Mitigating Contention in Distributed Systems

General Exam

Brandon Holt
University of Washington
bholt@cs.uw.edu

Abstract

The online world is a dangerous place for interactive applications. A single post by Justin Bieber can slow Instagram to a crawl; a black and blue dress going viral sends Buzzfeed scrambling to stay afloat; Star Wars movie ticket pre-sales cause service interruptions for Fandango and crash others. Developers of distributed applications must work hard to handle these high-contention situations because though they are not the average case, they affect a large fraction of users.

Techniques for mitigating contention abound, but many require programmers to make tradeoffs between performance and consistency. In order to meet performance requirements such as high availability or latency targets, applications typically use weaker consistency models, shifting the burden of reasoning about replication to programmers. Developers must balance mechanisms that reign in consistency against their effect on performance and make difficult decisions about where precision is most critical.

Abstract data types (ADTs) hide implementation details behind a clean abstract representation of state and behavior. This high-level representation is easy for programmers to build applications with and provides distributed systems with knowledge of application semantics, such as commutativity, which are necessary to apply contention-mitigating optimizations. With *inconsistent, probabilistic, and approximate (IPA) types*, we can express weaker semantics than with traditional ADTs, enabling techniques for weak consistency to be used and allowing precision to be explicitly traded for performance where applications can tolerate error. In previous work, we have demonstrated that ADT awareness can result in significant performance improvements: 3-50x speedup in transaction throughput for high-contention workloads such as an eBay-like auction service and a Twitter-like social network. IPA types are proposed for future work.

1. Introduction

Imagine a young ticket selling app, embarking on a mission to help people everywhere get a seat to watch their favorite shows. Bright-eyed and ready to face the world, it stores everything in a key/value store so that it will not forget a thing. It uses a distributed key/value store, so when it expands into new cities and reaches its millionth customer, the datastore continues to grow with it. The app uses strong consistency and transactions to ensure that no tickets are sold twice or lost, and its customers praise its reliability. “Five stars. That little app always gets my tickets, fast!” they say.

Then one day, pre-sales for the 7th Star Wars movie come out and suddenly it is inundated under a surge over 7 times the usual load, as a record-breaking number of people try to purchase tickets for this one movie. This concentrated traffic causes *hot spots* in the datastore; while most nodes are handling typical traffic, the traffic spike ends up being funneled to a handful of nodes which are responsible for this movie. These hotspots overload this app’s “scalable” datastore, causing latencies to spike, users’ connections to time out, and disappointed users

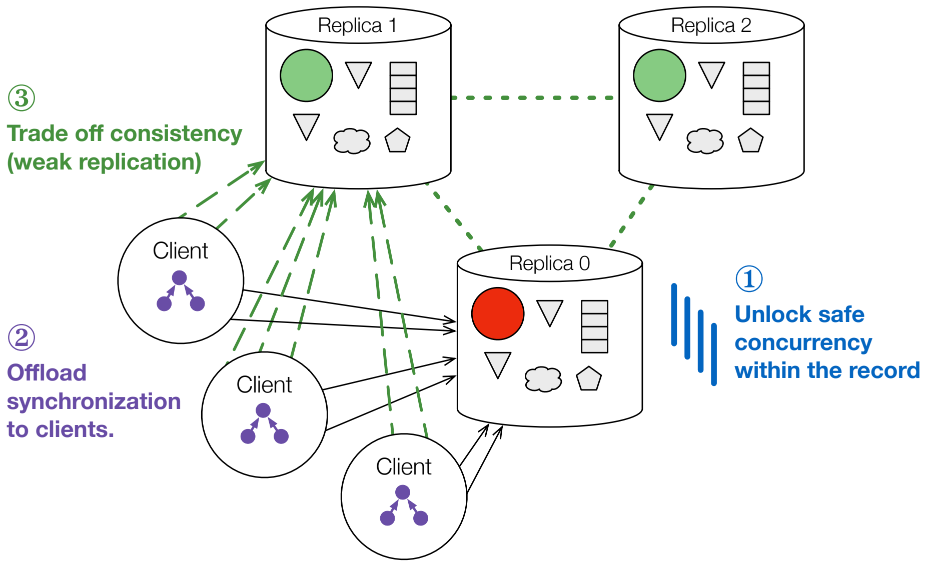
Replica 1
Replica 2
Replica 0
③ Trade off consistency (weak replication)
② Offload synchronization to clients.
① Unlock safe concurrency within the record
Client
Client
Client

