Speed for the elite, consistency for the masses: differentiating eventual consistency in large-scale distributed systems

Davide Frey*, Achour Mostefaoui†, Matthieu Perrin†, Pierre-Louis Roman‡, François Taïani‡
*Inria Rennes, France
davide.frey@inria.fr
†University of Nantes – LINA, France
achour.mostefaoui@univ-nantes.fr, matthieu.perrin@univ-nantes.fr
‡University of Rennes 1 – IRISA – ESIR, France
pierre-louis.roman@irisa.fr, francois.taiani@irisa.fr

***Abstract*—Eventual consistency is a consistency model that emphasizes liveness over safety; it is often used for its ability to scale as distributed systems grow larger. Eventual consistency tends to be uniformly applied to an entire system, but we argue that there is a growing demand for differentiated eventual consistency requirements.**

We address this demand with *UPS*, a novel consistency mechanism that offers differentiated eventual consistency and delivery speed by working in pair with a two-phase epidemic broadcast protocol. We propose a closed-form analysis of our approach's delivery speed, and we evaluate our complete mechanism experimentally on a simulated network of one million nodes. To measure the consistency trade-off, we formally define a novel and scalable consistency metric that operates at runtime. In our simulations, *UPS* divides by more than 4 the inconsistencies experienced by a majority of the nodes, while reducing the average latency incurred by a small fraction of the nodes from 6 rounds down to 3 rounds.

I. Introduction

Modern distributed computer systems have reached sizes and extensions not envisaged even a decade ago: modern datacenters routinely comprise tens of thousands of machines [1], and on-line applications are typically distributed over several of these datacenters into complex geo-distributed infrastructures [2], [3]. Such enormous scales come with a host of challenges that distributed system researchers have focused on over the last four decade. One such key challenge arises from the inherent tension between fault-tolerance, performance, and consistency, elegantly captured by the CAP impossibility theorem [4]. As systems grow in size, the data they hold must be replicated for reasons of both performance (to mitigate the inherent latency of widely distributed systems) and fault-tolerance (to avoid service interruption in the presence of faults). Replicated data is unfortunately difficult to keep consistent: *strong consistency*, such as linearizability or sequential consistency, is particularly expensive to implement in large-scale systems, and cannot be simultaneously guaranteed together with availability, when using a failure-prone network [4].

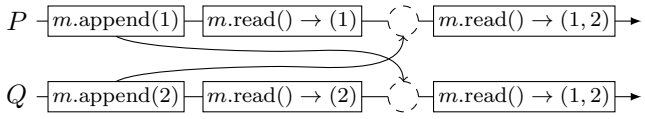


Fig. 1: An eventually consistent append-only queue.

The cost and limitations of strong consistency have prompted an increased interest in weaker consistency conditions for large-scale systems, such as PRAM or causal consistency [5], [6], [7]. Among these conditions, *eventual consistency* [8], [9] aims to strike a balance between agreement, speed, and dynamicity within a system. Intuitively, eventual consistency allows the replicas of a distributed shared object to temporarily diverge, as long as they eventually converge back to a unique global state.

Formally, this global consistent state should be reached once updates on the object stop (with additional constraints usually linking the object's final value to its sequential specification) [10]. In a practical system, a consistent state should be reached every time updates stop for *long enough* [3]. How long is long enough depends on the properties of the underlying communication service, notably on its *latency* and *ordering guarantees*. These two key properties stand in a natural trade-off, in which latency can be traded off for better (probabilistic) ordering properties [11], [12], [13]. This inherent tension builds a picture in which an eventually consistent object must find a compromise between speed (how fast are changes visible to other nodes) and consistency (to which extend do different nodes agree on the system's state).

Figure 1, for instance, shows the case of a distributed append-only queue[1] m manipulated by two processes P and Q. m supports two operations: $\mathrm{append}(x)$, which appends an integer x to the queue, and $\mathrm{read}()$, which returns the current content of m. In Figure 1, both P and Q eventually converge

[1] An append-only queue may for instance be used to implement a distributed blockchain ledger in cryptocurrency systems, or to realize a distributed log.

to the same consistent global state $(1, 2)$, that includes both modifications: $m.\text{append}(1)$ by P and $m.\text{append}(2)$ by Q, in *this* order. However, Q experiences a temporary inconsistent state when it reads (2): this read does not "see" the append operation by P which has been ordered before $m.\text{append}(2)$, and is thus inconsistent with the final state $(1, 2)$[2]. Q could increase the odds of avoiding this particular inconsistency by delaying its first read operation, which would augment its chances of receiving information regarding P's append operation on time (dashed circle). Such delays improve consistency, but reduce the speed of change propagation across replicas, and must be chosen with care.

Most existing solutions to eventual consistency resolve this tension between speed and consistency by applying one trade-off point uniformly to all the nodes in a system [3], [11]. However, as systems continue to grow in size and expand in geographic span, they become more diverse, and must cater for diverging requirements. In this paper, we argue that this heterogeneity increasingly call for differentiated consistency levels in large-scale systems. This observation has been made by other researchers, who have proposed a range of hybrid consistency conditions over the years [14], [15], [16], [17], but none of them has so far considered how eventual consistency on its own could offer differentiated levels of speed and consistency within the same system.

Designing such a protocol raises however an important methodological point: how to measure consistency. Consistency conditions are typically formally defined as predicates on execution histories; a system execution is thus either consistent, or it is not. However, practitioners using eventual consistency are often interested in the current *level* of consistency of a live system, i.e., how far the system currently is from a consistent situation. Quantitatively measuring a system's inconsistencies is unfortunately not straightforward: some practical works [3] measure the level of *agreement* between nodes, i.e., how many nodes see the same state, but this approach has little theoretical grounding and can thus lead to paradoxes. For instance, returning to Figure 1, if we assume a large number of nodes (e.g., $Q_1, ..., Q_n$) reading the same inconsistent state (2) as Q, the system will appear close to agreement (many nodes see the same state), although it is in fact largely inconsistent.

