**" means the opposite. The first column lists the admissibility criteria. The other columns are divided into two scenarios: one where the client remains connected to the same server after a failure and one where it migrates to a different server. Each group is further subdivided into two columns. Primary copy and quorum-based techniques for the minority partition are grouped in one column; they both disallow writes and use similar techniques to enforce admissibility constraints for reads. Therefore, entries in these columns refer only to read availability. By contrast, multi-master allows writes to all copies, so entries in that column refer to read and write availability. We now explain the entries in each row, one by one.

If the client remains connected to the same server, then a client's read is guaranteed to see all its earlier writes. Thus read-your-writes can be supported. However, if the session migrates to another server S, then the client's earlier writes might not have

propagated yet to S. Thus, read-your-writes can only be supported if the client session caches its writes locally and reads are served locally if the server version is stale. This is depicted by a "**?W**" since satisfying the constraint is contingent on the writes being cached.

When the session remains connected to a server, enforcing monotonic writes is straightforward. If the session migrates, primary copy disables writes and monotonicity of earlier writes is preserved. However, with multi-master, if the session migrates, the client session must cache its writes locally and re-apply them at the new server if its version is stale.

**Table 1. Consistency-availability tradeoff when partitioned.**

| Admissibility criteria | Session maintains connection to server | | Session migrates to another server | |
|---|---|---|---|---|
| | Read availability for Primary copy & Quorum-based in Minority partition | Read & write availability in Multi-master | Read availability for Primary copy & Quorum-based in Minority partition | Read & write availability in Multi- master |
| Read-your-writes | ✓ | ✓ | ✓ ?W | ✓ ?W |
| Monotonic writes | ✓ | ✓ | ✓ | ✓ ?W |
| Bounded staleness | X | X | X | X |
| Consistent prefix | ✓ | X | ✓ | X |
| Monotonic reads | ✓ | ✓ | ✓ ?R | ✓ ?R |
| Causality | ✓ | ✓ | X | X |

Bounded staleness cannot be guaranteed in a minority partition P, because P cannot determine its staleness. In particular, P does not know how many writes (if any) executed elsewhere, or when the last one ran, while it was disconnected from the quorum partition.

A consistent prefix can be enforced with primary copy or quorum consensus in a minority partition, since they disallow applying new updates. Consistent prefix cannot be enforced with multi-master either, since a total order on all writes is not known.

If the client session remains connected to the same server after a network partition, enforcing monotonic reads is automatic. However, if the session migrates to a different server, the version of data at the newly-connected server might be stale. Monotonic reads can be supported only if the client session caches the values returned by its reads and serves repeated reads locally if the server version is stale. This is depicted by a "**?R**" in the table.

Causality can be enforced with high availability only if the client session does not migrate to another copy. To handle session migration, it is tempting to cache reads and writes locally. But this is not enough since transitive dependencies across client sessions can break causality. For example, suppose a client session $S_1$ wrote data item $x$ ($w_1[x]$) and then $y$ ($w_1[y]$). Then $S_2$ read $y$ ($r_2[y]$) and wrote $z$ ($w_2[z]$). Hence, $w_1[x]$ causally precedes $w_2[z]$. So if $S_2$ loses its server connection, reconnects elsewhere, and reads $x$, it must see the result of $w_1[x]$. To handle this, the client session would need to cache all such operations on which it is transitively dependent, which is impractical.

In all cases of "**?R**" and "**?W**" in Table 1, even if the client does not cache all of the relevant operations, it can maintain a synopsis of the dependencies, such as operations' timestamps. After a session migration, it can check whether it is safe to execute the relevant operation. For example, with read-your-writes, a client can maintain the timestamp of its last write to each data item. Before reading the new copy, it can check whether its previous writes arrived. The same technique can be used to test for causality, using vector clocks, as in [15][18][19].

Since reads do not commute with writes, commutative writes are not enough to enforce linearizability during a network partition. For example, suppose two copies each execute increment[$x$] and then r[$x$]. There is no equivalent linear sequence of the four operations in which both reads get the same value as in the original execution.