Figure 1. Overview of approaches for mitigating contention, such as the hotspot in red.

who will not get to see the movie on opening night because the app was not able to sell them a ticket. On this dark night for moviegoers, even major players like Fandango, Regal, and AMC are plagued with service interruptions; some sites even crash or lose data.

This kind of concentrated surge in traffic is a perfect illustration of the contention that occurs all the time in real-world workloads, from ticket sales to social networks. Power law distributions are everywhere, from popularity of pages to number of followers, leading to network effects which can magnify even the slightest signals. Events – such as the FIFA World Cup or the Oscars – happening in realtime drive spikes in traffic. All together, this can create memes that propagate virally through social media and news sites. Interactivity is crucial to all of this — it both fuels the propagation of memes and causes contention.

Paradoxically, though contention is not the average case, it is responsible for many of the most challenging problems: tail latency, failures, and inconsistency bugs. Most of the time, an application may work correctly, with low latency and no noticeable inconsistency. However, in that 99th percentile case, contention results in a hot spot, where latencies spike, failure rates go up, and consistency can degrade, exposing bugs or resulting in user-visible inconsistencies.

1.1. Mitigating contention

Many techniques, over many years, have tackled this problem from different angles, from research on escrow and fragmenting aggregate fields in the 80s [51], to modern research on auto-generated coordination based on annotations [15, 63]. Some require a variety of changes to the programming model, while others improve underlying protocols and mechanisms. All are focused on exposing concurrency and making tasks simpler for programmers, but they can be broken down into three broad approaches, shown in Figure 1.

1. Is there any concurrency within the contended record that can be exposed? If operations on the record are commutative, they can safely run concurrently without changing the semantics; *abstract locks* (§6.2.1) leverage this to allow transactions to overlap and avoid false conflicts. *Escrow* (§4.1.3) allows the record to be treated as if it was split into a pool of fragments.

2. If clients are multithreaded, as is the case with frontend web servers that typically han-

dle many end-user requests concurrently, then some of the synchronization work can be offloaded to them. *Combining* (§6.1) leverages associativity to allow operations to be merged locally and executed as a single operation remotely, reducing the amount of work done on the overloaded datastore. Other techniques like *leases* (§4.2.1) let client-side caches avoid costly invalidation messages.

3. If clients are allowed to interact with weakly-synchronized replicas, the load on the contended shard can be reduced. However, this comes at a significant programmability cost: the illusion of a single copy of data is broken and programmers must now reason about replicated state. Weakly synchronized replicas share updates asynchronously, and clients may communicate with multiple replicas, so they can observe effects out of order, or perform updates which conflict and result in inconsistent states.

To be clear, techniques in the first and second categories can use replicas for fault tolerance and still maintain strong consistency. Strong consistency requires writes to be linearizable [37], which can be accomplished with replicas either by funneling through a single master or by a consensus protocol like Paxos [43] that makes replicas appear like a single machine. Techniques like E-Paxos [48] would fall under (1) because they expose concurrency while maintaining the single-machine view.

The third category requires much more drastic changes to programming models than the first two because it forces programmers to sacrifice consistency guarantees. However, weak consistency unlocks further improvements to performance properties like availability and lower latency that are impossible with strong consistency. In this work I will focus mostly on techniques that fall into this category because these are the most challenging to understand and require the most significant changes for programmers.