To address the above challenges, this paper makes the following contributions:

- We propose a novel consistency mechanism, termed *UPS* (for *Update-Query Consistency with Primaries and Secondaries*), that provides different levels of eventual consistency within the same system (Sections III-B,III-C). *UPS* combines the update-query consistency protocol proposed in [10] with a two-phase epidemic broadcast protocol (called *GPS*) involving two types of nodes: *Primary* and *Secondary*. *Primary* nodes (the *elite*) seek to receive object modifications as fast as possible while *Secondary* nodes (the *masses*) strive to minimize the amount of temporary inconsistencies they perceive (Figure 2).
- We formally analyze the latency behavior of the *GPS*-part of *UPS* by providing closed-form approximations for the latency incurred by *Primary* and *Secondary* nodes (Section III-D).
- We introduce a novel consistency metric that enables us to quantify the amount of inconsistency experienced by *Primary* and *Secondary* nodes executing *UPS* (Section IV).
- We experimentally evaluate the performance of *UPS* by measuring its consistency and latency properties in large-scale simulated networks of one million nodes (Section V). We show in particular that the cost paid by each class of nodes is in fact very small compared to an undifferentiated system: *Primary* nodes experience a lower latency with levels of inconsistency that are close to those of undifferentiated nodes, while *Secondary* nodes observe less inconsistencies at a minimal latency cost.

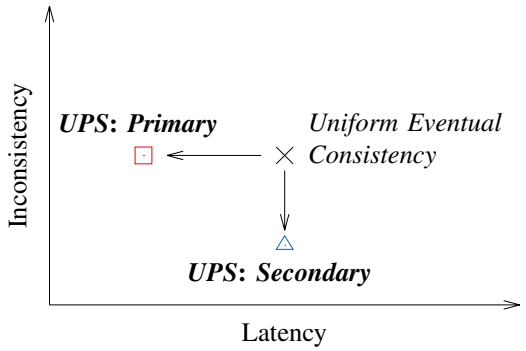


Fig. 2: Aimed consistency/latency trade-off in *UPS*.

II. Background and Problem Statement

A. Update Consistency

As we hinted at, in Section I, eventual consistency requires replicated objects to converge to a globally consistent state when update operations stop for *long enough*. By itself, this condition turns out to be too weak as the convergence state does not need to depend on the operations carried out on the object. For this reason, actual implementations of eventually consistent objects refine eventual consistency by linking the convergence state to its sequential specification.

In this paper, we focus on one such refinement, *Update consistency* [10]. Let us consider the append-only queue object of Figure 1. Its sequential specification consists of two operations:

- $\text{append}(x)$, appends the value x at the end of the queue;
- $\text{read}()$, returns the sequence of all the elements ever appended, in their append order.

When multiple distributed agents update the queue, update consistency requires the final convergence state to be the result of a total ordering of all the append operations which respects the program order. For example, the scenario in Figure 1

[2]This inconsistency causes in particular Q to observe an illegal state transition from (2) to $(1, 2)$.

satisfies update consistency because the final convergence state results from the ordering $< m.\text{append}(1), m.\text{append}(2) >$, which itself respects the program order. An equivalent definition [10] states that an execution history respects update consistency if it contains an infinite number of updates or if it is possible to remove a finite number of reads from it, so that the resulting pruned history is sequentially consistent. In Figure 1, we achieve this by removing the read operation that returns 2.

Algorithm 1 shows an algorithm from [10] that implements the update-consistent append-only queue of Figure 1. Unlike CRDTs [9] that rely on commutative ("conflict-free") operations, Algorithm 1 exploits a broadcast operation together with Lamport Clocks, a form of logical time-stamps that makes it possible to reconstruct a total order of operations after the fact [10]. Relying on this after-the-fact total order allows update consistency to support non-commutative operations, like the queue in this case.

Algorithm 1 Update consistency for an append-only queue

```
1:  variables
2:      int id                                          ▷ Node identifier
3:      set <int, int, V> U ← ∅                         ▷ Set of updates to the queue
4:      int clock_id ← 0                                ▷ Node's logical clock
5:  procedure APPEND(v)                                 ▷ Append a value v to the queue
6:      clock_id ← clock_id + 1
7:      U ← U ∪ {< clock_id, id, v >}
8:      BROADCAST (< clock_id, id, v >)
9:  upon receive (< clock_msg, id_msg, v_msg >) do
10:     clock_id ← MAX(clock_id, clock_msg)
11:     U ← U ∪ {< clock_msg, id_msg, v_msg >}
12: procedure READ( )                                   ▷ Read the current state of the queue
13:     q ← ()                                          ▷ Empty queue
14:     for all < clock, id, v > ∈ U sorted by (clock, id) do
15:         q ← q · v
16:     return q
```

B. Problem Statement

The key feature of update consistency lies in the ability to define precisely the nature of the convergence state reached once all updates have been issued. However, the nature of temporary states also has an important impact in practical systems. This raises two important challenges. First, existing systems address the consistency of temporary states by implementing uniform constraints that all the nodes in a system must follow [18]. But different actors in a distributed application may have different requirements regarding the consistency of these temporary states. Second, even measuring the level of inconsistency of these states remains an open question. Existing systems-oriented metrics do not take into account the ordering of update operations (append in our case) [3], [19], [20], [21], while theoretical ones require global knowledge of the system [22] which makes them impractical at large scale.

In the following sections, we address both of these challenges. First we propose a novel broadcast mechanism that, together with Algorithm 1, satisfies update consistency, while supporting differentiated levels of consistency for read operations that occur before the convergence state. Specifically, we exploit the evident trade-off between speed of delivery and consistency, and we target heterogeneous populations consisting of an elite of *Primary* nodes that should receive fast, albeit possibly inconsistent, information, and a mass of *Secondary* nodes that should only receive stable consistent information, albeit more slowly. Second, we propose a novel metric to measure the level of inconsistency of an append-only queue, and use it to evaluate our protocol.

III. The *GPS* Broadcast Protocol

A. System Model

We consider a large set of nodes $p_1, ..., p_N$ that communicate using point-to-point messages. Any node can communicate with any other node, given its identifier. We use probabilistic algorithms in the following that are naturally robust to crashes and message losses, but do not consider these aspects in the rest of the paper for simplicity. Nodes are categorized in two classes: a small number of *Primary* nodes (the *elite*) and a large number of *Secondary* nodes (the *masses*). The class of a node is an application-dependent parameter that captures the node's requirements in terms of update consistency: *Primary* nodes should perceive object modification as fast as possible, while *Secondary* nodes should experience as few inconsistencies as possible.

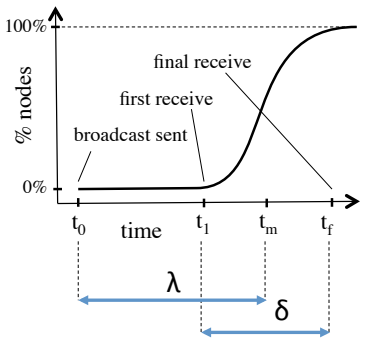


Fig. 3: Two sorts of speeds: latency (λ) and jitter (δ).