Our discussion so far focused on single read and write operations. Now consider the world of multi-operation transactions. Figure 2 still applies for reasoning about eventual consistency of writes. With transactions, we have another issue: characterizing which isolation constraints are enforceable. Consider a minority partition. If reads and writes are allowed, then 1SR is unenforceable, because reads in one partition might conflict with writes in another. For example, if the sequence r[$x$] w[x] executes in two partitions, there is no one-copy serial order because neither transaction read the other's output.

If a failure requires a client to migrate its session from a quorum partition (which enforces 1SR) to a minority partition, it must stop performing writes. However, it can continue to execute reads, since data in the minority partition is consistent with a 1SR execution.

Commutative or mergeable operations ensure eventual consistency but do not help isolation, especially in the presence of reads. We therefore need to consider isolation levels weaker than 1SR. One isolation level that can be supported is *read committed*, since a transaction can read any committed data, no matter how stale or inconsistent it might be. A further step is to support *snapshot reads*, which ensures each transaction reads consistent data, though again it is possibly stale. Snapshot Isolation (SI) or Parallel Snapshot Isolation (PSI) can also be supported if all updates are commutative, since first-writer-wins is irrelevant in the absence of conflicts. But they cannot be supported with non-commutative writes, since writes may execute in different partitions, in which case the conflict will not be detected.

If operations are not commutative or mergeable, and if we are not using weak transaction isolation levels, then the best we can do is enforce operation-level admissibility criteria discussed in Table 1.

# 4. CONCLUDING REMARKS

When there is a partition, it is not a binary choice to give up either consistency or availability—a system can give up a little of each. For instance, a system supporting PSI can remain available to writes of data items whose primary is accessible while being unavailable to other writes. We discussed the different tradeoffs for different operation types and the consequences of ordering constraints and admissibility criteria.

Another approach is to ensure that every atomic operation preserves consistency, and design the application to depend only on causal consistency. For instance, when uploading a photo to an album, first upload the photo and then add the link to the album. Similarly, for an e-commerce application, assemble the order in a shopping cart and then place the order. Unfortunately, it is hard to encode all application logic such that causal consistency is enough. In these cases, transactions are very useful. For instance, exchanging, purchasing, or bartering items requires each party to

be credited and debited atomically. Similar scenarios include maintaining referential integrity, and using queued transactions where a queue and a database must both be updated. Although some of these scenarios can enforce these constraints without transactions by using dependency tracking and vector clocks, more error cases arise due to partial failures.

Another useful tradeoff is consistency vs. latency. By allowing a read or write to complete before it has attained a quorum, latency improves, but some reads will see stale data. Probabilistic bounds on staleness for such cases are presented in [3].

As future research, one interesting direction is to design encapsulated solutions that offer good isolation for common scenarios. Examples are CRDTs and convergent merges for non-commutative operations. Another direction is scenario-specific patterns, such as compensations and queued transactions, which can be leveraged to achieve high availability while providing consistency that applications can reason about.

# 5. GOALS OF THIS TUTORIAL

This paper summarizes a tutorial that targets database researchers and system designers with a basic understanding of transactions. Some knowledge of replication mechanisms is helpful but not essential. For novices, the tutorial offer a survey of consistency criteria and mechanisms for synchronizing replicated data. For experts, it explains tradeoffs between criteria, and presents a framework to reason about them.

# 6. BIOGRAPHICAL SKETCHES

**Philip A. Bernstein** is a Distinguished Scientist at Microsoft and an Affiliate Professor at Univ. of Washington. His research interests are transaction processing and data integration. He is an Editor-in-Chief of The VLDB Journal, an ACM Fellow, a winner of the SIGMOD Innovations Award, and a member of the National Academy of Engineering.

**Sudipto Das** is a Researcher at Microsoft. His research interests are in scalable, distributed, and multi-tenant DBMSs for cloud platforms. He received the CIDR 2011 Best Paper award and UC Santa Barbara's Lancaster Dissertation Award for 2012.

## Acknowledgements

We are grateful to Sameh Elnikety, Alan Fekete, and Doug Terry for suggested improvements to earlier drafts.

# 7. REFERENCES

[1] Alsberg, P. A., and Day, J. D.: A principle for resilient sharing of distributed resources. In ICSE, pp. 562-570, 1976.