1.2. Balancing requirements

Mitigating contention is just one of the many competing requirements placed on distributed applications. They are expected to be *scalable*, *fault tolerant*, and *highly available*. Users demand responsive applications, no matter how many other users are active or where they are. The common wisdom is that companies lose money for every increase in response latency. Many systems have service-level agreements (SLAs) promising responses within a certain latency for all but the 99th percentile of requests. Throughput during peak times must be able to keep up with the load.

These performance constraints compete with the desire for *programmability* and *correctness*. Fundamentally, strong consistency cannot be provided with high availability — replication must be exposed in some way. Strict serializability and ACID transactions are among the many useful programming abstractions that must be broken to achieve those performance properties. Applications are still expected to appear mostly consistent, so developers are forced to think carefully about how to build correct systems with weaker guarantees and choose where to focus their effort.

Luckily, not everything requires the same level of precision or consistency. Some actions, such as selling the last ticket to Star Wars' opening night, require precise, consistent execution. Other situations, such as viewing the number of retweets for a popular tweet, do not need to be exactly correct — users may be satisfied as long as the number is the right order of magnitude. These *hard* and *soft constraints* can be difficult or impossible to express in current systems. Furthermore, whatever tradeoffs are made to improve programmability or enforce constraints must be balanced against meeting the performance targets, though it is not easy to quantify how changes will affect them. Programmers of these distributed applications must constantly juggle competing correctness and performance constraints.

1.3. Overview

To understand how the various techniques for managing consistency and performance relate to one another, we will explore them in terms of the properties they trade off:

- *Ordering* and *visibility* constraints between operations
- *Uncertainty* about the state in terms of staleness and possible values

We will also discuss how each technique operates, in terms of:

- *Granularity:* Does it affect the whole system, specific records, or specific operations?
- *Knowledge vs control:* Are users granted additional information about performance or consistency or are they given explicit control?

After exploring the space of existing techniques, we will propose a programming model that incorporates these disparate solutions into a single abstraction – using *abstract data types* (ADTs) to concisely describe application semantics and hide the details of the underlying consistency and coordination techniques. Our implementations, in a distributed data analytics system, *Grappa*, and a prototype ADT-store, *Claret*, show significant performance improvements, especially for high contention workloads. In future work, we propose using *inconsistent, probabilistic, and approximate (IPA) types* to trade off precision in order to take advantage of weak consistency and replication for high availability and scalability.

2. Trading off consistency for performance

In order to meet scalability, availability, and latency requirements, distributed systems programmers must routinely make tradeoffs between consistency and performance [11, 21, 32]. In the *typical* case, consistency does not pose a problem, even for geo-replicated systems – most requests are reads, requests for the same user typically go to the same server, and inconsistency windows after updates are usually small [11, 47]. Consistency becomes problematic in exceptional *high contention* cases and the *long tail* of disproportionately slow requests [28]. However, these cases are almost pathologically bad – many users together create conflicts leading to inconsistency, but this also means that many users are there to observe the inconsistency, so it is more likely to be noticed. The events most likely to cause problems are also the most newsworthy.

Targeting performance for the average case can lead to catastrophic failures in the contentious cases, but the handling all possible cases damages programmability. This burden will become abundantly clear over the next few sections as we look at the axes which programmers must traverse when making implementation choices.

3. Ordering and visibility constraints

Understanding the behavior of a sequence of operations in a program requires knowing all of the ordering and visibility constraints that govern what each operation returns and when each update actually runs. How these constraints are set is determined by the programming model.

3.1. Consistency models

Analogous to memory models in the field of computer architecture, consistency models refer to the allowable reorderings of operations and their visibility in a distributed system. Consistency models differ from memory models primarily in one way: exposing the existence of replication. Due to the reliability and speed of CPU cache hierarchies, memory models can afford to assume *coherence* will ensure that there appears to be only one copy of memory. In distributed systems, where failure is a possibility and synchronization is expensive, it is often necessary to expose

[43] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32, 2001.

