Weak Consistency as a Last Resort

Marco Serafini and Flavio Junqueira
Yahoo! Research
Barcelona, Spain
{ serafini, fpj }@yahoo-inc.com

ABSTRACT

It is well-known that using a replicated service requires a tradeoff between availability and consistency. Eventual Linearizability represents a midpoint in the design spectrum, since it can be implemented ensuring availability in worst-case periods and providing strong consistency in the regular case. In this paper we show how Eventual Linearizability can be used to support master-worker schemes such as workload sharding in Web applications. We focus on a scenario where sharding maps the workload to a pool of servers spread over an unreliable wide-area network. In this context, Linearizability is desirable, but it may prevent achieving sufficient availability and performance. We show that Eventual Linearizability can significantly reduce the time to completion of the workload in some settings.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliablity—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms

Algorithms, Design, Reliability

Keywords

eventual linearizability, master-worker schemes, availability

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 principle captures this trade-off (Consistency, Availability, and Partition-Tolerance: pick two [7]). 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.

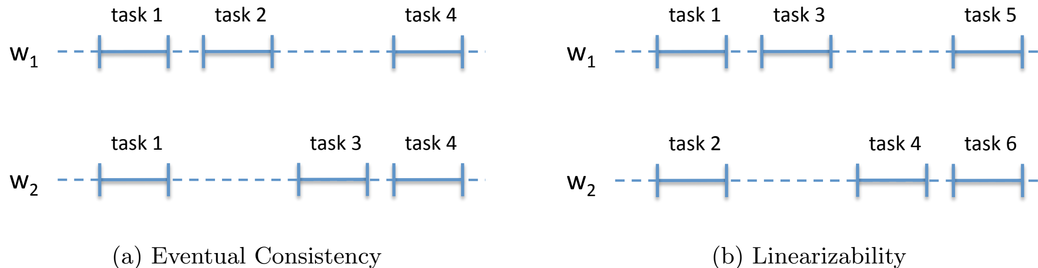
A strong consistency guarantee that is often used as a correctness property is *Linearizability* [9]. Linearizability ensures that all clients observe changes of service state according to the real-time precedence order of operations and that operations are serialized. At a high level, it provides clients with the view of a single, robust logical server. The simplicity of this abstraction explains its popularity in modern replication libraries [1, 10]. In some scenarios, however, the high latency and low availability entailed in providing Linearizability motivates the use of weaker consistency semantics. Weakly consistent replication can ensure termination of each operation in worst-case runs and has lower latency. An established weak consistency guarantee is *Eventual Consistency*: if no new operation is invoked, all replicas eventually converge to the same state [14, 18]. Whenever concurrent operations are present, however, replicas can diverge and transition to an inconsistent state that violates Linearizability. This can occur infinitely often, even in periods when both availability and Linearizability could be guaranteed. Eventual Consistency requires these inconsistencies to be reconciled only eventually, after they have been observed by clients. Other similar consistency properties that admit observable divergences at any point in time and only require eventual reconciliation, such as Eventual Serializability [6], have been proposed.

Current consistency semantics ensure Linearizability either *always* or *never*. In this position paper, we argue that there are alternatives when designing replicated 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. Replication algorithms implementing *Eventual Linearizability*, like Aurora, can obtain this behavior [15]. Aurora guarantees that Linearizability is only relaxed when necessary to preserve availability.

We give insight on Eventual Linearizability and argue that applications using master-worker approaches are among potential use cases for eventually-linearizable replication. In such applications, the master splits the workload into tasks and assigns tasks to workers, making sure that these operate correctly. These applications can often tolerate duplication of work (same task performed multiple times) and might not require reconciliation of inconsistencies, 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. Figure 1 illus-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
LADIS '10 Zürich, Switzerland
Copyright 2010 ACM 978-1-4503-0406-1 ...$10.00.



