Saturn: a Distributed Metadata Service for Causal Consistency

Manuel Bravo[*†], Luís Rodrigues[†], Peter Van Roy[*]

[*]Université Catholique de Louvain, Belgium

[†]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Abstract

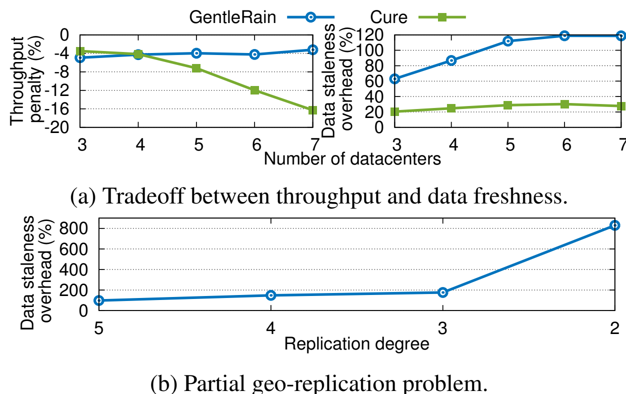
This paper presents the design, implementation, and evaluation of SATURN, a metadata service for geo-replicated systems. SATURN can be used in combination with several distributed and replicated data services to ensure that remote operations are made visible in an order that respects causality, a requirement central to many consistency criteria.

SATURN addresses two key unsolved problems inherent to previous approaches. First, it eliminates the tradeoff between throughput and data freshness, when deciding what metadata to use for tracking causality. Second, it enables *genuine* partial replication, a key property to ensure scalability when the number of geo-locations increases. SATURN addresses these challenges while keeping metadata size constant, independently of the number of clients, servers, data partitions, and locations. By decoupling metadata management from data dissemination, and by using clever metadata propagation techniques, it ensures that the throughput and visibility latency of updates on a given item are (mostly) shielded from operations on other items or locations.

We evaluate SATURN in Amazon EC2 using realistic benchmarks under both full and partial geo-replication. Results show that weakly consistent datastores can lean on SATURN to upgrade their consistency guarantees to causal consistency with a negligible penalty on performance.

1. Introduction

The problem of ensuring consistency in applications that manage replicated data is one of the main challenges of distributed computing. The observation that delegating consistency management entirely to the programmer makes the application code error prone [11] and that strong consistency conflicts with availability [18, 28] has spurred the quest for meaningful consistency models, that can be supported effectively by the data service.



(a) Tradeoff between throughput and data freshness.

(b) Partial geo-replication problem.

Figure 1: Problems faced by current causally consistent geo-replicated storage systems. Results are normalized against eventual consistency.

Among the several invariants that may be enforced, ensuring that updates are applied and made visible respecting causality has emerged as a key ingredient among the many consistency criteria and client session guarantees that have been proposed and implemented in the last decade. Mechanisms to preserve causality can be found in systems that offer from weaker [5, 39, 45, 53] to stronger [12, 38, 49] consistency guarantees. In fact, causal consistency is pivotal in the consistency spectrum, given that it has been proved to be the strongest consistency model that does not compromise availability [8, 41].

Unfortunately, the designer of a causally consistent geo-replicated storage system is still faced today with a dilemma: there appears to be a tradeoff between throughput and data freshness, derived from the granularity at which causality is tracked [17, 31]. Figure 1a reports the results of an experiment that illustrates this tradeoff in current systems. Namely, we compare the performance of two state-of-the-art solutions, GentleRain [26] and Cure [3]. The former opts for a coarse-grained tracking by compressing metadata into a single scalar. The latter opts for a more fine-grained approach by relying on a vector clock with an entry per datacenter. The performance is compared against a weakly consistent store

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '17, April 23-26, 2017, Belgrade, Serbia

© 2017 ACM. ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064210

that does not need to manage metadata. As it can be seen, by keeping little metadata, GentleRain induces a low penalty on throughput but hampers the visibility latency. This is due to the large amount of *false dependencies* inevitably introduced when compressing metadata [21, 22] (a false dependency is created when two concurrent operations are serialized as an artifact of the metadata management). The opposite happens with Cure, that exhibits a low (constant) visibility latency penalty but severely penalizes the throughput due to the computation and storage overhead associated with the metadata management [9, 26].

Furthermore, current solutions are not designed to fully take advantage of partial geo-replication, a setting of practical relevance [19, 23]. The culprit is that causal graphs are not easily partitionable, which may force sites to manage not only the metadata associated with the data items stored locally, but also the metadata associated with items stored remotely [9, 39, 53]. This fact exacerbates the problem of false dependencies, forcing solutions to delay the visibility of remote updates due to updates on data items that are not even replicated locally. To illustrate this problem we run an experiment in which we start from full replication incrementally decreasing the replication degree of each item, until only datacenters close to each other replicate the same data. Figure 1b shows the additional latency that is introduced to enforce causal consistency. One can observe that GentleRain is incapable of taking advantage of partial replication, imposing longer delays as the number of geolocations grows.