B. Intuition and Overview

We have repeatedly referred to the inherent trade-off between speed and consistency in eventually consistent systems. On deeper examination, this trade-off might appear counter-intuitive: if *Primary* nodes receive updates faster, why should not they also experience higher levels of consistency? This apparent paradox arises because we have so far silently confused *speed* and *latency*. The situation within a large-scale broadcast is in fact more subtle and involves two sorts of speeds (Figure 3): *latency* (λ, shown as an average over all nodes in the figure) is the time a message m takes to reach individual nodes, from the point in time of m's sending (t_0). *Jitter* (δ), by contrast, is the delay between the first (t_1)

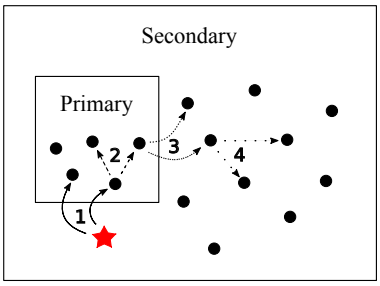


Fig. 4: Model of *GPS* and path of an update in the system.

and the last receipt (t_f) of a broadcast. (In most large-scale broadcast scenarios, $t_0 - t_1$ is small, and the two notions tend to overlap.) Inconsistencies typically arise in Algorithm 1 when some updates have only partially propagated within a system, and are thus predominantly governed by the jitter δ rather than the average *latency* λ.

The gossip-based broadcast protocol we propose, *Gossip Primary-Secondary* (*GPS*), exploits this distinction and proposes different δ/λ trade-offs. *Primary* nodes have a reduced λ (thus increasing the speed at which updates are visible) but also a slightly increased δ, while *Secondary* nodes have a reduced δ (thus increasing consistency by lowering the probability for a node to receive updates in the wrong order) but at the cost of a slightly higher λ.

More precisely, *GPS* uses the set of *Primary* nodes as a sort of message "concentrator" that accumulates copies of an update u before collectively forwarding it to *Secondary* nodes. The main phases of this sequence is shown in Figure 4:

1) A new update u is first sent to *Primary* nodes (**1**);
2) *Primary* nodes disseminate u among themselves (**2**);
3) Once most *Primary* nodes have received u, they forward it to *Secondary* nodes (**3**);
4) Finally, *Secondary* nodes disseminate u among themselves (**4**).

A key difficulty in this sequence consists in deciding when to switch from Phase **2** to **3**. A collective, coordinated transition would require some global synchronization mechanism, a costly and generally impracticable solution in a very large system. Instead, *GPS* relies on a less accurate but more scalable local procedure based on broadcast counts, which enables each *Primary* node to decide locally when to start forwarding to secondaries.

C. The GPS Algorithm

The pseudo-code of *GPS* is shown in Algorithm 2. *GPS* follows the standard models of reactive epidemic broadcast protocols [23], [24]. Each node keeps a history of the messages received so far (in the R variable, line 8), and decide whether to re-transmit a received broadcast to fanout other nodes based on its history. Contrary to a standard epidemic broadcast, however, *GPS* handles *Primary* and *Secondary* nodes differently.

- *First*, *GPS* uses two distinct *Random Peer Sampling* protocols (RPS) [25] (lines 10-11) to track the two classes of nodes. Both *Primary* and *Secondary* nodes use the RPS view of their category to re-transmit a message they receive for the first time to fanout other nodes in their own category (lines 23 and 24), thus implementing Phases **2** and **4**.
- *Second*, *GPS* handles retransmissions differently depending on a node's class (*Primary* or *Secondary*). *Primary* nodes use the inherent presence of message duplicates in gossip protocols, to decide locally when to switch from Phase **2** to **3**. More specifically, each node keeps

Algorithm 2 – *Gossip Primary-Secondary* for a node

```
1:  parameters
2:      int fanout                          ▷ Number of nodes to send to
3:      int rpsViewSize                     ▷ Out-degree per class of each node
4:      boolean isPrimary                   ▷ Class of the node
5:  initialization
6:      set{node} Γ_P ← ∅                   ▷ Set of Primary neighbors
7:      set{node} Γ_S ← ∅                   ▷ Set of Secondary neighbors
8:      map{message, int} R ← ∅             ▷ Counters of message
                                            ▷ duplicates received
9:  periodically
10:     Γ_P ← rpsViewSize nodes from Primary-RPS
11:     Γ_S ← rpsViewSize nodes from Secondary-RPS
12: procedure BROADCAST(msg)                ▷ Called by the application
13:     R ← R ∪ {(msg, 1)}
14:     GOSSIP(msg, Γ_P)
15: procedure GOSSIP(msg, targets)
16:     for all j ∈ {fanout random nodes in targets} do
17:         SENDTONETWORK(msg, j)
18: upon receive (msg) do
19:     counter ← R[msg] + 1
20:     R ← R ∪ {(msg, counter)}
21:     if counter = 1 then DELIVER(msg)    ▷ Deliver 1st receipt
22:     if isPrimary then
23:         if counter = 1 then GOSSIP(msg, Γ_P)
24:         if counter = 2 then GOSSIP(msg, Γ_S)
25:     else
26:         if counter = 1 then GOSSIP(msg, Γ_S)
```

Algorithm 3 – *Uniform Gossip* for a node (baseline)

(only showing main differences to Algorithm 2)

```
9':  periodically Γ ← rpsViewSize nodes from RPS
15': procedure GOSSIP(msg)
16':     for all j ∈ {fanout random nodes in Γ} do
17':         SENDTONETWORK(msg, j)
18': upon receive (msg) do
19':     if msg ∈ R then
20':         return  ▷ Infect and die: ignore if msg already received
21':     R ← R ∪ {msg}
22':     DELIVER(msg)                       ▷ Deliver 1st receipt
23':     GOSSIP(msg)
```

count of the received copies of individual messages (lines 8, 13, 20). *Primary* nodes use this count to detect duplicates, and forward a message to fanout *Secondary* nodes (line 24) when a duplicate is received for the first time, thus triggering Phase **3**.

We can summarize the behavior of both classes by saying that *Primary* nodes *infect twice and die*, whereas *Secondary* nodes *infect and die*. For comparison, a standard *infect and die* gossip without classes (called *Uniform Gossip* in the following), is shown in Algorithm 3. We will use *Uniform Gossip* as our baseline for our analysis (Section III-D) and our experimental evaluation (Section V).

D. Analysis of GPS

In the following we compare analytically the expected performance of *GPS* and compare it to *Uniform Gossip* in terms of message complexity and latency.