(a) Eventual Consistency

(b) Linearizability

Figure 1: Admissible runs of a master-worker application with a replicated master and different consistency semantics. Workers w_1 and w_2 act as clients and concurrently query the master to fetch the next task they must execute, which is reported above each operations. Figure 1(a): If the master is replicated in an eventually-consistent manner, concurrent operations might repeatedly lead to duplicate work even if reconciliation takes place immediately after every operation. Figure 1(b): A linearizable implementation provides the abstraction of a single master and ensures that no duplicate work is ever executed. Eventual Linearizability alternates linearizable periods (in the regular case) and eventually-consistent periods (when relaxing consistency is necessary to preserve availability).

trates two admissible runs of a master-worker scheme with a replicated master and different consistency semantics. 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. Implementing arbitrary wait-free linearizable objects, like the master, requires solving consensus [8]. This choice can lead to idle workers when the master becomes unavailable because consensus cannot be solved, for example due to partitions. 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 when at least one replica is not faulty [17], 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 no duplicate work is executed because Linearizability is satisfied [15]. The risk of executing duplicate work is only taken when consensus would be blocking progress.

In the remainder of this paper, we first discuss benefits and drawbacks of relaxing Linearizability, also reviewing many known concepts. Next, we present a preliminary analysis comparing Linearizability, Eventual Linearizability, and Eventual Consistency. Focusing on master-worker applications, we show that eventually-linearizable replication algorithms can complete workloads in a shorter time than algorithms implementing the other two properties. The results presented here aim at highlighting the high-level design tradeoffs and only constitute some evidence of the benefits of Eventual Linearizability. A more thorough evaluation is certainly necessary to assess how practical and beneficial Eventual Linearizability is 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 should degradation occur?

Reasons for weakening Linearizability.

The CAP principle establishes that one major advantage of relaxing consistency is to achieve availability and partition tolerance. Availability is typically described as the ability of reaching *eventual* progress, but this does not tell the whole story about the advantages of using weakly consistent systems. The availability advantages of weakly consistent algorithms can in fact be seen along two distinct 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.

We first discuss the fault-tolerance dimension of availability. Implementing arbitrary wait-free linearizable objects entails solving consensus. Consensus, in turn, requires the ability of electing a unique correct leader that can communicate with a majority of replicas [2, 3].

Electing a unique leader can be implemented with high probability even over a wide-area network [12]. From a fault-tolerance viewpoint, one important reason to relax consistency is to tolerate situations in which no unique leader can communicate with a majority of replicas due to either crashes or network partitions. Consider, for example, an application replicated over two data centers for disaster tolerance (a configuration that, in our experience, is not uncommon). Linearizable master replication cannot tolerate the failure of a data center because solving consensus 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 when tolerating rare failure events is necessary.

Even if a majority of correct replicas connected through timely links to a correct leader exists, some replicas may be partitioned off. Linearizable implementations do not enable such disconnected replicas to achieve progress. This is

problematic for replicated applications mandating that *all* replicas must be able to reply to clients.

In terms of fault-tolerance, Eventual Linearizability and Eventual Consistency enable higher availability because their implementation does not require a majority of correct replicas. Furthermore, they allow replicas that are disconnected by a partition to execute requests locally (see the Aurora [15] and Bayou [17] algorithms, respectively).

Latency is the second dimension of availability. Practical replication algorithm might have a variety of latency requirements, depending on the replicated application. These requirements are often expressed as a Service Level Agreement (SLA) specifying an upper bound on a given latency percentile [5].

In systems with strict latency requirements, it may be necessary to relax consistency even if one could assume the absence of faults or partitions. In fact, algorithms implementing weaker consistency guarantees also present lower latency, ensuring higher availability in the presence of strict latency requirements. This is easy to see if we consider three algorithms for implementing arbitrary shared objects: Paxos [13], Bayou [17] and Aurora [15]. Bayou, which implements the weakest consistency semantic, has the lowest latency, whereas Paxos has the highest latency.

The Paxos protocol is a consensus algorithm used to implement linearizable objects. Paxos specifies that requests must be ordered by a leader replica, which then must contact a majority of replicas before requests complete (Figure 2(a)). If a unique leader replica is not available or if it is available but it cannot contact a majority of replicas, the protocol blocks. Bayou is a well-known algorithm for implementing eventually-consistent objects. With Bayou, replicas receiving client requests reply without first communicating with other replicas. Replicas then forward requests in the background to all other replicas for reconciliation. Many other practical weakly consistent algorithms have a similar communication pattern and offer similar properties [14]. Aurora can implement eventually-linearizable objects by using weak operations. It optimistically tries to get requests ordered by a leader (Figure 2(b)). If no leader is available, then any replica that is accessible by the client can reply to requests to ensure availability.

