

Weak Consistency as a Last Resort

Marco Serafini* and Flavio Junqueira
Yahoo! Research Barcelona

1 Introduction

Production systems often use replication to guarantee that services operate correctly and are available despite faults. When designing such replicated systems, there is a fundamental trade-off between consistency of service state and availability. The CAP Theorem captures this trade-off (Consistency, Availability, and Partition-Tolerance: pick two [4]). Strong consistency guarantees simplify the task of developing applications for such systems, but have stronger requirements on the connectivity of replicas for progress. Weakly consistent replication provides higher availability, but is harder to program with.

A strong consistency guarantee often used as a correctness property is *Linearizability* [5]. Linearizability ensures that all clients observe changes of service state according to a precedence order of operations and that operations are serializable. At a high level, it provides clients with the view of a single, robust logical server. The simplicity of this abstraction explains its popularity in modern replication libraries [1, 6]. However, the high latency and low availability entailed in providing Linearizability motivates the use of weaker consistency semantics. Weakly consistent replication can terminate in worst-case runs and has lower latency. An established weak semantic is *Eventual Consistency*: if no new operation is invoked, all replicas converge to the same state [?, 11]. Whenever concurrent operations are present, however, replicas can transition to an inconsistent state and thus violate Linearizability. This is common to weakly consistent replication algorithms [?].

Current consistency semantics ensure Linearizability either *always* or *never*. In this position paper, we argue that there are alternatives when designing repli-

cated systems. We claim that some applications can benefit from having Linearizability most of the time, if it is acceptable for them to degrade consistency in favor of progress when this is the last resort. A replication algorithm can obtain this behavior by implementing *Eventual Linearizability*: it guarantees that Linearizability is only violated for finite periods of time [9].

We give insights on the rationale behind Eventual Linearizability and argue that applications implementing a master-worker approach are among potential candidates for implementing Eventual Linearizability. In such systems, the master splits the workload into shards¹ and assigns shards to workers, making sure that these operate correctly. Such applications can often tolerate duplication of work (same task performed multiple times) and might not require or even admit roll backs, but it is undesirable for them to have duplicate work at high rates.

In such applications, the master is a single point of failure and is often replicated for availability. If the master replication algorithm ensures Linearizability, it implements the abstraction of a single fault-tolerant master that never assigns the same task to multiple workers. This choice can lead to idle workers, for example if the master becomes unavailable due to partitions, because implementing arbitrary wait-free linearizable objects for synchronization requires solving consensus [?]. On the other hand, if Linearizability is relaxed then multiple masters may be present and this can lead to work performed redundantly. Eventually consistent replication is always available [10], but it can violate Linearizability arbitrarily often and thus produce an unbounded amount of duplicate work. Eventually linearizable replication is always available too, and it additionally ensures that in most of the time

*The author is currently also with the CS Dept. of TU Darmstadt, Germany.

¹A shard is a partition of work.

no duplicate work is executed because Linearizability is satisfied [9]. The risk of executing duplicate work is only taken when consensus would be blocking progress anyway.

In the remainder of this paper, we discuss the benefits and drawbacks of implementing weaker forms of consistency, many of them already known, and present a preliminary analysis comparing Linearizability, Eventual Linearizability, and Eventual Consistency. Focusing on master-worker applications, we show that Eventual Linearizability is able to provide a lower time overhead compared to other two approaches for different rates of duplicate work. Note that these results presented here constitute some evidence of the benefits of Eventual Linearizability, but a more thorough evaluation is certainly necessary to conclude that it is practical and beneficial for deployable systems.

2 Essential trade-offs

Eventual Linearizability raises three main questions. If an application benefits from Linearizability, why should it accept weaker consistency semantics? What are the costs of degrading consistency? When does degradation occur?

Why is weakening needed? The CAP theorem establishes the advantage of relaxing consistency with respect to availability and partition tolerance, two concepts that are in fact closely related. Describing availability as the ability of reaching *eventual* progress does not describe the whole story about the advantages of using weakly consistent systems. Practical systems might have a variety of latency requirements depending on the application. These requirements are often expressed as a Service Level Agreement (SLA) specifying an upper bound on a given latency percentile [3]. We can thus see the availability advantages of weakly consistent algorithms along two axes.

- **Fault tolerance:** The system is able to make progress eventually in the presence of failures that would not be tolerated by consensus;
- **Latency:** The system is able to satisfy strict latency requirements with higher probability.

These two dimensions of availability are both critical because they refer to two different domains: the set of possible failure modes and response time.

