Declared Causality in Wide-Area Replicated Storage

Kyle Lady
University of Michigan
kylelady@umich.edu

Minkyong Kim
IBM Research
minkyong@us.ibm.com

Brian Noble
University of Michigan
bnoble@umich.edu

***Abstract*—Wide-area replicated storage systems are increasingly popular, despite significant shortcomings. Such systems allow for geographic placement of replicas near clients at a global scale, but they struggle to balance consistent global views with update performance. Eventual consistency—best-effort propagation of updates—often fails to meet expectations for update order. Causal consistency provides stronger ordering guarantees, but it can order updates unnecessarily, resulting in decreased performance.**

Understanding which updates require ordering relies on input from the application, as it alone knows the semantics of its computation. In this paper, we report on our initial efforts at providing such an application programming interface, and we show through a simple workload the potential benefits and costs of this approach. We also describe our future plans with application-declared causality.

I. Introduction

Wide-area, replicated storage systems have found increasing favor, but their use brings increasing frustration. Geographic distribution provides a variety of fault-tolerance properties. It is also possible to place replicas so that far-flung clients have nearby access, which reduces latency for many operations.

Unfortunately, wide-area distribution presents a fundamental challenge: the latency between replicas has material implications for the underlying speed of the overall system. Worse, the latency tail grows several orders of magnitude larger than the average [1], and these long-tailed distributions create outsized impacts on user-visible services [2].

Such latency patterns have significant implications for the consistency of updates in wide-area replicated storage systems. Many systems simply offer *eventual consistency*, allowing updates to arrive at each replica on a best-effort basis [3], [4]. However, this creates frustration for developers and users alike, as updates known at one replica are not seen at another.

To rectify this, Eiger [5] added *causal consistency* to an eventually-consistent wide-area replicated database. In essence, any read or write that *could have* influenced a later update is guaranteed to happen before that update at all locations. This is an appealing model, as it captures many programmer expectations for behavior, but it is not without its flaws. In particular, the presence of some nodes with orders-of-magnitude increases in latency significantly slows the entire system due to its causality chain.

It is important to recognize that causality is, in many cases, a stronger guarantee than is needed by the application using wide-area replicated storage. First, most updates do not need to be committed until they are *externalized* in some way, because until that point no other entity is expecting the update's existence [6]. More generally, even though a write *could have* influenced a later update, it is possible that it did not *actually* do so. Such sitautions allow for out-of-order commitments with no adverse effects to program behavior.

In order to provide for safe out-of-order commitments, the underlying storage system needs to know what updates do and do not influence later ones. This requires programmer annotation in some form. Such annotation has been used to allow mobile devices to use multiple connections [7], using explicit atomicity and happens-before declarations. Earlier, annotations of lockset coverage were used to improve the performance of distributed shared memory systems [8].

In this paper, we describe our early attempts at adding programmer-specific annotation for relaxing causal consistency in wide-area replicated storage. We describe a simple API for programmers to declare which prior writes do and do not have causal influence, as well as the implementation of that mechanism on top of Eiger's more conservative approach. We then use a simple workload to demonstrate the potential benefits and pitfalls of such relaxations for workloads that make use of it. In particular, we examine the performance of the system in terms of latency and provide upper and lower bounds for performance of the experimental setup, which will support future experiments. We go on to discuss our next steps, including plans for clients to use very fine-grained control of potential dependency tracking.

II. Background

An important design choice for a distributed system is the consistency model. Several commonly discussed types of consistency offer substantial guarantees. *Linearizibility* effectively guarantees atomicity of each operation [9]. *Sequential consistency* requires that the final state appears as though all the operations have been executed atomically in some order; this order may be different than a wall-clock ordering of the operations, but it remains acceptable, provided the final state reflects the correct ordering of each processor's operations [10].