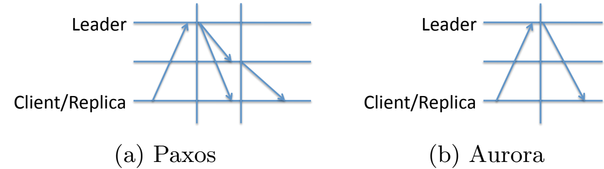
Therefore, the need to relax consistency can arise for one (or more) of these reasons:

- Tolerating the absence of a unique leader which can communicate with a majority of replicas;
- Enabling replicas that are partitioned off to execute operations locally;
- Fulfilling strict latency requirements.

Potential drawbacks.

Relaxing Linearizability 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. Handling them can make client software more complex and prone to software faults;



(a) Paxos (b) Aurora

Figure 2: Communication pattern for Paxos and Aurora in good runs. We assume for simplicity that clients and replicas are co-located, for example in the same site of a wide-area system. For Aurora, we depict the case where the leader is available. If the leader does not respond within a given time, the local replica directly replies to the client.

- It makes it difficult, if not impossible, to handle operations that have adverse external side-effects.

Unlike Paxos, both Bayou and Aurora can cause replicas to diverge. With Bayou, replicas reply to clients independently, so concurrent operations can always generate inconsistencies. With Aurora, replicas optimistically try to order requests through a leader. During regular periods when this leader is available and unique, the system behaves as a linearizable system. Concurrent operations can generate inconsistencies only in asynchronous periods when these conditions do not hold. Background request forwarding is also done in Aurora to ensure Eventual Consistency in such asynchronous periods.

The second drawback is the need for *reconciliation* after divergences. Replicas need to find ways to merge inconsistent local histories into a unique history to guarantee eventual convergence. There are different options to reconcile divergent states, many of which are efficient but application-specific [14]. Reconciliation 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 [5]. A more generic, but expensive, approach is to roll back to a checkpoint and to re-execute all conflicting operations in a consistent order. Conflicts can be detected and merged in parallel, as done for example in Bayou.

The impact of divergences and reconciliation is highly application-dependent. In the next section, we consider the case of master-worker applications.

Degradation as a last resort.

We have discussed that Linearizability has many advantages over weaker consistency semantics. Therefore, it should only be degraded when needed to reduce latency or to ensure progress in presence of a faulty environment. Consistency degradation as a last resort is one key design principle of the Aurora algorithm.

Aurora achieves Linearizability in periods when a unique leader is available and able to respond to all replicas in a timely manner, regardless of the number of faulty replicas. If these conditions are not met, Aurora degrades to Eventual Consistency. Aurora thus ensures Linearizability under the same conditions needed by Paxos for progress, and degrades it only in periods when using Paxos would result in blocking the operations of some replica. On the other hand, Aurora

has similar availability properties as Bayou: it can tolerate asynchrony and an unbounded number of faults. Like Bayou, Aurora can also meet stringent latency requirements by letting the replica closer to the client reply to requests if the leader replica is too slow.

Some algorithms in the literature have analogies with Aurora, but they not implement Eventual Linearizability and may degrade Linearizability also in periods when this is not necessary for availability. The COReL algorithm is a total order broadcast algorithm using partitionable group membership, rather than an unreliable failure detector, as underlying building block [11]. It optimistically outputs operations before their total order is known. In runs where all correct processes are eventually in the same partition, COReL ensures that eventually the optimistic order of the operations is equal to the definitive total order. The Zeno algorithm implements Eventual Consistency in systems prone to Byzantine faults [16]. Zeno uses a leader to order operations and keep a small quorum of replicas consistent. Clients can then select a reply from a correct replica by checking that the same reply is received by the all replicas in the quorum. In some "good runs", this mechanisms also implicitly implements Eventual Linearizability.

3. CONSISTENCY IN MASTER-WORKER APPLICATIONS

We previously argued that master-worker applications can benefit from Eventual Linearizability. In this section, we present a simple analysis to show when this is or is not the case, and compare Eventual Linearizability with other consistency semantics. To make the comparison practical, we consider three algorithms implementing the considered consistency semantics: Paxos, Bayou, and Aurora.

We consider the problem of a set of workers trying to execute a workload, and analyze the time it takes to complete this task using different consistency models. We highlight the main tradeoffs between the consistency degree, the likelihood of having *unperformed 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).