Implementing Linearizability entails solving consensus, which requires the ability of electing a unique correct leader and the presence of a majority of correct replicas [2]. From a fault-tolerance viewpoint, given that the first condition can be implemented with high availability even over a wide-area network [7], the main reason to relax consistency is the need to tolerate the failure of a majority of correct replicas.

Consider, for example, an application replicated over two data centers for disaster tolerance (a configuration that, in our experience, is far from being uncommon). Linearizable master replication is not tolerant to a data center failure. In fact, implementing Linearizability implies solving consensus, which in turn requires the presence of a majority of correct replicas. Additional redundancy can be extremely expensive and only affordable by a minority of market players, especially as the purpose is of tolerating rare failure events. Eventual Linearizability and Eventual Consistency enable higher availability because their implementations do not require a majority of correct replicas (see the Aurora [9] and Bayou [10] algorithms, respectively).

If the cost of additional replicas is sufficiently cheap, then guaranteeing the presence of a majority of correct replicas is feasible. The reason why weakly-consistent replication is so popular is related to the need of reducing latency (e.g. [3]). In fact, algorithms implementing weaker consistency guarantees also present lower latency, ensuring higher availability in the presence of strict latency requirements.

Therefore, the reason to relax consistency can be the need for availability expressed as:

- Tolerating the failure of a majority of replicas;
- Fulfilling strict latency requirements.

What are the potential drawbacks? Relaxing consistency might also have negative consequences that are hard to assess in general. There are two main drawbacks. The first is *divergence*; that is, replicas can execute operations following different orders. Divergence has two important consequences:

- Clients can observe inconsistent transitions of state, making the complexity of client software higher and prone to software faults;
- It makes it difficult, if not impossible, to handle operations that have external side-effects.

	Linearizability (Paxos)	Eventual Consistency (Bayou)	Eventual Linearizability (Aurora)
u	p	0	0
r	0	d	$p \cdot d$
T	$t_0/(1-p)$	$t_0/(1-d)$	$t_0/(1-p \cdot d)$

Table 1: Time to workload completion for different consistency semantics (Eqn. 1)

The second drawback implies a need for *reconciliation*. Replicas need to find ways to merge inconsistent local histories into a unique history to guarantee eventual convergence. This is easier if, for example, the application is a read/write storage where inconsistent states can just be overwritten with any current state. The commutativity of operations can also simplify reconciliation, as in the example of a shopping cart list [3]. Note that the impact of these two drawbacks is highly application-dependent and is hard to generalize. In the next section, we discuss them for master-worker applications.

When is degradation the last resort? Eventual Linearizability can be implemented relaxing Linearizability only when consensus can not be solved. The presence of a single correct leader followed by all replicas is sufficient for Aurora to linearize operations. If there is no majority of correct replicas or no majority can pairwise communicate, Aurora can ensure Linearizability while consensus can not be solved. Otherwise, Linearizability is provided under the same conditions needed by Paxos for progress.

3 Choosing the right consistency guarantee in master-worker applications

We previously argued that master-worker applications can benefit from Eventual Linearizability. In this section, we make a simple evaluation to show when this is or is not the case, and compare Eventual Linearizability with other consistency semantics.

We consider the problem of a set of workers trying to execute a workload, and evaluate the time it takes to complete this task using different consistency models. We highlight the main tradeoffs between the consistency degree, the likelihood of clients *dropping work* because the master is unavailable, and the likelihood of executing *duplicate work* due to divergences among

the different master replicas. We use two parameters, u and r , to express these two aspects. Different master replication algorithms result in different values of u and r (see Table 1).

Model We consider a system consisting of a set of n processes communicating over an asynchronous and unreliable network. For simplicity, each process is a client, a master replica and a worker. Among others, this models also scenarios where clients, master replicas and workers are spread over multiple sites and where intra-site communication is timely and reliable.

We evaluate the time it takes for the system to complete w units of workload. Each worker is capable of executing c workload units per time unit. In absence of master unavailability or divergences, the workload is executed in the *fault-free execution time* $t_0 = w/(c \cdot n)$.

Runs are as follows: clients continuously submit units of works by querying the replicated master to identify the responsible workers. If the master is not available or too slow, the client drops its unit of work. Otherwise, the work is executed by some worker.

We rule out trivial comparisons and only consider runs where all semantics can be implemented. For Paxos and Aurora, we assume the existence of a leader oracle which outputs the same correct leader to all processes infinitely often. For Aurora, we assume that the leader election algorithm always suspects a slow leader before this results in a client dropping its submitted work. For Paxos, we assume that a majority of correct processes exists.

For Paxos and Aurora, we denote with $p \in [0, 1)$ the fraction of time where the client can not observe a linearizable history in a timely manner. This can happen for two reasons. First, the leader election algorithm may not be currently electing a single correct leader at all correct processes due to, for example, network connectivity issues. Second, the latency of the replication algorithm violates the requirements of the clients. This can also happen if the client is isolated by a partition.