Our analysis uses the following parameters:

- Network N of size: $|N| \in \mathbb{N}$;
- Fanout: $f \in \mathbb{N}$;
- Density of *Primary* nodes: $d \in \mathbb{R}_{[0,1]}$ (assumed $\leq 1/f$).

Uniform Gossip uses a simple *infect and die* procedure. For each unique message it receives, a node will send it to f other nodes. If we consider only one source, and assume that most nodes are reached by the message, the number of messages exchanged in the system can be estimated as

$$|msg|_{uniform} \approx f \times |N|. \tag{1}$$

In the rest of our analysis, we assume, following [26], that the number of rounds needed by *Uniform Gossip* to infect a high proportion of nodes can be approximated by the following expression when $N \to \infty$:

$$\lambda_{uniform} \approx \log_f(|N|) + C, \tag{2}$$

where C is a constant, independent of N.

GPS distinguishes two categories of nodes: *Primary* nodes and *Secondary* nodes, noted P and S, that partition N. The density of *Primary* nodes, noted d, defines the size of both subsets:

$$|P| = d \times |N|, \qquad |S| = (1-d) \times |N|.$$

Primary nodes disseminate twice, while *Secondary* nodes disseminate once. Expressed differently, each node disseminates once, and *Primary* nodes disseminate once more. Applying the same estimation as for *Uniform Gossip* gives us:

$$\begin{aligned} |msg|_{GPS} &\approx f \times |N| + f \times |P| \\ &\approx f \times |N| + f \times d \times |N| \\ &\approx (1+d) \times f \times |N| \\ &\approx (1+d) \times |msg|_{uniform}. \end{aligned} \tag{3}$$

(1) and (3) show that *GPS* only generates d times more messages than *Uniform Gossip*, with $d \in \mathbb{R}_{[0,1]}$. For instance, having 1% *Primary* nodes in the network means having only 1% more messages compared to *Uniform Gossip*.

If we now turn to the latency behavior of *GPS*, the latency of the *Primary* nodes is equivalent to that of *Uniform Gossip* executing on a sub-network composed only of $d \times |N|$ nodes, i.e.,

$$\lambda_P \approx \log_f(d \times |N|) + C. \tag{4}$$

Combining (2) and (4), we conclude that *Primary* nodes gain $log_f(d)$ rounds compared to nodes in *Uniform Gossip*:

$$\Delta\lambda_P \approx -log_f(d). \tag{5}$$

Considering now *Secondary* nodes, their latency can be estimated as a sum of three elements:

- the latency of *Primary* nodes;
- an extra round for *Primary* nodes to receive messages a second time;
- the latency of a *Uniform Gossip* among Secondary nodes with $d \times |N|$ nodes, corresponding to the *Primary* nodes, already infected;

which we approximate for $d \ll 1$ as

$$\begin{aligned} \lambda_S &\approx \log_f(d \times |N|) + 1 + \log_f(\frac{(1-d) \times |N|}{d \times |N|}) + 2C, \\ &\approx \log_f((1-d) \times |N|) + 1 + 2C. \end{aligned} \tag{6}$$

In summary, this analysis shows that *GPS* only generates a small number of additional messages, proportional to the density d of *Primary* nodes, and that the latency cost paid by *Secondary* is bounded by a constant value $(1 + C)$ that is independent of the *Primary* density d.

IV. Consistency Metric

Our second contribution is a novel metric that measures the consistency level of the temporary states of an update-consistent execution. As discussed in Section II, existing consistency metrics fall short either because they do not capture the ordering of operations, or because they cannot be computed without global system knowledge. Our novel metric satisfies both of these requirements.

A. A General Consistency Metric

We start by observing that the algorithm for an update-consistent append-only queue (Algorithm 1) guarantees that all its execution histories respect update consistency. To measure the consistency level of temporary states, we therefore evaluate how the history deviates from a stronger consistency model, *sequential consistency* [27]. An execution respects sequential consistency if it is equivalent to some sequential (i.e., totally ordered) execution that contains the same operations, and respects the sequential (process) order of each node.

Since update consistency relies itself on a total order, the gist of our metric consists in counting the number of read operations that do not conform with a total order of updates that leads to the final convergence state. Given one such total order, we may transform the execution into one that conforms with it by removing some read operations. In general, a data object may reach a given final convergence state by means of different possible total orders, and for each such total order

we may have different sets of read operations whose removal makes the execution sequentially consistent. We thus count the level of inconsistency by taking the minimum over these two degrees of freedom: choice of the total order, and choice of the set.

More formally, we define the *temporary inconsistencies* of an execution Ex as a *finite* set of read operations that, when removed from Ex, makes it sequentially consistent. We denote the set of all the temporary inconsistencies of execution Ex over all compatible total orders by $TI(Ex)$. We then define the *relative inconsistency* RI of an execution Ex as the minimal number of read operations that must be removed from Ex to make it sequentially consistent.

$$RI(Ex) = \begin{cases} \min\limits_{E \in TI(Ex)} |E| & \text{if } TI(Ex) \neq \emptyset \\ +\infty & \text{otherwise} \end{cases}$$

For example, in Figure 1, if we consider the total order $<$ $m.\text{append}(1)$, $m.\text{append}(2)$, $m.\text{read}(1)$, $m.\text{read}(2)$, $m.\text{read}(1,2)$, $m.\text{read}(1,2)$ $>$ then we need to remove both $m.\text{read}(1)$ and $m.\text{read}(2)$ to make the execution sequentially consistent. Removing only $m.\text{read}(2)$, instead, suffices to make the execution sequentially consistent with respect to $<$ $m.\text{append}(1)$, $m.\text{read}(1)$, $m.\text{append}(2)$, $m.\text{read}(2)$, $m.\text{read}(1,2)$, $m.\text{read}(1,2)$ $>$. Therefore the level of inconsistency of the execution in Figure 1 is 1.

The metric RI is particularly adapted to compare the consistency level of implementations of update consistency: the lower, the more consistent. In the best case scenario where Ex is sequentially consistent, $TI(Ex)$ is a singleton containing the empty set, resulting in $RI(Ex) = 0$. In the worst case scenario where the execution never converges (i.e., some nodes indefinitely read incompatible local states), every set of reads that needs to be removed to obtain a sequentially consistent execution is infinite. Since TI only contains finite sets of reads, $TI(Ex) = \emptyset$ and $RI(Ex) = +\infty$.