On the other end of the consistency spectrum, *eventual consistency* is a very weak form of consistency that only requires that all updates will appear everywhere eventually [11]. One common semantic model for distributed systems is ACID: **A**tomicity, **C**onsistency, **I**solation, and **D**urability [12]. By the CAP theorem [13], [14], ACID's design choices require the sacrifice of availability in exchange for partition-tolerance (since consistency is tautologically part of ACID). In contrast to ACID, BASE is a set of alternate semantics: **B**asically **A**vailable, **S**oft state, and **E**ventually consistent [15]. BASE prioritizes availability over consistency in a partition-tolerant database. This is the model used by systems such as Cassandra [3], CouchDB [4], and Dynamo [16].

One example of the current state of the art in large-scale production databases is the Windows Azure Storage service [17]. Azure uses a model where, by default, all reads and writes are performed at one datacenter. A recent addition to the service is a model that provides read-only access to the redundant datacenters, which are only guaranteed to be eventually consistent; all writes still have to be performed at a single location. This model allows the storage system to side-step the problem of updates happening simultaneously in wide-spread locations.

Causal consistency is defined in terms of Lamport's happened-before relation [18]. This relation also implies a notion of *potential causality* and is defined by three conditions:

1) An operation performed by one process exists causally after all previous operations performed by that process.
2) An operation reading a result of another process's operation exists causally after that operation.
3) Causality is transitive. Two events are considered concurrent if there is no transitive closure between them.

Lloyd et al. developed a causally consistent database interface named COPS [19]; our system is built with the follow-on work to COPS, Eiger [5]. COPS is a layer on top of the eventually-consistent Cassandra [3] database. The COPS client tracks operations such that it can transmit these as a minimal set of dependencies with each write. The use of these dependencies satisfies rule (2), as each read implies that all future operations depend on that datum.

Cassandra's logic ensures replication both within the local datacenter and in remote datacenters. Under the COPS/Eiger system, when a new write is to be replicated to other datacenters, the origin datacenter transmits the data along with the associated dependencies. When a datacenter receives a propagated write, it first checks the attached dependency data. If the dependencies are satisfied, the write is applied to the datacenter. If not, the node that received the update holds the incoming write until its dependencies are satisfied. This ensures that each datacenter will always have a causally consistent view of the data, since writes from "the future", by definition, are absent.

```
class Client {
    write(key, value,
        add_deps=true, return_deps=false,
        custom_deps=null)
    read(key,
        add_deps=true, return_deps=false)
    ...
}
```

Figure 1. This simplified API provides add flags to indicate the desired causality behavior. The `custom_deps` parameter allows for previously acquired dependency objects to be passed into write methods to be added to the final set of dependencies associated with a particular write.

III. Declaring Causality

The model of always enforcing causal dependencies is ideal in some applications, where all activity in a session is going to be important. However, we can certainly envision applications where this is not the case. Particularly, applications that read lots of data but then only interact with some of it can be encumbered by the overhead of full causality. This can be seen in most social networks: a user may see a feed of other users' posts, but they only favorite or comment on one at a time. Were it the case that a given comment action incurred dependencies on all other dynamic content on the page, the availability of the new comment could substantially decrease. Other datacenters would then have to ensure that each and every piece of data the user could have seen was present before processing the update with this new comment. Additional risks and limits to scalability have previously been explored in more detail by Bailis et al. [20].

The CAP Theorem establishes limitations and drawbacks of demanding strict consistency [13], [14]. Partition-tolerance is nearly a requirement for wide-area distributed systems. Assuming that the system needs partition-tolerance, the CAP Theorem requires that there be a trade-off between consistency and availability. The intuitive solution is to only use systems that are Pareto-optimal [21], [22] in terms of these two properties, but few systems exist that provide more consistency than eventual but less than causal or higher.

Our solution to the problem of expensive and potentially unnecessary overhead is to provide application developers with tools to manually specify which level of consistency they want and when they want it. Using a modified API such as that in Figure 1 provides a facility for manipulating the causality-tracking subsystem of the client-side library. Developers then have the option to specify how they want to handle each read in terms of how and when dependencies get added to a running set of dependencies in the client-side library. This takes the form of a tunable "knob" with three options:

Add Dependencies: Similar to the philosophy of Isard and Birrell in the Automatic Mutual Exclusion system [23],

this system defaults to the most conservative setting. If the developer does not specify the desired level of consistency, the system will use causal+ consistency, exactly as implemented by Eiger, which is the strongest consistency level in our system. While this incurs overhead, it guarantees that correctness will not be sacrificed without the explicit consent of the developer at an access-by-access level.