As discussed in Section 2, for a given client timeout, algorithms with lower latency have higher availability. Unlike Aurora and Bayou, Paxos needs to contact a majority of replicas to implement consensus so its value of p is typically higher. It is also important to remark that the value of p is proportional to the length of the client timeout. Shorter timeouts result in lower values of p .

For all protocols, we denote u as the fraction of time a replica remains idle because a client drops units of work due to unavailability of the replicated master. Bayou and Aurora have $u = 0$ since each process locally responds to client requests before the client drops its work. For Aurora, this holds even if a leader is not available or untimely. Paxos has $u = p$ instead.

As for duplicate work, this never occurs if the replication algorithm guarantees Linearizability. However, a worker can execute duplicate work in periods where it does not observe a linearizable history. This occurs for a fraction p of time. We say that the replica is *divergent* in these periods. The time during which a process diverges is called *divergence time*. We assume that each replica spends a fraction $d \in [0, 1)$ of its divergent time performing duplicate work.

We call r the overall fraction of time, both divergent and not, when a replica executes duplicate work.² In Bayou, a replica is always divergent so $r = d$. In Aurora, a replica may only execute duplicate work when consistency degrades so $r = p \cdot d$. Paxos does not allow replicas to diverge, so $r = 0$.

Preliminary evaluation We now calculate the time it takes for the system to process w units of workload. In ideal runs, this time is t_0 . During this time window, a process can drop work or execute duplicate work for a time $(u+r)t_0$. The amount of work units that are not utilized to complete the workload is thus $(u+r)t_0cn$. These units of work need to be completed after t_0 . The additional time it takes to complete them, assuming that there are neither partitions nor divergences after t_0 , is $t_1 = (u+r)t_0$.

In general, the additional time needed after a time t_i to complete work lost during t_i , assuming that no further partitions or divergences occur, is $t_{i+1} = (u+r)t_i = (u+r)^i t_0$. The overall time T needed to complete the workload in presence of divergence and partitions is the sum of all t_i with $i \in [0, \infty)$ is:

$$T = t_0 \cdot \sum_{i=0}^{\infty} (u+r)^i = \frac{t_0}{1-(u+r)} \quad (1)$$

The values of T for replication algorithms implementing Linearizability, Eventual Consistency and Eventual Linearizability are illustrated in Table 1. Independen-

²Note that $u+r < 1$ because, by definition, some replica needs to do some work before doing any duplicate work.

dent of the value of p and d , Aurora enables shorter completion times than other algorithms. The significance of this gain can be determined by the time overhead that different semantics impose. The additional execution time compared to a fault-free execution time t_0 , normalized over t_0 is:

$$O = \frac{T - t_0}{t_0} = \frac{1}{1-(u+r)} - 1 \quad (2)$$

Figure 1(a) shows time overhead if $1-p = 0.9$. In this case, the use of Linearizability results in more than 10% performance degradation. Eventual Consistency is very sensible to the likelihood of duplicate work. If $d > 0.1$, there is little benefit in using it compared to Linearizability, unless the extra cost of implementing on top of Eventual Linearizability is negligible.

Aurora outperforms Paxos by at least one order of magnitude if $d \leq 0.1$. Even if $d = 0.5$, which is a high value, Paxos has still half the overhead of Aurora. In general, as d grows compared to p , the relative advantage of Aurora over Paxos can be approximated by d .

If p decreases, as shown in Figure 1(b), the absolute difference between Paxos and Aurora decreases too, but the same relative difference remains the same. Bayou is not sensible to changes of p so it keeps the same performance, which is comparatively worse now.

Paxos is more sensible than Aurora to variation of p , as shown in Figure 1(c). If p is high, Paxos does not achieve high performance and the difference with Aurora is quite high. However, as p decreases, the absolute difference between these two semantics decreases. Bayou is not sensible to availability variations.

How to choose Paxos and Aurora perform similarly if p is low and thus Linearizability can be frequently achieved in a timely manner. If the application does not have stringent latency requirements and if a majority of correct replicas always exists, then Linearizability is the best choice. It simplifies the system design and prevents external side effects.