B. The Case of our Update-Consistent Append-Only Queue

In general, $RI(Ex)$ is complex to compute: it is necessary to consider all possible total orders of events that can fit for sequential consistency and all possible finite sets of reads to check whether there are temporary inconsistencies. But in the case of an append-only queue implemented with Algorithm 1, we can easily show that there exists exactly one minimal set of temporary inconsistencies.

To understand why, we first observe that the append operation is non-commutative. This implies that there exists a single total order of append operations that yields a given final convergence state. Second, Algorithm 1 guarantees that the size of the successive sequences read by a node can only increase and that read operations always reflect the writes made on the same node. Consequently, in order to have a sequentially consistent execution, it is necessary and sufficient to remove all the read operations that return a sequence that is not a prefix of the sequence read after convergence. These read operations constitute the minimal set of temporary inconsistencies TI_{min}.

V. Experimental Results

We perform the evaluation of *UPS* via PeerSim [28], a well-known Peer-to-Peer simulator. A repository containing the code and the results is available on-line[3]. To assess the trade-offs between consistency level and latency as well as the overhead of *UPS*, we focus the evaluation on three metrics:

- the level of consistency of the replicated object;
- the latency of messages;
- the overhead in number of messages.

A. Methodology

1) Network Settings: We use a network of 1 million nodes, a fanout of 10 and an RPS view size of 100. These parameters yield a broadcast reliability (i.e., the probability that a node receives a message) that is above 99.9%. Reliability could be further increased with a higher fanout [23], but these parameters, because they are all powers of 10, make it convenient to understand our experimental results.

We use density values of *Primary* nodes of: $10^{-1}, 10^{-2}$ and 10^{-3}. According to Equation 5 in Section III-D, we expect to see a latency gain for *Primary* nodes of 1, 2, and 3 rounds respectively. We evaluate four protocol configurations: one for *Uniform Gossip* (baseline), and three for *UPS* with the three above densities, then run each configuration 25 times and record the resulting distribution.

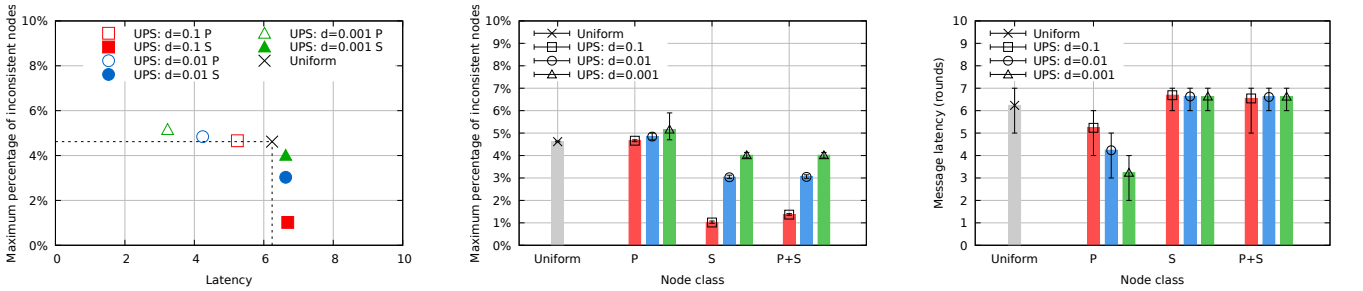
2) Scenario: We consider a scenario where all nodes share an instance of an update-consistent append-only queue, as defined in Section II-A. Following the definition of update consistency, nodes converge into a strongly consistent state once they stop modifying the queue.

We opt for a scenario where 10 $\text{append}(x)$ operations, with $x \in \mathbb{Z}$, are performed on the queue by 10 random nodes, over the first 10 rounds of the simulations at the frequency of one update per round. In addition, all nodes repeatedly read their local copy of the queue in each round.

We expect the system to experience two periods: first, a temporary situation during which updates are issued and disseminated (simulating a system continuously performing updates), followed by a stabilized state after updates have finished propagating and most nodes have converged to a strongly consistent state.

3) Consistency Metric: Our experimental setting enables us to further refine the generic consistency metric we introduced in Section IV. Since nodes perform a finite number of update operations and each node reads the state of the queue at each round, we define a per-round metric to compare the evolution of the inconsistency of *Primary* and *Secondary* nodes through time. For each round r, we define the sets $R_P(r)$ and $R_S(r)$ as the sets of all the read operations performed at round r by the *Primary* and *Secondary* nodes respectively. Then we define per-round inconsistency, $Incons_P(r)$ and $Incons_S(r)$, as the proportion of *Primary* and *Secondary* nodes that see an inconsistent read at round r (i.e., the corresponding reads are in TI_{min}).

[3]https://gforge.inria.fr/projects/pgossip-exp/



(a) Experimental trade-offs between consistency and latency (closer to the bottom left corner is better). A big gain on one side of the balance incurs a small penalty on the other side of the balance for a class.

(b) Consistency trade-off in *UPS* (lower is better). *Secondary* nodes are much more consistent than nodes in the baseline while *Primary* nodes only experience a small consistency penalty. The consistency of both *Primary* and *Secondary* nodes improves as the density of *Primary* nodes increases.

(c) Latency trade-off in *UPS* (lower is better). *Primary* nodes latency is lower than that of the baseline; the less *Primary* nodes, the lower their latency. *Secondary* nodes remain half a round slower than the baseline, regardless of the density of *Primary* nodes in the network.

Fig. 5: Consistency and latency trade-off in *UPS*. *Primary* nodes are faster and a bit less consistent, while *Secondary* are more consistent and a bit slower. The top of the bars is the mean while the ends of the error bars are the 5th and 95th percentiles.

$$Incons_P(r) = \frac{|TI_{min} \cap R_P(r)|}{d \times |N|}$$

$$Incons_S(r) = \frac{|TI_{min} \cap R_S(r)|}{(1-d) \times |N|}$$

$$Incons_{P+S}(r) = d \times Incons_P(r) + (1-d) \times Incons_S(r)$$

In the following, we focus on these *instantaneous* per-round metrics, which provide an equivalent but more accurate view than the simple relative inconsistency of an experimental run, $RI(Ex)$. We can in fact compute $RI(Ex)$ as follows.

$$RI(Ex) = |N| \times \sum_{r=0}^{\infty} (Incons_{P+S}(r)).$$

4) Plots: Unless stated otherwise, plots use boxes and whiskers to represent the distribution of measures obtained for the represented metric. For the inconsistency measures, the distribution is over all runs. For the latency measures, the distribution is over all nodes within all runs. The end of the boxes show the first and third quartiles, the end of the whiskers represent the minimum and maximum values, while the horizontal bar inside the boxes is the mean. In some cases, the low variability of the results makes it difficult to see the boxes and whiskers.