In this paper, we present a novel metadata service, named SATURN, that can be used by geo-replicated data services to efficiently ensure causal consistency across geo-locations. The design of SATURN brings two main contributions to mitigate the above problems:

1. SATURN eliminates the tradeoff between throughput and data freshness inherent to previous solutions. To avoid impairing throughput, SATURN keeps the size of the metadata small and constant, independently of the number of clients, servers, partitions, and locations. By using clever metadata propagation techniques, we also ensure that the visibility latency of updates approximates that of weak-consistent systems that are not required to maintain metadata or to causally order operations.
2. SATURN allows data services to fully benefit from partial geo-replication, by implementing *genuine* partial replication [30], requiring datacenters to manage only the metadata concerning data items replicated locally.

Furthermore, by decoupling the metadata management from the data service, SATURN relieves the storage system from tasks related to the maintenance of causal consistency across different geolocations. This is relevant because, as our extensive evaluation will show, when performing fine-grained causality tracking under partial geo-replicated scenarios, such maintenance tasks may have a large negative impact on the system performance.

We have built a prototype of SATURN that we have deployed on Amazon EC2. Our evaluation using both microbenchmarks and a realistic Facebook-based benchmark shows that eventually consistent systems can use SATURN to upgrade to causal consistency with negligible performance overhead (namely, with only 2% reduction in throughput and 11.7ms of extra data staleness) under both full and partial geo-replication. Furthermore, our solution offers significant improvements in terms of throughput (38.3%) compared to previous solutions that favor data freshness [3]; while providing significantly lower remote visibility latencies (76.9ms less on average) compared to previous solutions that favor high throughput [26].

2. Design Overview

SATURN is a metadata service designed to be easily attached to already existing geo-replicated data services to orchestrate inter-datacenter update visibility.

Goal SATURN aims to transform, with negligible performance penalty under both full and partial geo-replication, storage systems providing almost no consistency guarantees into systems ensuring that: *clients always observe a causally consistent state (as defined in [1, 37]) of the storage system independently of the accessed datacenter*. It follows that: i) the metadata handled by SATURN has to be small and fixed in size, independently of the system's scale; ii) the impact of *false dependencies* [21, 22], unavoidably introduced when compressing metadata, has to be mitigated; and, for obvious scalability reasons, (iii) a datacenter should not receive or store any information relative to data that it does not replicate (i.e., it must support *genuine* partial replication [30]).

SATURN is devoted exclusively to metadata management. Thus, it assumes the existence of some bulk-data transfer scheme that fits the application business requirements. The decoupling between data and metadata management is key in the design of SATURN. First, it relieves the datastore from managing consistency across datacenters, a task that may be highly costly [9, 26]. Second, this separation permits SATURN to handle heavier loads independently of the size of the managed data. To the best of our knowledge, SATURN is the first decentralized implementation of a metadata manager for causal consistency (a centralized metadata service has been previously proposed in [27]).

SATURN only manages small pieces of metadata, called *labels*. Labels uniquely identify operations and have constant size. Datacenters are responsible for (i) generating labels (when clients issue update requests), (ii) passing them to SATURN, in an order that respects causality; and (iii) attaching them to its corresponding update payload before delivering the updates to the bulk-data transfer mechanism. SATURN is then responsible for propagating labels among datacenters and for delivering them to each interested datacenter in causal order. In turn, each datacenter applies remote updates locally when it has received both the update payload

(via the bulk-data transfer mechanism), and its corresponding label from SATURN.

In addition, labels include information w.r.t the data being updated. Based on this information, SATURN can selectively deliver labels to only the set of interested datacenters, enabling genuine partial replication.

Labels are comparable and can be totally ordered globally. The total order defined by labels respects causality. In particular, given two updates, a and b, if b causally depends on a (denoted $a \rightsquigarrow b$) then $l_a < l_b$. Similarly to Lamport clocks [37], the converse is not necessarily true, i.e. having $l_x < l_y$ does not necessarily indicate that $x \rightsquigarrow y$. In reality, x could be concurrent with y and still $l_x < l_y$. This derives from the fact that causal order is a partial order and, therefore, there are several serializations of the labels that respect causality (the serialization defined by their timestamps is just one of these). This property of causality is used by SATURN to provide to each datacenter a different serialization of the labels that is crafted to maximize the performance of that datacenter. Thus, datacenters do not necessarily apply updates in the global total order defined by timestamps, but in the order defined by SATURN to each of them (these orders may differ, but all respect causality). Interestingly, a datacenter may always fallback to make updates visible in timestamp order in the unlikely case of a SATURN outage (SATURN has been implemented as a fault-tolerant service), increasing the robustness and availability of the architecture.