If the application has strict latency requirements and must operate over an unreliable network, such as the Internet, then p tends to be higher. Multiple studies observe that Internet latencies tend to be quite higher than the norm for a quite relevant fraction of measurements (e.g [12]). The choice is thus between setting very high timeouts, to reduce p , and setting more aggressive timeouts and handle a higher p . Eventual Lin-

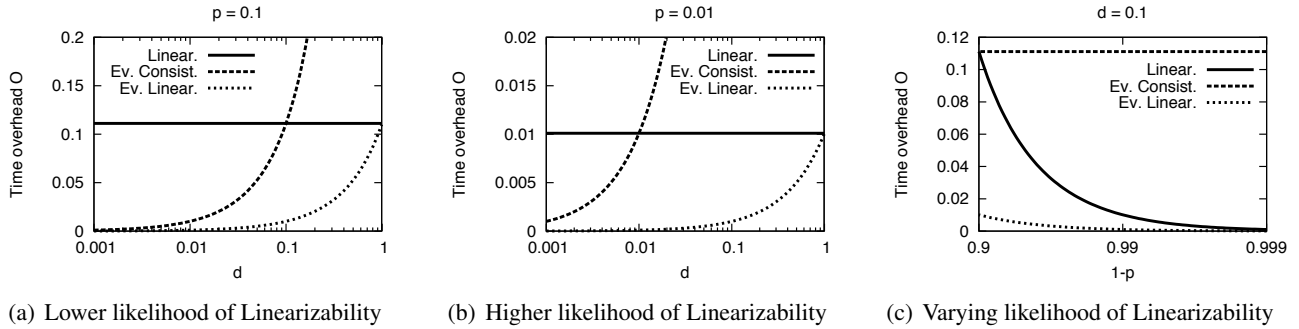


Figure 1: Time overhead O with varying likelihood of duplicate work and availability (Eqn. 2)

earizability makes the second choice very attractive. Aurora achieves a time to workload completion even with a high p that is generally close to the fault-free execution time. In applications that have strict latency requirements and where duplicate work has little or no external side effects, Aurora makes a sharding over multiple data centers practically as efficient as sharding within a single data center. This results in a significant leap toward higher flexibility in the utilization of resources spread different geographical locations.

4 Discussion

Our preliminary results show that there is potentially important benefit in considering weaker models than Linearizability for distributed master-worker applications. The particular model we focused on here is Eventual Linearizability, which guarantees Linearizability in periods of stability. Compared to Linearizability, it guarantees progress despite the absence of a single leader and in the presence of network partitions, which occur in both wide-area systems and data centers [11]. Compared to Eventual Consistency, Eventual Linearizability enables a significant reduction in the amount of duplicate work. As production systems are stable most of the time, a system implementing Eventual Linearizability will satisfy Linearizability most of the time.

Weak consistency enables duplicate work. Duplicate work comes both in the form of duplicate work that two or more processes perform and of a process having to reconcile inconsistent views of the system states. There are different options to reconcile a divergent state [?]. Among these, rollback and re-executions are a very generic options [10]. Both

Eventual Linearizability and Eventual Consistency require that designers include reconciliation mechanisms. However, Eventual Linearizability exercises them much less compared to Eventual Consistency, which leads to performance advantages for systems implementing Eventual Linearizability.

An open question is the real impact of such weaker forms of consistency on real applications. For instance, for a Web crawler of a search engine, we are yet to understand what consequences such a duplication of work has given the business model used by commercial Web search engines. For a crawler, duplication of work may lead to higher bandwidth utilization and violation of Web site policies. In general, the impact of weaker forms of consistency is poorly understood in the context of distributed coordination for Web-scale systems, such as in the case of master-worker systems. We believe, however, that there is an important class of non-mission-critical applications that can highly benefit from such techniques as we have illustrated in this work.

Acknowledgment

This work has been partially supported by the FP7 COAST (FP7-ICT-248036) project, funded by the European Community.

References

- [1] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

- [2] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
- [3] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.
- [4] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!’s hosted data serving platform. In *Very Large Data Bases Conference*, 2008.
- [5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. In *ACM Symposium on Operating Systems Principles*, pages 205–220, 2007.
- [6] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. In *PODC ’96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 300–309, New York, NY, USA, 1996. ACM.
- [7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM Symposium on Operating Systems Principles*, 2003.
- [8] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.
- [9] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Ben Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Annual Technical Conference*, 2010.
- [10] Idit Keidar and Alexander Shraer. How to choose a timing model. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1367–1380, 2008.
- [11] Marco Serafini, Dan Dobre, Matthias Majuntke, Peter Bokor, and Neeraj Suri. Eventually linearizable shared objects. In *Proceedings of the 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2010.
- [12] 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 *SOSP ’95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM.
- [13] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.
- [14] Bo Zhang, T. S. Eugene Ng, Animesh Nandi, Rudolf Riedi, Peter Druschel, and Guohui Wang. Measurement based analysis, modeling, and synthesis of the internet delay space. In *IMC ’06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 85–98, New York, NY, USA, 2006. ACM.