[44] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

[45] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2.

[46] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 503–517, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/liu_jed.

[47] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP, pages 295–310, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi:10.1145/2815400.2815426.

[48] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP. Association for Computing Machinery (ACM), 2013. doi:10.1145/2517349.2517350.

[49] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Brigg, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, July 2015. URL http://sampa.cs.washington.edu/papers/grappa-usenix-2015.pdf.

[50] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[51] Patrick E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11 (4): 405–430, December 1986. doi:10.1145/7239.7265.

[52] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st international conference on Mobile systems, applications and services - MobiSys 03*, MobiSys. Association for Computing Machinery (ACM), 2003. doi:10.1145/1066116.1189038.

[53] Dan Pritchett. Base: An acid alternative. *Queue*, 6 (3): 48–55, May 2008. ISSN 1542-7730. doi:10.1145/1394127.1394128.

[54] Andreas Reuter. *Concurrency on high-traffic data elements*. ACM, New York, New York, USA, March 1982.

[55] Salvatore Sanfilippo. Redis. http://redis.io/, 2015a.

[56] Salvatore Sanfilippo. Design and implementation of a simple Twitter clone using PHP and the Redis key-value store. http://redis.io/topics/twitter-clone, 2015b.

[57] Peter Schuller. Manhattan, our real-time, multi-tenant distributed database for twitter scale. https://blog.twitter.com/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale, April 2014.

[58] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2 (3): 223–250, August 1984. doi:10.1145/989.1188.

[59] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS, pages 386–400, 2011. ISBN 978-3-642-24549-7.

[60] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: algorithms and performance studies. *ACM Transactions on Database Systems*, 20 (3): 325–363, September 1995. doi:10.1145/211414.211427.

[61] Nir Shavit and Asaph Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60 (11): 1355–1387, 2000.

[62] Liuba Shrira, Hong Tian, and Doug Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Middleware 2008*, Middleware, pages 42–61. Springer Science $$ Business Media, 2008. doi:10.1007/978-3-540-89856-6_3.

[63] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, PLDI. Association for Computing Machinery (ACM), 2015. doi:10.1145/2737924.2737981.

[64] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles - SOSP'11*, SOSP. Association for Computing Machinery (ACM), 2011. doi:10.1145/2043556.2043592.

[65] Michael Stonebraker. Inclusion of new types in relational data base systems. In *Proceedings of the Second International Conference on Data Engineering, February 5-7, 1986, Los Angeles, California, USA*, pages 262–269, 1986.

[66] Michael Stonebraker, W. Bradley Rubenstein, and Antonin Guttman. Application of abstract data types and abstract indices to CAD data bases. In *Engineering Design Applications*, pages 107–113, 1983.

[67] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM Symposium on Operating Systems Principles - SOSP'95*, SOSP. Association for Computing Machinery (ACM), 1995. doi:10.1145/224056.224070.

[68] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, PDIS. Institute of Electrical & Electronics Engineers (IEEE), 1994. doi:10.1109/pdis.1994.331722.

[69] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP 13*. ACM Press, 2013. doi:10.1145/2517349.2522731.

[70] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52 (1): 40, January 2009. doi:10.1145/1435417.1435432.

[71] G.D. Walborn and P.K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings. 14th Symposium on Reliable Distributed Systems*. Institute of Electrical
& Electronics Engineers (IEEE), 1995. doi:10.1109/reldis.1995.518721.

[72] W. E. Weihl. Commutativity-based Concurrency Control for Abstract Data Types. In *International Conference on System Sciences*, pages 205–214, 1988. ISBN 0-8186-0842-0.

[73] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/xie.

[74] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-Performance ACID via Modular Concurrency Control. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP, pages 276–291, 2015. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522729.

[75] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 100 (4): 388–395, 1987.

[76] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP, pages 276–291, 2013. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522729.