We assume that clients communicate via the storage system (with no direct communication among them). A client normally connects to a single datacenter (named the preferred datacenter). Clients may switch to other datacenters if they require data that it is not replicated locally, if their preferred datacenter becomes unreachable, or when roaming. Clients maintain a label that captures their causal past (more precisely, this is maintained by library code that runs with the client). This label is updated whenever the client reads or writes an item in the datastore if the new operation is not already included in the client's causal history. The client label is also used to support safe—without violating causality—client migration among datacenters.

Finally, like many other competing systems [3, 25, 26, 39, 40], SATURN assumes that each storage system datacenter is linearizable [33]. This simplifies metadata management without incurring any significant drawback: previous work has shown that linearizability can be scalably implemented in the local area [6], where latencies are low and network partitions are expected to only occur very rarely, especially in modern datacenter networks that incorporate redundant paths among servers [4, 29].

3. Labels

SATURN implements labels as follows. Each label is a tuple $\langle$*type*, *src*, *ts*, *target*$\rangle$ that includes the following fields: *type* captures the type of the label. SATURN uses two different label types, namely *update* and *migration*. An *update* label is generated when a client issues a write request. A *migration* label is created when a client needs to migrate to another datacenter. Migration labels are not strictly required to support client movement but may speedup this procedure; (ii) *src* (source) includes the unique identifier of the entity that generated the label; (iii) *ts* (timestamp) is a single scalar; and (iv) *target* indicates either the data item that has been updated (meaningful for update labels), or the destination datacenter (meaningful for migration labels).

Labels have the following properties:

Uniqueness. The combination of the *ts* and *src* fields makes each label unique.

Comparability. Let l_a and l_b be two labels assigned to different updates by SATURN. Assuming that source ids are totally ordered, we say that $l_a < l_b$ iff:

$$l_a.ts < l_b.ts \lor l_a.ts = l_b.ts \land l_a.src < l_b.src$$

4. Datacenter Operation

The design of SATURN is decoupled from the implementation details of each datacenter. In this way, SATURN can be cast to operate with different geo-replicated data services. Naturally, SATURN needs to interact with each datacenter, and the datacenter implementation must allow attaching the hooks that provide the functionality required by SATURN, such as the generation of labels associated with each update.

In this section, we present an abstract decomposition of the datacenter operation, which highlights the interactions that are relevant for the understanding of SATURN's operation. These components may be implemented in many different ways; in fact, the literature is rich in systems that can be modeled using our abstract decomposition [5, 32, 45, 53]. Our goal is not to delve in the details of those implementations, but to equip the reader with sufficient information to understand how clients interact with datacenters, how labels are generated, how labels are handled to SATURN, and how remote updates can be applied locally.

We abstract the operation of a datacenter by decomposing its functionality in the following subcomponents, as illustrated by Figure 2:

- Stateless *frontends* shield clients from the details of the internal operation of the datacenter (how many storage servers exist, how many replicas of each item are kept, etc). Frontends intercept client requests before they are processed by storage servers. They have two roles: (i) to ensure that clients observe a causally consistent snapshot of the datastore; and (ii) to forward updates to responsible storage servers and later return the labels assigned to the operations to clients.
- *Gears* are responsible for generating labels, and propagating the data and metadata associated to each update. A *gear* is associated to each storage server; it intercepts update requests (coming from a frontend) and, once it

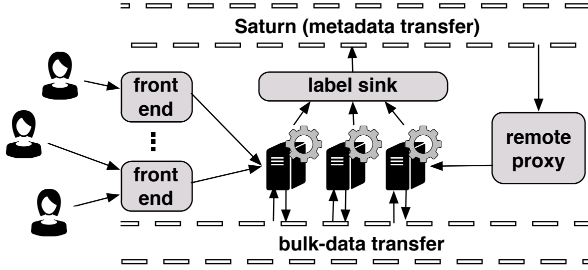


Figure 2: Datacenter Operation.

is made persistent, ships the update to remote datacenters via the bulk-data transfer service. Furthermore, it forwards updates associated labels to the *label sink*.

- The *label sink* is a logically centralized component that collects all the labels associated with the updates performed in the local datacenter and forwards them to the inter-dc module of SATURN, in a serial order that is compliant with causality. Several techniques can be used to establish such order. We found that the most efficient way of doing this is by asynchronously collecting labels from all gears and then periodically ordering such labels according to their timestamps [32].
- The *remote proxy* applies remote operations in causal order. For this, it relies on the order proposed by SATURN and on the label timestamp order as explained in §4.3.