For clarity purposes, curves are slightly shifted to the right to avoid overlap between them. All the points between rounds r and $r+1$ belong to round r. In the following plots, *Primary* nodes are noted P, *Secondary* nodes are noted S and the system as a whole is noted $P+S$. Since only a small fraction of nodes are *Primary* nodes, the results of $P+S$ are naturally close to those of S.

B. Overall Results

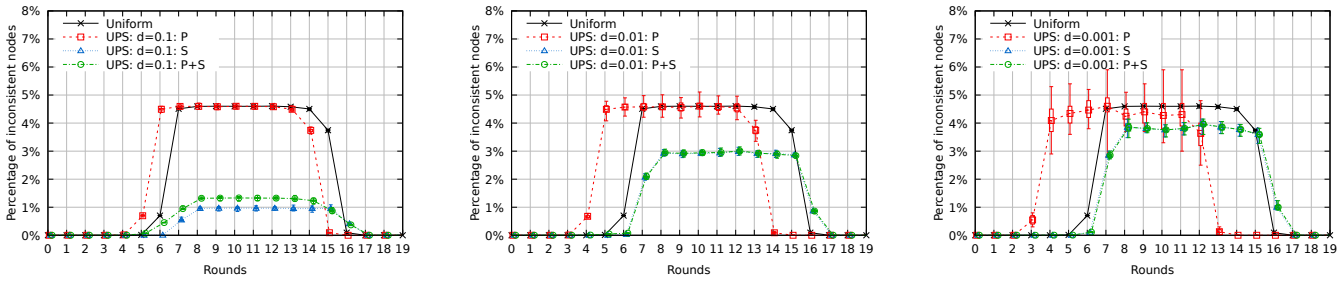
Figure 5a mirrors Figure 2 discussed in Section I and provides an overview of our experimental results in terms of consistency/latency trade-offs for the different sets of nodes involved in our scenario. Each *UPS* configuration is shown as a pair of points representing *Primary* and *Secondary* nodes: *Primary* nodes are depicted with hollow shapes, while *Secondary* nodes use solid symbols. *Uniform Gossip* is represented by a single black cross. The position on the x-axis charts shows the average update latency experienced by each set of nodes, and the y-axis their perceived level of inconsistency, taken as the maximum $Incons_X(r)$ value measured over all runs.

The figure clearly shows that *UPS* delivers the differentiated consistency/latency trade-offs we set out to achieve in our introduction: *Secondary* nodes enjoy higher consistency levels than they would in an uniform update-query consistency protocol, while paying only a small cost in terms of latency. The consistency boost strongly depends on the density of *Primary* nodes in the network, while the cost in latency does not, reflecting our analysis of Section III-D. *Primary* nodes present the reverse behavior, with the latency gains of *Primary* nodes evolving in the reverse direction of the consistency gains of *Secondary* nodes. We discuss both aspects in more details in the rest of this section.

C. Consistency Level

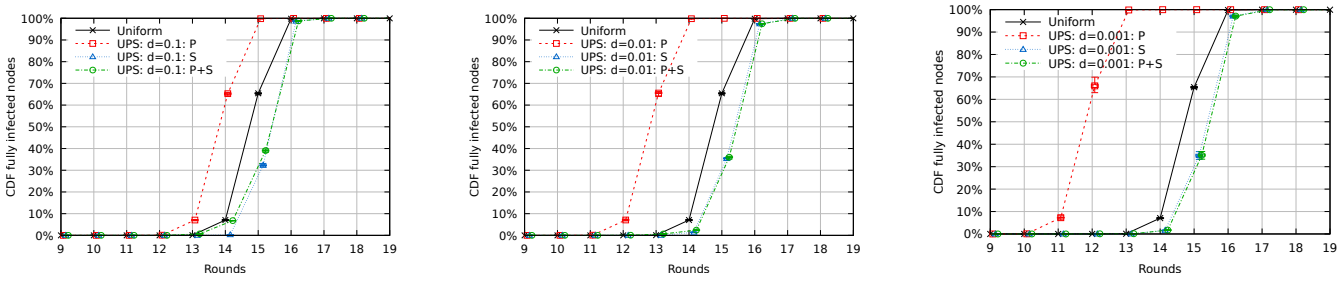
Figure 5b details the consistency levels provided by *UPS* by showing the worst consistency that nodes experience over all simulations. We note an evident improvement of the maximum inconsistency level of *Secondary* nodes over the baseline and a slight decrease for *Primary* nodes. In addition, this figure clearly shows the impact of the density of *Primary* nodes over the consistency of *Secondary* nodes, and to a lesser extent over the consistency of *Primary* nodes.

Figures 6a, 6b and 6c show the evolution over time of the inconsistency measures $Incons_P(r)$, $Incons_S(r)$ and $Incons_{P+S}(r)$ defined in Section V-A3. We can observe an increase of inconsistencies during the temporary phase for all configurations and a return to a consistent state once every node has received every update.



(a) Density $= 10^{-1}$ (b) Density $= 10^{-2}$ (c) Density $= 10^{-3}$

Fig. 6: While updates are being disseminated, *Secondary* nodes are more consistent than the baseline and *Primary* nodes are on average as consistent as the baseline. The more *Primary* nodes are in the system, the more *Secondary* nodes are consistent. Once updates stop being performed, *Primary* nodes converge faster to a strongly consistent state.



(a) Density $= 10^{-1}$ (b) Density $= 10^{-2}$ (c) Density $= 10^{-3}$

Fig. 7: *Primary* nodes receive all the updates faster than the baseline: they gain 1, 2 and 3 rounds for densities of $10^{-1}, 10^{-2}$ and 10^{-3} respectively. Meanwhile *Secondary* nodes receive all the updates only half a round later on average compared to nodes in the baseline.

During the temporary phase, the inconsistency level of *Primary* nodes is equivalent to that of nodes in *Uniform Gossip*. In that phase, around 4.6% of *Primary* nodes are inconsistent with a higher variability for lower densities.

Meanwhile, the inconsistency level of *Secondary* nodes is much lower than that of nodes in *Uniform Gossip*. The density of *Primary* nodes plays a key role in this difference; the higher the density, the lower the inconsistency level of *Secondary* nodes. This level remains under 1.0% for the highest density but goes up to 4.0% for the lowest density.