Ignore Dependencies: This mode instructs the system to discard any dependencies that would be recorded during a specific access. If a write has no dependencies, whether by having read no data since the last write or by setting all accesses to “Ignore” mode, the database will behave identically to stock Cassandra: the write will be applied as soon possible everywhere.

Custom Manage: This does not immediately record the dependencies in the client’s running record. Instead, it indicates to the function doing the access that it should return any dependencies generated to the client software (via the `return_deps` option in Figure 1). This provides an opportunity for the client to implement more complex logic regarding which writes depend on which read values. Methods that consume this set of dependencies take a set of user-specified dependencies (`custom_deps` in Figure 1), which are combined with the a running record of dependencies before being propagated to remote data centers.

This ability to manage dependencies at the application level allows for accurate notation of semantic dependencies, rather than a purely all-or-nothing approach. For example, we can observe this feature’s use in many social network applications. Users have a feed of posts from friends, and each of which must be read from a data store. A user may comment in an ongoing discussion on a post in their feed. Intuitively, this specific comment should appear after all the other comments they have viewed. In a causally consistent system, this would be guaranteed; however, the user’s comment would depend on every other post, like, and advertisement that has been read from the database in order to service their requests. Of course, in an eventually consistent system, this would not be guaranteed: it would only guarantee that their comment will appear globally—eventually. With this ability to assign dependencies semantically, only the dependencies from the post and comments would be applied to the user’s new comment. Therefore, this system would be causally consistent where and when it is beneficial, but it would take advantage of the availability that comes from eventual consistency for unrelated data.

For some applications, it would be difficult to manage custom dependency logic due simply to the complexity of the application. One possible solution would be for a developer to construct a semantically meaningful abstraction of the application’s dependencies. For example, a developer of an application that is heavily oriented toward geographic location, such as Yelp, could decide to add logic to partition its dependency graphs based on region. The policy could follow from a decision that a user’s activity related to New York cannot impact data associated with Berlin. A semantic dependency management layer could enforce this policy by requiring that geographic information is supplied for every operation, at which point a “Is same region?” predicate would filter out potential dependencies that are not in the same region. Through the use of a semantically aware dependency-management facility, the complexity of low-level dependency information can be abstracted away.

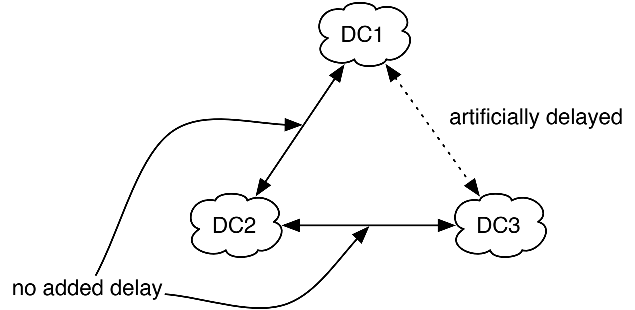

Figure 2. This three-datacenter experiment has a fully connected topology, with one of the links having symmetric delay artifically introduced.

One concern for any distributed data store involves how one handles concurrent writes to the same location. Even with a fully causally consistent data store, there is still the possibility of a conflict, e.g., two clients each write a new value to key `foo` at roughly the same time in different datacenters. Neither write has a happens-before relationship with regard to the other, so the causality logic cannot resolve this conflict. COPS, and thus our system, resolves conflicts like these with the last-writer-wins rule [24], based on the Lamport timestamp associated with each of the writes.

IV. Evaluation

In our evaluation, we use a modified version of Eiger which is itself based on Cassandra. We demonstrate the potential performance problems that can arise by applying dependency-tracking to all operations with an experiment.