Our design assumes that a datacenter is able to export a single serial stream of updates, as if it were a logically centralized store, even when it is implemented by multiple storage servers. However, the coordination required to establish such serialization may be performed in the background, and does not need to be in the critical path of clients, in opposition to implementations such as [5, 53]. An example of a highly performant datastore that executes such serialization in the background is described in [32].

4.1 Client Interaction

Clients interact directly with SATURN frontends through the client library. A frontend exports four operations: *attach*, *read*, *write*, and *migrate*. The latter is described in §4.4. Algorithms 1 and 2 describe how events are handled by frontends and gears, the two key subcomponents to understand the generation of labels and the interaction with clients.

Attach. Before issuing an update, read, or migration request, a client c is required to *attach* to a datacenter. Being attached to a datacenter m signifies that client c causal past is visible in m and, therefore, c can safely interact with m without violating causality. A client attaches to a datacenter by providing the latest label it has observed (stored in the client's library). The frontend waits until that label is *causally stable*, i.e., until it is sure that all updates that are in the causal past of the client have been locally applied. When this condition is met, it replies back to the client. From

Algorithm 1 Operations at frontend q of dc m

1: ▷ Handles the attachement of a client to a datacenter
2: **function** ATTACH($Label_c$)
3: $\quad g_n^k \leftarrow Label_c.src$ ▷ gear n of dc k
4: $\quad$ **if** $k == m$ **then** ▷ $Label_c$ was locally generated
5: $\quad\quad$ **return** *ok*
6: $\quad$ **else** ▷ $Label_c$ was remotely generated
7: $\quad\quad$ WAIT_FOR_STABILIZATION($Label_c$)
8: $\quad\quad$ **return** *ok*
9: ▷ Forwards an update request to the responsible storage server. Note the operation is intercepted by the gear attached to it.
10: **function** UPDATE(*Key*, *Value*, $Label_c$)
11: $\quad$ *server* ← RESPONSIBLE(*Key*)
12: $\quad$ **send** UPDATE(*Key*, *Value*, $Label_c$) **to** *server*
13: $\quad$ **receive** *Label* **from** *server*
14: $\quad$ **return** *Label*
15: ▷ Forwards a read request to the responsible storage server. Note the operation is intercepted by the gear attached to it.
16: **function** READ(*Key*)
17: $\quad$ *server* ← RESPONSIBLE(*Key*)
18: $\quad$ **send** READ(*Key*) **to** *server*
19: $\quad$ **receive** $\langle$*Value*, *Label*$\rangle$ **from** *server*
20: $\quad$ **return** $\langle$*Value*, *Label*$\rangle$
21: ▷ Forwards a migration request to any gear
22: **function** MIGRATE(*TargetDC*, $Label_c$)
23: $\quad g_n^m \leftarrow$ GEAR(*random_key*) ▷ gear n of local dc m
24: $\quad$ **send** MIGRATION(*TargetDC*, $Label_c$) **to** g_n^m
25: $\quad$ **receive** *Label* **from** g_n^m
26: $\quad$ **return** *Label*

this point on, the client may issue requests. The condition that indicates the stability of the presented label depends on the type and source of the label. If the label was created on the same datacenter, the fronted may return immediately (Alg. 1, line 5). If the label was created on a remote datacenter (Alg. 1, line 6), and it is of type *migration* (§4.4 discusses the generation of this type of labels), it waits until SATURN delivers that label and all previous labels have been applied (in the order provided by SATURN). Finally, if the label was created on a remote datacenter, and it is of type *update*, the frontend waits until an update with an equal or greater timestamp has been applied from every remote datacenter.

Update. A client c's update request is first intercepted by the client library, then tagged with the label that captures the client's causal past ($Label_c$), and forwarded to any local frontend. The frontend forwards the update operation to the local responsible storage server (Alg. 1, line 11). This operation is intercepted by the gear attached to that storage server. The gear first generates a new label for that update (Alg. 2, line 3). Then, the value and its associated label are persistently written to the store. Subsequently, the update's payload —tagged with its corresponding label— is sent to the remote replicas (Alg. 2, lines 6–7), and the label is handed to the *LabelSink* (Alg. 2, line 8). The new label is then returned to the frontend that forwards it to the client library. Finally, the new label replaces the client's old label, capturing the update operation in the client's causal past.