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

Table 1: Comparison of algorithms implementing different consistency properties.

Model.

We consider a system comprising a set of workers and master replicas processes communicating over an asynchronous and unreliable network. The workers are the clients of the replicated master, which acts as a server. They fetch from the master the task that they execute. We assume that workers always communicate in a timely manner with one *local* master replica. This models, for example, scenarios where master replicas are spread across multiple sites on a wide-area system, and where workers are co-located with master replicas. Note that if Paxos is used, a single master replica cannot reply to workers without first contacting other replicas.

We analyze the time it takes for n workers to complete w units of workload. Each worker is capable of executing c workload units per time unit. Runs are as follows: workers continuously query the replicated master to fetch some task to execute. If the replicated master is unavailable and does not reply, the worker wastes time and some amount of work remains unperformed.

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, but not necessarily always.

For Paxos and Aurora, we denote with $p \in [0,1)$ the fraction of time when a worker either remains idle waiting for master replies satisfying Linearizability, or receives weakly consistent replies. We do not consider the case $p = 1$, where the comparison is trivial. Note that, unlike Aurora and Bayou, weakly consistent replies are never observed by clients with Paxos. Different from Aurora, Paxos also needs to contact a majority of replicas to implement consensus, and its value of p is typically higher. In the analysis, we abstract this aspect away for simplicity.

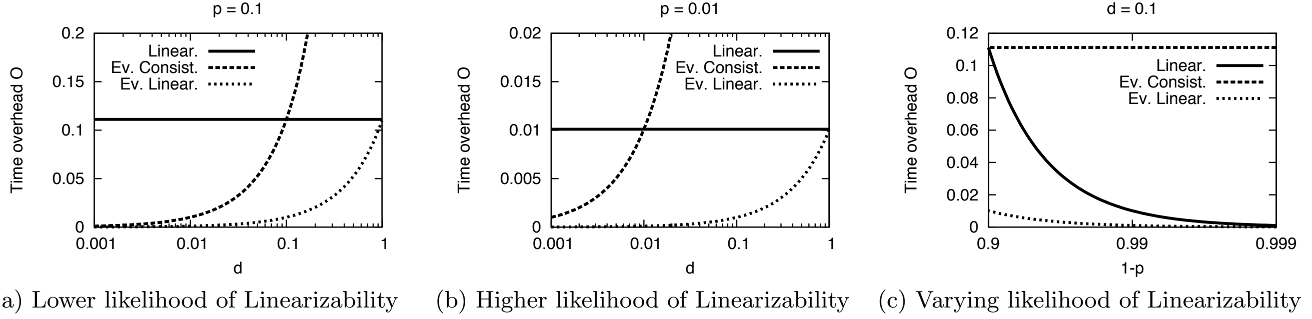
For all protocols, we use u to denote the fraction of time a worker remains idle. Bayou and Aurora have $u = 0$ because each master replica can always respond to requests of local workers in a timely manner, without causing significant unperformed work. Paxos has $u = p$ instead.

As for duplicate work, this never occurs if the replication algorithm guarantees Linearizability. However, a worker can execute duplicate work during periods in which it does not observe a linearizable history. We say that the worker is *divergent* in these periods. The time during which a worker diverges is called *divergence time*. We assume that a worker spends a fraction $d \in [0,1)$ of its divergent time performing duplicate work. Note that $d < 1$ because, by definition, some worker must perform some work before any worker can start doing duplicate work.

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

Preliminary analysis.

We now calculate the time it takes for the system to process w units of workload. In the absence of unperformed or duplicate work, the workload is executed in the fault-free execution time $t_0 = w/(c \cdot n)$. During this time window, a worker can remain idle or execute duplicate work for a time $(u + r) \cdot t_0$. The amount of work units that are not utilized to complete the workload is thus $(u+r)\cdot t_0 \cdot c \cdot n$. These units of work must be completed after t_0. The additional time it takes to complete them is $t_1 = (u + r) \cdot t_0$ if we assume that after t_0, no partition occurs and there is no divergence.