The goal of this experiment is to demonstrate the direct impact of dependency tracking. In this experiment, we have three clients, one attached to each datacenter (DC) as shown in Figure 2. The link between DCs 1 and 3 has artifical delay applied. The first client starts the experiment by writing a well-known key-value pair to its local datacenter. The second client polls its DC (#2) for the key that will be written by the first client. Once it sees it, it writes a second well-known key-value pair to its local DC. If dependency tracking is enabled, this write will have a dependency on the first key-value pair, since it was read prior to the write in question. This dependency data will be attached to the data structure that conveys the update during propogation to remote DCs. Finally, the third client waits for the second well-known

Table I
Summary statistics of the experiment's response time (in ms) between DC1 and DC3 with delay and dependency-tracking varied.

Delay	No Dependencies Mean	No Dependencies Std. Dev.	All Dependencies Mean	All Dependencies Std. Dev.
50 ms	15.50	4.33	24.30	6.08
100 ms	15.75	4.18	57.75	5.34
200 ms	15.25	3.16	157.25	5.30

value to be propogated to its local DC and reports the time at which it first observed this new value. We measured the time between client 2 writing the second value and client 3 observing the second value.

We ran this experiment with varying delays applied to the link between DC1 and DC3, as shown in Figure 2. For each delay setting, we repeated the experiment 20 times. Table I shows the summary statistics for each delay setting. When we do not track dependencies, the average reponse time is almost constant across different delay settings: 15.25 ms to 15.75 ms. When we track dependnecies, the delay between DC1 and DC3 heavily affects the average delay. One counterintuitive observation is that the average delay is smaller than the artificial delay that we introduced. In order to explore this phenomenon, we adjusted all the links to have the same (large) amount of network lag. This results in the response time being slightly larger than the amount of lag, as expected, which strongly suggests that DC2, using its two fast links, eventually "helps out" with the propagation efforts.

To better understand the relationship between dependencies and response time, in Figure 3, we present the cumulative distribution of response times from the experiment presented in Table I. The trials of all no-dependency experiments have nearly identical distributions of response times, regardless of network lag. When tracking all casuality-introduced dependencies, the distributions follow similar trends as the no-dependency experiments; the all-dependency trials are right-shifted due to network delay, as expected.

This experiment demonstrates the cost of universal causal consistency: even though the two key-value pairs were unrelated, tracking all dependencies substantially increased propagation time. However, if the writes were not explicity annotated for reduced consistency, tracking irrelevant dependencies—the fallback behavior of this system—provides a fail-safe strategy.

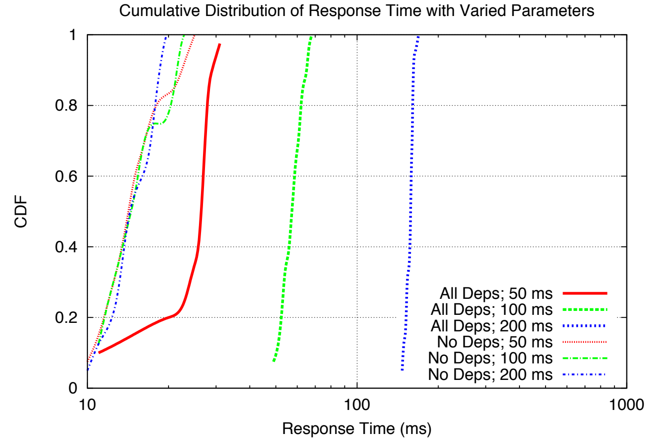

Figure 3. To better understand the relationship between dependency tracking and response time, we present a CDF of the data from the experiment described previously.