[2] Attar, R., Bernstein, P.A, and Nathan Goodman: Site Initialization, Recovery, and Backup in a Distributed Database System. *IEEE Trans. Soft. Eng*. 10(6), pp. 645-650, 1984.

[3] Bailis, P., Venkataraman, S., Franklin, M. J., Hellerstein, J. M., and Stoica, I.: Probabilistically Bounded Staleness for Practical Partial Quorums. *PVLDB* 5(8), pp. 776-787, 2012.

[4] Bernstein, P. A. and Newcomer, E.: Principles of Transaction Processing, *Morgan Kaufmann*, 2nd ed., 2009.

[5] Brewer, E. A.: Towards Robust Distributed Systems (abstract). In PODC, p. 7, 2000.

[6] Burckhardt, S., Leijen, D., Fähndrich, M.,Sagiv, M.: Eventually Consistent Transactions, In ESOP, pp. 67-86, 2012.

[7] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D., and Yerneni., R.: PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.* 1(2), pp. 1277-1288, 2008.

[8] Davidson, S. B., Garcia-Molina, H. and Skeen, D.: Consis-tency in a Partitioned Network: a Survey. *ACM Comput. Surv.* 17(3), pp. 341-370, 1985.

[9] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W.: Dynamo: Amazon's highly available key-value store. In SOSP, pp. 205-220, 2007.

[10] Ellis, C. A. and Gibbs, S. J.: Concurrency control in Groupware systems. In SIGMOD, pp. 399-407, 1989.

[11] Fischer, M. J. and Michael, A.: Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. In PODS, pp. 70-75, 1982.

[12] Gilbert, S. and Lynch, N. A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33(2), pp. 51-59, 2002.

[13] Herlihy, M. P. and Wing, J. M.: Linearizability: a Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12(3), pp. 463-492, 1990.

[14] Kemme, B. and Alonso, G.: Database Replication: a Tale of Research across Communities. *PVLDB*, 3(1), pp. 5-12, 2010.

[15] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S.: Providing high availability using lazy replication. *ACM Trans. Comput. Syst.* 10 (4), 360-391, 1992.

[16] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Comm. ACM,* 21(7), pp. 558-565, 1978.

[17] Li, C., Porto, D., Clement, A., Gehrke, J., Preguic, N., and Rodrigues, R.: Making Geo-Replicated Systems Fast if Possible, Consistent when Necessary. OSDI, pp. 265-278, 2012.

[18] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. SOSP, pp. 401-416, 2011.

[19] Lloyd, W., Freedman, M.J., Kaminsky, M. and Andersen, D.G.: Stronger Semantics for Low-Latency Geo-Replicated Storage. NSDI '13, pp. 313-328, 2013.

[20] Parker Jr., D. S., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C. : Detection of Mutual Inconsistency in Distributed Systems. *IEEE Trans. Software Eng* 9(3), pp. 240-247, 1983.

[21] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In SOSP, Oct. 1997.

[22] Rothnie. J. B., and Goodman, N.: A Survey of Research and Development in Distributed Database Management. In VLDB, pp. 48-62, 1977.

[23] Saito, Y. and Shapiro, M.: Optimistic replication. *ACM Comput. Surv.* 37(1), pp. 42-81, 2005.

[24] Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Convergent and Commutative Replicated Data Types. *Bulletin of the EATCS*, No.104, pp. 67-88, 2011.

[25] Stonebraker, M. and Neuhold, E. J.: A Distributed Database Version of INGRES. Berkeley Workshop, pp. 19-36, 1977.

[26] Sovran, Y., Power, R., Aguilera, M. K., and Li, J.: Transactional Storage for Geo-replicated Systems. In SOSP, pp. 385-400, 2011.

[27] Terry, D. B.: Replicated Data Management for Mobile Computing. *Morgan Claypool Publishers*, 2008.

[28] Terry, D.B.: Replicated Data Consistency Explained Through Baseball, MSR-TR-2011-137, http://research.microsoft.com

[29] Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M.J., Theimer, M.M., Welch, B. B.: Session guarantees for Weakly Consistent Replicated Data. In PDIS, pp. 140-149, 1994.

[30] Thomas, R. H.: A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems,* 4(2), pp. 180–209, 1979.