(a) Lower likelihood of Linearizability (b) Higher likelihood of Linearizability (c) Varying likelihood of Linearizability

Figure 3: Normalized time overhead O with varying likelihood of duplicate work and availability (Eqn. 2)

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

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

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

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

Figure 3(a) shows the normalized time overhead if $1-p = 0.9$. In this case, the use of Linearizability results in more than 10% performance degradation compared to fault-free runs. Eventual Consistency is very sensitive to the likelihood of duplicate work. If $d > 0.1$, there is little benefit in using it compared to Linearizability.

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 3(b), the absolute difference between Paxos and Aurora decreases too, but the same relative difference remains the same. Bayou is not sensitive to changes of p so it keeps the same performance, which is comparatively worse now.

Paxos is more sensitive than Aurora to variations of p, as shown in Figure 3(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 sensitive to availability variations.

How to choose a consistency property.

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 unique correct leader and 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. Authors of [4] extensively studied multiple Internet traces and reported failure rates as high as 7%. The authors conclude that even if commercial hosting services advertise very high availability figures, the actual end-to-end availability of the system is typically limited by the unavailability of wide-area links. Multiple studies observe that Internet latencies tend to be high often enough (e.g., [19]). Applications that need low latency must choose between setting very high timeouts, to reduce p, and setting more aggressive timeouts and handle a higher p. Eventual Linearizability makes the second choice very attractive. The time to workload completion of Aurora even with a high p is generally close to the fault-free execution time.

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 [18]. 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 normally satisfy Linearizability.

By admitting rare degradations of Linearizability, Aurora shares the drawbacks of weakly consistent algorithms, i.e., degradations and reconciliations. However, Aurora makes their impact less significant compared to eventually-consistent algorithms, like Bayou, thanks to the “degradation as a last resort” principle. Consider first the impact of divergence. With Aurora, clients still need to be prepared for divergence, but this occurs rarely so its side effects will be limited. Consider the example of a master-worker application. The amount of duplicate work, which is the side effect of divergence in this example, can be significantly reduced by using Aurora instead of Bayou. Similar simple observations can be done for other applications. For example, Aurora

could be used to implement a highly available flight booking system which limits the amount of unintended overbooking.

The second drawback of relaxing Linearizability, reconciliation, is also mitigated by the "degradation as a last resort" principle. Aurora reduces the frequency of reconciliation compared to Bayou or other eventually-consistent systems. This is a critical aspect in presence of expensive reconciliation algorithms such as, for example, the generic roll back and re-execute algorithm of Bayou. In fact, both Aurora and Bayou can require reconciliation, but Aurora exercises them much less compared to Bayou.

Our preliminary analysis shows that in applications running over wide-area systems and where a little amount of duplicate work can be accepted, Aurora has the potential to make task assignment over multiple data centers as efficient as within a single data center. This would result in a significant leap toward higher flexibility in the utilization of resources spread across different geographical locations.

An open question is the real impact of such weaker forms of consistency on real applications. Consider, for instance, the Web crawler of a search engine. Eventual Linearizability could be beneficial because it can simplify the distribution of the crawling workload over multiple data centers. However, we are yet to understand the consequences of duplication of work given the business model used by commercial Web search engines. For example, 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 still poorly understood in the context of distributed coordination for Web-scale systems. We believe, however, that there is an important class of 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.

5. REFERENCES

[1] M. 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, 2006.

[2] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[3] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[4] M. Dahlin, B. B. V. Chandra, L. Gao, and A. Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking (TON)*, 11(2):300–313, April 2003.

[5] 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 *SOSP'07: ACM Symposium on Operating Systems Principles*, pages 205–220, 2007.

[6] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 300–309, 1996.

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

[8] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programing Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[9] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC'10: Proceedings of USENIX Annual Technical Conference*, 2010.

[11] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *PODC'96: Proceedings of fifteenth ACM Symposium on Principles of Distributed Computing*, pages 68–76, 1996.

[12] I. Keidar and A. Shraer. How to choose a timing model. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 19(10):1367–1380, 2008.

[13] L. Lamport. The part-time parliament. *ACM Transactions on Computing Systems (TOCS)*, 16(2):133–169, 1998.

[14] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[15] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, and N. Suri. Eventually linearizable shared objects. In *PODC '10: Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 95–104, 2010.

[16] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 169–184, 2009.

[17] D. B. Terry, M. M. Theimer, K. 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, 1995.

[18] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[19] B. Zhang, T. S. E. Ng, A. Nandi, R. Riedi, P. Druschel, and G. 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, 2006.