Algorithm 2 Operations at gear n of dc m (g_n^m)

```
 1: ▷ Updates the local store and propagates to remote datacenters
 2: function UPDATE(Key, Value, Label_c)
 3:     TStamp ← GENERATE_TSTAMP(Label_c)
 4:     Label ← ⟨update, TStamp, g_n^m, Key⟩
 5:     ok ← KV_PUT(Key, ⟨Value, Label⟩)
 6:     for all k ∈ REPLICAS(Key) \ {m} do
 7:         send NEW_PAYLOAD(Label, Value) to k
 8:     send NEW_LABEL(Label) to LabelSink
 9:     return Label
10: ▷ Reads the most recent version of Key from the local store
11: function READ(Key)
12:     ⟨Value, Label⟩ ← KV_GET(Key)
13:     return ⟨Value, Label⟩
14: ▷ Generates a migration label
15: function MIGRATION(TargetDC, Label_c)
16:     TStamp ← GENERATE_TSTAMP(Label_c)
17:     Label ← ⟨migration, TStamp, g_n^m, TargetDC⟩
18:     send NEW_LABEL(Label) to LabelSink
19:     return Label
```

Read. A read request on a data item *Key* is handled by a frontend f_q^m by forwarding the request to the local responsible storage server (Alg. 1, line 17). The request is intercepted by the gear g_n^m attached to the storage server that returns the associated value and label (Alg. 2, line 13). If the label associated with the value is greater than the label stored in the client's library ($Label_c$), the library will replace the old label by the new one, including thus the retrieved update into the client's causal past.

4.2 Label Generation

SATURN requires labels to be unique and their timestamp order to respect causality. The former is accomplished by ensuring that each gear generates monotonically increasing timestamps, as the combination of a label's fields timestamp and source make it unique. The latter requires that, when generating a label for an update issued by some client c, the timestamp assigned to that label is strictly greater than all the labels that c has previously observed. In SATURN, each client's causal past is represented by the greatest label the client has observed when interacting with the system ($Label_c$). As clients are not tied to a specific frontend, this label has to be stored in the client's library and be piggybacked with client requests. Therefore, upon an update request, gears only need to guarantee that the timestamp of the label being generated (Alg. 2, lines 3 and 16) is greater than the client's label timestamp. Note that to ensure correctness, client libraries have to update client's labels (as described in §4.1) when they interact with SATURN frontends to ensure that all operations observed by the client are included in the client's causal past.

4.3 Handling Remote Operations

The remote proxy collects updates generated at remote datacenters and applies them locally, in causal order. To derive an order that does not violate causality, it combines two sources of information: the timestamp order of the labels associated with the updates (that defines one valid serialization order), and the label serialization provided by SATURN, which also respects causal order (although it may differ from the timestamp order). As we will show in the evaluation section, SATURN can establish a valid remote update serialization order significantly faster than what is feasible when just relying on timestamp values. Therefore, unless there is an outage on the metadata service, the serialization provided by SATURN is used to apply remote updates, and timestamp order is used as a fallback. These two serialization orders can also be leveraged by the remote proxy to infer that two remote operations a and b are concurrent: this can be inferred if SATURN delivers their corresponding labels (l_a and l_b) in an order that does not match timestamp order. This fact can be exploited by remote proxies to increase the parallelism when handling remote operations. By using this optimization, remote proxies can issue multiple remote operations in parallel to the local datacenter.

4.4 Client Migration Support

Applications may require clients to switch among datacenters, especially under partial replication, in order to read data that is not replicated at the client's preferred datacenter. In order to speedup the attachment at remote datacenters, SATURN (frontends specifically) expose a migration operation. When a client c –attached to a datacenter m– wants to switch to a remote datacenter, a migration request is sent to any local frontend f_q^m, specifying the target datacenter (*TargetDC*) and the client's causal past $Label_c$. f_q^m forwards the request to any local gear. The receiving gear g_n^m generates a new label and hands it to the local *LabelSink* (Alg. 2, lines 16–18). g_n^m guarantees that the generated label is greater than $Label_c$ to ensure that *LabelSink* hands it to SATURN after any update operation that c has potentially observed. In turn, SATURN will deliver the label in causal order to the target datacenter, which will immediately allow client c to attach to it, as c's causal past is ensured to be visible locally.

The procedure above, in particular the creation of a migration label, is not strictly required to support client migration, but aims at optimizing this process. In fact, when attaching to a new datacenter, the client could just present the update label that captures its causal past. However, the stabilization procedure could force the client to wait until an update from each remote datacenter with a timestamp equal or greater than the timestamp of the client's label has been applied locally. The creation of an explicit migration label prevents the client to wait for a potentially large number of false dependencies.

5. Label Propagation

In this section, we present the design of SATURN's metadata service, in charge of propagating labels among datacenters.

We start with an intuitive example that aims at introducing the tradeoffs involved in the design of SATURN and at highlighting the potential problems caused by false dependencies. We then define precisely the goals that SATURN should meet. Finally, we discuss how to configure SATURN in order to achieve those goals.