V. Related Work

Many different models of causal consistency have been proposed to improve on eventual consistency. As there have been papers that categorize them and summarize their differences [25], [26], we do not discuss the entire consistency space here. Instead, we focus on consistency management techniques for planetary-scale distributed data stores which are then used to support cloud-based systems [27]. Our system provides causal consistency as the Eiger system [5], but we remove unnecessary dependency tracking based on hints from application developers. Eiger itself was built upon the foundation set by Lloyd et al. with the COPS system [19], which provided wide-scale "causal+" consistency; however, it did so without a rich data model or write-only transactions. Eiger is an abstraction built on top of Cassandra [3], [28]. Cassandra can use either eventual or linearizible consistency; when using eventual consistency, there are options referred to as "consistency", but they actually address durability. Indeed, Cassandra does not provide a fine-grained consistency control, while our system allows applications to specify different consistency levels per operation. While Eiger is a layer on top of Cassandra, Bailis et al. created a shim layer that can provide causal consistency regardless of any semantics in the underlying store. The key insight of this shim (to achieve causal consistency without sacrificing availability) is to buffer writes and make them visible to shim clients only when a "causal cut" condition has been satisfied. This shim layer performs well with explicitly annotated consistency; however, its "metadata requirements are prohibitive for full potential causality" [29]. Conversely, our system is intended to provide causality by default, and we only weaken it upon request. A competing proposal to COPS is ChainReaction [30], which, in a departure from COPS, does not require the underlying data store to provide per-datacenter linearizability. ChainReaction uses replication chains [31] to provide causal+ consistency for a distributed key-value store with a similar interface to COPS. However, it does not support a data model more complex than key-value; Eiger provides the rich data model provided by the underlying Cassandra data store.

Previous work by Bailis et al. [20] discusses causal

consistency and the potential problems that can arise in a causally consistent distributed systems, particularly at scale. They advocate the benefits of explicitly declaring potentially causal relationships with weak consistency as the default. This work is a follow-on to that discussion. We are pursuing the evaluation of a system with such explicitly declared causality. We deviate from the recommendation to fall back to eventual consistency when there are no explicit causality relationships expressed for performance reasons; instead, we fall-back to a more conservative option: full causal consistency. This ensures causally correct behavior unless the developer explicitly opts-in to the fewer guarantees provided by eventual consistency.

Our system is not the first that allows different levels of consistency for wide-area distributed systems. Kraska et al. [32] allows developers to define consistency guarantees for files. Their prototype was implemented on top of Amazon's S3 service, which provides eventual consistency. The key difference is the unit of consistency in the two systems: their system requires a file to have a fixed consistency level, while our system allows the client to change the consistency requirement for each operation on key-value pairs. Li et al. [33] provide the option for operations to be treated either as "Blue"—eventually consistent—or "Red"—serialized with respect to all other Red operations—to enable developers to express the level of consistency that application semantics require. Our system certainly provides a mode for eventual consistency and one for stronger consistency like the RedBlue system. In the parlance of their work, the custom-management mode in our system provides for the spectrum of modes between Blue and Red.

Before the emergence of the cloud pattern of distributed systems, similar efforts on replica consistency management were motivated by mobile clients. The research community acknowledged the need to relax strong consistency to support mobile clients who temporarily become unavailable. Kordale and Ahamad [34] provided a design for supporting multiple levels of consistency, with causal consistency being the weakest. In contrast, our system provides multiple levels of consistency, with causal consistency being the strongest. Hara and Madria [35], [36] proposed multiple consistency levels for replica consistency in mobile ad-hoc networks. Through simulation, they demonstrated the effect of quorum size on read/write success with mobile clients.

VI. Conclusion

We have described our initial efforts at providing an application programming interface for distributed systems that accepts information regarding the need—or lack thereof—of causality with respect to previous operations. We have demonstrated some performance pitfalls of universal causal consistency in a distributed data store, but we have provided semantics to improve performance while still maintaining a causally consistent view for all clients.

Going forward, we want to explore the potential practical uses for causal consistency in web-scale databases. Using real-world workflow traces will provide an opportunity to discuss how this model of programming can be productively applied to production systems. We also plan to explore fine-grained dependency tracking in more depth. Certainly, if a programming feature is sufficiently burdensome for developers to use, it will remain unused. We will examine the burden that would be placed on developers to take full advantage of this fine-grained dependency tracking feature.

Wide-area distributed systems currently struggle provide the needed update performance and consistent views that are appropriate for each application—and each subsystem of each application. A system that allows for program-specific annotation greatly increases the flexibility of data stores, enabling new tradeoffs of consistency and availability.

Acknowledgments

This work was supported by IBM Research, U.S. Office of Naval Research, and U.S. National Science Foundation.

References

[1] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding Long Tails in the Cloud," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, April 2013.