The jitter, as defined in Section III-B, is a good metric to compare the consistency of different sets of nodes: a lower jitter implies a lower probability for a node to receive updates in the wrong order, which in turn leads to a higher consistency. The error bars in Figure 5c provide an approximation of the jitter of a set of nodes since they represent the bounds in latency of 90% of the nodes in a set. These error bars show that 90% of *Secondary* nodes receive updates within 1 round of margin, while the same proportion of *Primary* nodes and nodes in *Uniform Gossip* receive updates within 2 rounds of margin. This difference in jitter explains why *Secondary* nodes are more consistent than *Primary* nodes and nodes in *Uniform Gossip*.

Once the dissemination reaches a critical mass of *Primary* nodes, the infection of *Secondary* nodes occurs quickly. If the density of *Primary* nodes is high enough, then it becomes possible for a majority of *Secondary* nodes to receive the same update at the same time. Since it takes fewer rounds for *Secondary* nodes to be fully infected compared to *Primary* nodes, *Secondary* nodes turn out to be more consistent.

D. Message Latency

Figure 5c represents the distribution of all the values of message latency over all the simulation runs. The three $P+S$ bars and the *Uniform* bar contain each 250 million message latency values (25 runs $\times$ 1 million nodes $\times$ 10 sources with a reliability above 99.9%).

This figure shows that *UPS* infects *Primary* nodes faster than *Uniform Gossip* would. Specifically, *Primary* nodes obtain a latency gain of 1, 2 and 3 rounds with densities of 10^{-1}, 10^{-2} and 10^{-3} respectively. *Secondary* nodes, on the other hand, are infected half a round slower than nodes in *Uniform Gossip* for all density values.

Figures 7a, 7b and 7c compare the evolution of the dissemination of updates between *Uniform Gossip* and *UPS* with different densities. Again, *Primary* nodes have a 1, 2 and 3 rounds latency advantage over the baseline while *Secondary* nodes are no more than a round slower.

We can also observe this effect on Figure 6 by looking at how fast all the nodes of a class return to a consistent state. We notice similar latency gain for *Primary* nodes and loss for *Secondary* nodes.

Overall, the simulation results match the analysis in Section III-D and confirm the latency advantage of *Primary* nodes over *Uniform Gossip* (Equation 5) and the small latency penalty of *Secondary* nodes (Equation 6).

E. Network Overhead

The number of messages exchanged in the simulated system confirms Equation 3 in Section III-D. Considering the experienced reliability, we observe in *UPS* an increase in the number of messages of 10^{-1}, 10^{-2} and 10^{-3} compared to *Uniform Gossip* for all three densities of 10^{-1}, 10^{-2} and 10^{-3} respectively.

VI. Related Work

UPS lies at the crossroad between differentiated consistency and gossip protocols (for the *GPS*-part). In the following, we review some of the most relevant works from these two areas.

a) Differentiated Consistency: A large number of works have looked at hybrid consistency conditions, originally for distributed shared memory [16], [29], [30], and more recently in the context of geo-distributed systems [14], [15], [17], [31]. *Fisheye* [15] and *RedBlue* [17] for instance both propose to implement hybrid conditions for geo-replicated systems. Fisheye consistency provides a generic approach in which nodes that are topologically close satisfy a strong consistency criterion, such as sequential consistency, while remote nodes satisfy a weaker one, such as causal consistency. This formal work focuses exclusively on immediate (i.e., non-eventual) consistency criteria and does not take convergence speed into account. RedBlue consistency offers a trade-off similar to that of *UPS*, but focuses on operations, rather than nodes, as we do. Blue operations are fast and eventually consistent while red operations are slow and strongly consistent.

b) Measuring Inconsistency: Several papers have proposed metrics to evaluate a system's overall consistency. The approach of Zellag and Kemme [22] detects inconsistencies in cloud services by finding cycles in a *dependency graph* composed of transactions (nodes) and conflicts between them (edges). Counting cycles in the dependency graph yields a measure of consistency that is formally grounded. It requires however a global knowledge of the system, which makes it difficult to use in practice in large-scale systems. Golab et al. introduced first Δ-atomicity [19], [32] and then Γ [20], two metrics that quantify data staleness against Lamport-atomicity [33] in key-value store traces. These metrics are not suitable for our problem since they do not take into account the ordering of update operations.

More practical works [3], [21] evaluate consistency by relying on system specific information such as the similarity between different cache levels or the read-after-write latency (the first time a node reads the value that was last written).

Finally, CRDTs [9] remove the need to measure consistency by only supporting operations that cannot create conflicts. This naturally leads to eventual consistency without additional ordering requirements on communication protocols.

c) Biased Gossip Protocols: Many gossip broadcast protocols use biases to accommodate system heterogeneity. However, to the best of our knowledge, *GPS* is the first such protocol to target heterogeneous consistency requirements.

Directional gossip [34], for instance, favors weakly connected nodes in order to improve its overall reliability. It does not target speed nor consistency, as we do. The work in [35] looks at reducing a broadcast's message complexity by considering two classes of user-defined nodes: *good* and *bad* nodes. A new broadcast is disseminated to good nodes first using a reactive epidemic protocol, while bad nodes are reached through a slower periodic push procedure. As a result, the overall number of messages is reduced, at the cost of higher delivery latency for *bad* nodes. Similarly, *Gravitational gossip* [36] proposes a multicast protocol with differential reliability to better balance the communication workload between nodes, according to their capacities. Gravitational gossip associates each node with a susceptibility S_r and an infectivity I_r value that depend on a user-defined quality rating r. Nodes of rating r receive a fraction r of the messages before they time out. Gravitational gossip thus offers a cost/reliability trade-off, while *GPS* considers a consistency/latency trade-off. *Hierarchical gossip* [37] also aims to reduce overheads but focuses on those associated with the physical network topology. To this end, it favors gossip targets that are close in the network hierarchy, which leads to a slight decrease in reliability and an increase of delivery latency.

In the context of video streaming, HEAP [38] adapts the fanout of nodes to reduce delivery latency in the presence of heterogeneous bandwidth capabilities. In addition, nodes do not wait for late messages, they simply ignore them. This dropping policy is well adapted to video streaming but cannot be applied to *GPS*. Finally, epidemic total order algorithms such as EpTO [11] and ecBroadcast [12] can be used to implement (probabilistic) strong consistency conditions, but at the cost of higher latency, and a higher number of messages for EpTO.

VII. Conclusion

We have presented *Update-Query Consistency with Primaries and Secondaries* (*UPS*), a novel eventual consistency mechanism that offers heterogeneous properties in terms of data consistency and delivery latency. *Primary* nodes can deliver updates faster at the cost of a small consistency penalty, while *Secondary* nodes experience stronger consistency but with a slightly higher latency. Both sets of nodes observe a consistent state with high probability once dissemination completes.