5.1 Rationale

The role of SATURN is to deliver, at each datacenter, in a serial order that is consistent with causality, the labels corresponding to the remote updates that need to be applied locally. Given that there may exist several serial orders matching a given partial causal order of events, the challenge is to select (for each datacenter) the "right" serial order that allows enhancing system's performance.

The reader may have noticed that, in many aspects, SATURN acts as a publish-subscribe system. Datacenters publish labels associated with updates that have been performed locally. Other datacenters, which replicate the item associated, subscribe to those labels. SATURN is in charge of delivering the published events (labels) to the interested subscribers. However, SATURN has a unique requirement that, to the best of our knowledge, has never been addressed by any previously designed publish-subscribe system: SATURN must mitigate the impact of false dependencies that are inevitably introduced when information regarding concurrency is lost in the serialization process. As we have seen, this loss of information is an unavoidable side effect of reducing the size of the metadata managed by the system.

In this section, we use a concrete example to convey the intuition of the tradeoffs involved in the design of SATURN to match the goal above. Consider the scenario depicted in Fig. 3. Here, we consider a scenario with four datacenters. Some items are replicated at dc_1 and dc_4 and some other items are replicated at dc_3 and dc_4. Let us assume that the bulk-data transfer from dc_1 to dc_4 has a latency of 10 units while the transfer from dc_3 to dc_4 has a latency of just 1 unit (this may happen if dc_3 and dc_4 are geographically close to each other and far away from dc_1). For clarity of exposition, let us assume that these delays are constant. There are three updates, a, b and c. For simplicity, assume that the timestamp assigned to these updates is derived from an external source of real time, occurring at time $t = 2$, $t = 4$ and $t = 6$ respectively. Let's also assume that $b \rightsquigarrow c$ and that a is concurrent with both b and c. The reader will notice that there are three distinct serializations of these updates that respect causal order: abc, bac, and bca. Which serialization should be provided to dc_4?

In order to answer this question, we first need to discuss how the operation of SATURN can negatively affect the performance of the system. For this, we introduce the following two concepts: *data readiness* and *dependency readiness*. Data readiness captures the ability of the system to provide the most recent updates to clients, as soon as its bulk-data transfer is completed. Dependency readiness captures the

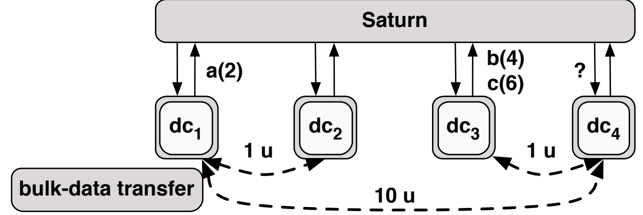


Figure 3: Label propagation scenario.

ability of the system to serve a request, because all of its causal dependencies (both real dependencies and false dependencies that are created as an artifact of the metadata compression) have been previously applied.

Considering data readiness alone, we would conclude that SATURN should deliver labels to remote datacenters as soon as possible. However, the reader may have noticed that there is no real advantage of delivering label a before instant $t = 12$, as the update can only be applied when the bulk-data transfer is completed. On the contrary, delivering label a very soon may create a false dependency that may affect the *dependency readiness* of other requests. Assume that SATURN opts to deliver to dc_4 the labels in the serial order abc. This is not only consistent with causality but also consistent with the real time occurrence of the updates. Unfortunately, this serialization creates a false dependency among update a and updates b and c, i.e., updates b and c need to be applied after update a as a result of the serialization procedure. This introduces unnecessary delays in the processing of the later updates: although update b and update c are delivered to dc_4 at times 5 and 7 respectively, they will have to wait until time 12 (while this was not strictly required by causality, given that a is concurrent with b and c). A more subtle consequence is that a correct but "inconvenient" serialization may increase the latency observed by clients. Assume that a client reads from dc_3 update b at time $t = 5$ and then migrates to dc_4 (to read some item that it is not replicated at dc_3). That client should be able to attach to dc_4 immediately, as by time $t = 5$ update b has been delivered to dc_4 and could be made visible. However, the false dependency introduced by the serialization above requires the client to wait until time $t = 12$ for the attachment to complete.

The example above shows that while trying to maximize data freshness by delivering labels not after the data-bulk transfer of its corresponding operations is completed, SATURN should avoid introducing false dependencies prematurely. A prematurely delivered label may unnecessarily delay the application of other remote operations, having a negative impact on the latency experienced by clients and increasing remote updates visibility latencies. Therefore, SATURN has to select a serialization per datacenter which does the best tradeoff between these two aspects. In the example above, if a is only delivered at dc_4 after b and c, by selecting the serialization bca, clients migrating from dc_3 to dc_4

would not be affected by the long latency of the bulk-data transfer link from dc_1 to dc_4; and a, b and c will become visible at dc_4 as soon as the bulk-data transfer is completed.