[2] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks," in *Proceedings of the ACM SIGCOMM 2012 Conference (SIGCOMM '12)*, August 2012.

[3] Apache, "Cassandra," https://cassandra.apache.org/.

[4] ——, "CouchDB," http://couchdb.apache.org/.

[5] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proc. 10th USENIX NSDI*, Apr. 2013.

[6] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn, "Rethink the sync," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Oct 2006.

[7] B. D. Higgins, A. Reda, T. Alperovich, J. Flinn, T. J. Giuli, B. Noble, and D. Watson, "Intentional networking: Opportunistic exploitation of mobile network diversity," in *Proceedings of the 16th International Conf. on Mobile Computing and Networking*, 2010.

[8] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad, "Software write detection for a distributed shared memory," in *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '94, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267638.1267646

[9] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[10] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979. [Online]. Available: http://dx.doi.org/10.1109/TC.1979.1675439

[11] I. Gopal and A. Segall, "Directories for networks with casually connected users," *Computer Networks and {ISDN} Systems*, vol. 18, no. 4, pp. 255 – 262, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0169755290901074

[12] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983. [Online]. Available: http://doi.acm.org/10.1145/289.291

[13] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '00, 2000. [Online]. Available: http://doi.acm.org/10.1145/343477.343502

[14] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent available partition-tolerant web services," in *ACM SIGACT News*, 2002.

[15] D. Pritchett, "Base: An ACID alternative," *Queue*, vol. 6, no. 3, pp. 48–55, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1394127.1394128

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294281

[17] "Windows azure storage redundancy options and read access geo redundant storage," http://blogs.msdn.com/b/windowsazurestorage/archive/2013/12/04/introducing-read-access-geo-replicated-storage-ra-grs-for-windows-azure-storage.aspx, Dec. 2013.

[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[19] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043593

[20] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12, 2012. [Online]. Available: http://doi.acm.org/10.1145/2391229.2391251

[21] V. Pareto, A. Montesano, A. Zanni, L. Bruni, J. Chipman, and M. McLure, *Manual of Political Economy: A Variorum Translation and Critical Edition*. OUP Oxford, 2013. [Online]. Available: https://encrypted.google.com/books?id=q9bQmQEACAAJ

[22] V. Pareto, *Manuale di economia politica*. Societa Editrice, 1906.

[23] M. Isard and A. Birrell, "Automatic mutual exclusion," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, ser. HOTOS'07, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1361397.1361400

[24] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, Jun. 1979. [Online]. Available: http://doi.acm.org/10.1145/320071.320076

[25] P. A. Bernstein and S. Das, "Rethinking eventual consistency," in *Proceedings of the 2013 ACM International Conference on Management of Data*, ser. SIGMOD '13, 2013. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465339

[26] D. Terry, "Replicated data consistency explained through baseball," Tech. Rep. MSR-TR-2011-137, Oct 2011. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=157411

[27] P. M. Mell and T. Grance, "The NIST Definition of Cloud Computing," NIST, Special Publication 800-145, Sep 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[28] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[29] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proceedings of the 2013 ACM International Conference on Management of Data*, ser. SIGMOD '13, 2013. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465279

[30] S. Almeida, J. a. Leitão, and L. Rodrigues, "Chainreaction: A causal+ consistent datastore based on chain replication," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465361

[31] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI '04, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251261

[32] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: Pay only when it matters," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 253–264, Aug. 2009. [Online]. Available: http://dx.doi.org/10.14778/1687627.1687657

[33] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '12, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387906

[34] R. Kordale and M. Ahamad, "A scalable technique for implementing multiple consistency levels for distributed objects," *16th IEEE International Conference on Distributed Computing Systems (ICDCS '96)*, p. 369, 1996.

[35] T. Hara and S. Madria, "Consistency management among replicas in peer-to-peer mobile ad hoc networks," in *24th IEEE Symposium on Reliable Distributed Systems*, Oct 2005.

[36] T. Hara and S. K. Madria, "Consistency management strategies for data replication in mobile ad hoc networks," *IEEE Transactions on Mobile Computing*, vol. 8, 2009. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TMC.2008.150