To measure the different levels of consistency observed by *Primary* and *Secondary* nodes, we have proposed a novel and scalable consistency metric grounded in theory. This metric

detects when a node performs read operations that conflict with the sequential specification of its associated object.

We formally analyzed the latency incurred by nodes as well as the overhead in message complexity in *GPS*, the underlying two-phase epidemic broadcast protocol. We then evaluated both the consistency and latency properties of *UPS* by simulating a 1 million node network. Results show how the density (fraction) of *Primary* nodes influences the trade-off between consistency and latency: lower densities favor fast dissemination to *Primary* nodes, while higher densities favor higher consistency for *Secondary* nodes.

Our future plans include deploying *UPS* in a real system and performing experiments to confront the algorithm to real-life conditions. We also plan to investigate how *UPS* could be combined with a complementary anti-entropy protocol [39] to reach the last few susceptible nodes and further improve its reliability.

ACKNOWLEDGMENTS

This work has been partially funded by the Region of Brittany, France, by the Doctoral school of the University of Brittany Loire (UBL) and by the French National Research Agency (ANR) project *SocioPlug* under contract ANR-13-INFR-0003 (http://socioplug.univ-nantes.fr).

REFERENCES

[1] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *EuroSys*. ACM, 2015.

[2] "Google data centers, data center locations," https://www.google.com/about/datacenters/inside/locations/index.html, accessed Sep. 10 2015.

[3] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd, "Existential Consistency: Measuring and Understanding Consistency at Facebook," in *SOSP*. ACM, 2015.

[4] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, 2002.

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *ACM SIGMOD Int. Conf. on Man. of Data*, 2013.

[6] S. Almeida, J. Leitão, and L. Rodrigues, "Chainreaction: a causal+ consistent datastore based on chain replication," in *EuroSys*. ACM, 2013.

[7] 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 *SOSP*. ACM, 2011.

[8] W. Vogels, "Eventually Consistent," *CACM*, vol. 52, no. 1, Jan. 2009.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *SSS*, 2011.

[10] M. Perrin, A. Mostefaoui, and C. Jard, "Update Consistency for Wait-free Concurrent Objects," in *IPDPS*. IEEE, 2015.

[11] M. Matos, H. Mercier, P. Felber, R. Oliveira, and J. Pereira, "EpTO: An Epidemic Total Order Algorithm for Large-Scale Distributed Systems," in *Middleware*. ACM, 2015.

[12] R. Baldoni, R. Guerraoui, R. R. Levy, V. Quéma, and S. T. Piergiovanni, "Unconscious Eventual Consistency with Gossips," in *SSS*, 2006.

[13] A. Sousa, J. Pereira, F. Moura, and R. Oliveira, "Optimistic total order in wide area networks," in *SRDS*. IEEE, 2002.

[14] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan, "Salt: Combining acid and base in a distributed database," in *OSDI*. USENIX, 2014.

[15] R. Friedman, M. Raynal, and F. Taïani, "Fisheye Consistency: Keeping Data in Synch in a Georeplicated World," in *NETYS*, 2015.

[16] R. Friedman, "Implementing hybrid consistency with high-level synchronization operations," *Dist. Comp.*, vol. 9, no. 3, 1995.

[17] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *OSDI*. USENIX, 2012.

[18] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, "Replicated data types: Specification, verification, optimality," *SIGPLAN Not.*, vol. 49, no. 1, 2014.

[19] W. Golab, X. Li, and M. A. Shah, "Analyzing consistency properties for fun and profit," in *PODC*. ACM, 2011.

[20] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and I. Gupta, "Client-centric benchmarking of eventual consistency for cloud storage systems," in *ICDCS*. IEEE, 2014.

[21] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "YCSB++: benchmarking and performance debugging advanced features in scalable table stores," in *Symp. on Cloud Comp. (SoCC)*. ACM, 2011.

[22] K. Zellag and B. Kemme, "How consistent is your cloud application?" in *Symp. on Cloud Comp. (SoCC)*. ACM, 2012.

[23] A.-M. Kermarrec, L. Massoulie, and A. Ganesh, "Probabilistic reliable dissemination in large-scale systems," *IEEE TPDS*, vol. 14, no. 3, 2003.

[24] F. Taïani, S. Lin, and G. S. Blair, "GossipKit: A unified component-framework for gossip," *IEEE TSE*, vol. 40, no. 2.

[25] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "Gossip-based peer sampling," *ACM ToCS*, vol. 25, no. 3, 2007.

[26] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, "From epidemics to distributed computing," *IEEE computer*, vol. 37, no. LPD-ARTICLE-2006-004, pp. 60–67, 2004.

[27] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE ToC*, vol. 100, no. 9, 1979.

[28] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *P2P*. IEEE, 2009.

[29] H. Attiya and R. Friedman, "Limitations of fast consistency conditions for distributed shared memories," *Inf. Proc. Letters*, vol. 57, no. 5, 1996.

[30] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *ISCA*. ACM, 1992.

[31] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *SOSP*. ACM, 2013.

[32] M. R. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. J. Wylie, "Toward a principled framework for benchmarking consistency," in *HotDep*. USENIX, 2012.

[33] L. Lamport, "On interprocess communication," *Distributed Computing*, vol. 1, no. 2, 1986.

[34] M.-J. Lin and K. Marzullo, "Directional Gossip: Gossip in a Wide Area Network," in *EDCC*, 1999.

[35] N. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues, "Emergent Structure in Unstructured Epidemic Multicast," in *DSN*, 2007.

[36] K. Hopkinson, K. Jenkins, K. Birman, J. Thorp, G. Toussaint, and M. Parashar, "Adaptive Gravitational Gossip: A Gossip-Based Communication Protocol with User-Selectable Rates," *IEEE TPDS*, vol. 20, no. 12, 2009.

[37] I. Gupta, A.-M. Kermarrec, and A. Ganesh, "Efficient and adaptive epidemic-style protocols for reliable and scalable multicast," *IEEE TPDS*, vol. 17, no. 7, 2006.

[38] D. Frey, R. Guerraoui, A.-M. Kermarrec, B. Koldehofe, M. Mogensen, M. Monod, and V. Quéma, "Heterogeneous Gossip," in *Middleware*. ACM, 2009.

[39] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic Algorithms for Replicated Database Maintenance," in *PODC*. ACM, 1987.