5.2 Selecting the Best Serializations

In order to precisely define which is the best serialization that should be provided to a given datacenter, we first need to introduce some terminology.

Let u_i be a given update i performed at some origin datacenter dc_o, and l_i the label for that update. Let t_i be the real time at which the update was created at the origin datacenter dc_o. Let dc_r be some other datacenter that replicates the data item that has been updated. Let $\Delta(dc_o, dc_r)$ be the expected delay of the bulk-data transfer from the origin datacenter to the replica datacenter. For simplicity of notation we assume that $\Delta(d_o, d_s) = 0$ if datacenter s does not replicate the item, and therefore, it is not interested in receiving the update. The expected availability time for the update at dc_r would be $t_i + \Delta(dc_o, dc_r)$. Finally let $\mathcal{H}(u_i) = \{u_j, u_k, ...\}$ the set of past updates that are in the causal past of u_i.

DEFINITION 1 (Optimal visibility time). *We then define the* optimal visibility time*, denoted vt_i^r of an update i at some replica dc_r, as the earliest expected time at which that update can be applied to dc_r. The optimal visibility time of an update at a target datacenter dc_r is given by:*

$$vt_i^r = max(t_i + \Delta(dc_o, dc_r), \max_{u_x \in \mathcal{H}(u_i)} vt_x^r)$$

From the example above it is clear that if the label l_i is delivered at dc_r after vt_i^r data freshness may be compromised. If l_i is delivered at dc_r before vt_i^r, delays in other requests may be induced due to false dependencies and lack of dependency readiness. Thus, SATURN should—ideally—provide to each datacenter dc_r a serialization that allows each label l_i to be delivered exactly at vt_i^r on that datacenter.

5.3 Architecture of SATURN's Metadata Service

SATURN is implemented by a set of cooperating servers, namely *serializers*, in charge of aggregating and propagating the streams of labels collected at each datacenter. We recall that our main goal is to provide to each datacenter a serialization of labels that is consistent with causality. This can be obtained by ensuring that *serializers* and datacenters are organized in a tree topology (with datacenters acting as leaves), connected with FIFO channels, and that *serializers* forward labels in the same order it receives them.

Let us illustrate its principles with the simplest example. Consider for instance a scenario with 3 datacenters dc_1, dc_2 and dc_3 connected to a single *serializer* S_1 in a star network. Consider a data item that is replicated in all datacenters and two causally dependent updates to that item, a and b ($a \rightsquigarrow b$), where a is performed at dc_1 and b is performed at dc_2 (both updates need to be applied at dc_3). In this scenario, the following sequence of events would be generated: a is applied at dc_1 and its label is propagated to S_1. In turn, S_1 propagates the label to dc_2 which, after receiving its payload (via the bulk-data transfer), applies update a locally. Following, at dc_2, some local client reads a and issues update b. The label associated with b is sent to S_1, that in turn will forward it to the other datacenters. Since *serializers* propagate labels preserving arrival order, datacenter dc_3 will necessarily receive a before b, as S_1 observes, independently of arbitrary network delays, a before b.[1]

Although a star network, with a single server, will trivially satisfy causality, such network may offer sub-optimal performance. In the previous section, we have seen that SATURN must deliver a label to a datacenter approximately at the same time the associated bulk data is delivered; for this requirement to be met, the metadata path cannot be substantially longer than the bulk data path. In fact, even if labels are expected to be significantly smaller that data items (and therefore, can be propagated faster), their propagation is still impaired by the latency among the SATURN *serializers* and the datacenters. Consider again the example of Fig. 3: we want labels from dc_3 to reach dc_4 within 1 time unit, thus any servers on that metadata path must be located close to those datacenters. Similarly, we want labels from $dc_1$to reach dc_2 fast. These two requirements cannot be satisfied if a single server is used, as in most practical cases, a single server cannot be close to both geo-locations simultaneously.

To address the efficiency problem above we use multiple *serializers* distributed geographically. Note that the tree formed is shared by all datacenters, and labels are propagated along the shared tree using the source datacenter as the root (i.e., there is no central root for all datacenters). This ensures that we can establish fast metadata paths between datacenters that are close to each other and that replicate the same data. Resorting to a network of cooperative *serializers* has another advantage: labels regarding a given item do not need to be propagated to branches of the tree that contain *serializers* connected to datacenters that do not replicate that item. This fact enables genuine partial replication and prevents all *serializers* from processing all labels, contributing to the scalability of the system.

Finally, since we expect labels to be disseminated faster that their correspondent bulkier payloads, it may happen that labels become available for delivery before their optimal visibility time. In fact, in current systems, and for efficiency reasons, bulk data is not necessarily sent through the shortest path [34]. Thus, for optimal performance, SATURN may introduce artificial delays in the metadata propagation, as discussed below.

[1] One can easily derive a correctness proof for any tree topology based on the idea that for any two causally related updates a and b ($a \rightsquigarrow b$) such that b was generated at dc_i (this implies that a was visible at dc_i before b was generated), the lowest common ancestor *serializer* between dc_i and any other datacenter interested in both updates, observes a label before b label. We do not include this proof due to lack of space.

5.4 Configuring SATURN's Metadata Service

The quality of the serialization served by SATURN to each datacenter depends on how the service is configured. A SATURN's configuration defines: (i) the number of *serializers* to use and where to place them; (ii) how these *serializers* are connected (among each other and with datacenters); and (iii) what delays (if any) should a *serializer* artificially add when propagating labels (in order to match the optimal visibility time). Let δ_{ij} denote the artificial delay added by *serializer* i when propagating metadata to *serializer* j.

In practice, when deploying SATURN, one has not complete freedom to select the geo-location of *serializers*. Instead, the list of potential locations for *serializers* is limited by the availability of suitable points-of-presence that results from business constraints. Therefore, the task of setting-up a *serializer* network is based on:

- The set V of datacenters that need to be connected (we denote $N = |V|$ the total number of datacenters).
- The latencies of the bulk data transfer service among these datacenters; lat_{ij} denotes the latency between datacenters i and j.
- The set W of potential locations for placing *serializers* ($M = |W|$). Since each datacenter is a natural potential *serializer* location, $M \geq N$. Let d_{ij} denote the latency between two *serializer* locations i and j.

However, given a limited set of potential locations to place *serializers*, it is unlikely (impossible in most cases) to match the optimal label propagation latency for every pair of datacenters. Therefore, the best we can aim when setting-up SATURN is to minimize the mismatch among the achievable label propagation latency and the optimal label propagation latency. More precisely, consider that the path for a given topology between two datacenters, i and j, denoted $P^M_{i,j}$ is composed by a set of *serializers* $P^M_{i,j} = \{S_k, ..., S_o\}$, where S_k connects to datacenter i and S_o connects to datacenter j. The latency of this path $\Delta^M(i,j)$ is defined by the latencies (d) between adjacent nodes in the path, plus any artificial delays that may be added at each step, i.e.:

$$\Delta^M(i,j) = \sum_{S_k \in P^M_{i,j} \setminus \{S_o\}} (d_{k,k+1} + \delta_{k,k+1})$$

and the mismatch between the resulting latency and the optimal label propagation latency is given by:

$$\mathit{mismatch}_{i,j} = |\Delta^M(i,j) - \Delta(i,j)|$$

Finally, one can observe that in general, the distribution of client requests, among items and datacenters may not be uniform, i.e., some items and some datacenters may be more accessed than others. As a result, a mismatch that affects the data visibility of a highly accessed item may have a more negative effect on the user experience than a mismatch on a seldom accessed item. Therefore, in the scenario where it is possible to collect statistics regarding which items and datacenters are more used, it is possible to assign a *weight* $c_{i,j}$ to each metadata path $P^M_{i,j}$, that reflects the relative importance of that path for the business goals of the application. Using these weights, we can now define precisely an optimization criteria that should be followed when setting up the *serializers* topology:

DEFINITION 2 (Weighted Minimal Mismatch). *The configuration that better approximates the optimal visibility time for data updates, considering the relative relevance of each type of update, is the one that minimizes the weighted global mismatch, defined as:*

$$min \sum_{\forall i,j \in V} c_{i,j} \cdot \mathit{mismatch}_{i,j}$$

5.5 Configuration Generator

The problem of finding a configuration that minimizes the Weighted Minimal Mismatch criteria, among all possible configurations that satisfy the constraints of the problem, is NP-hard.[2] Therefore, we have designed a heuristic that approximates the optimal solution using a constraint solver as a building block. We have modeled the minimization problem captures by Definition 2 as a constraint problem such that for a given tree, finds the optimal location of *serializers* (for a given set of possible location candidates) and the optimal (if any) propagation delays.

The proposed algorithm, depicted in Alg. 3, works as follows. Iteratively, starting with a full binary tree with only two leaves (Alg. 3, line 3), generates all possible isomorphism classes of full binary trees with N labeled leaves (i.e., datacenters). The algorithm adds one labeled leaf (datacenter) at each iteration until the number of leaves is equal to the total number of datacenters. For a given full binary tree T of f leaves, there exist $2 * f - 1$ isomorphic classes of full binary trees with $f + 1$ leaves. One can obtain a new isomorphic class by either inserting a new internal node within an edge of T from which the new leaf hangs (Alg. 3, line 14), or by creating a new root from which the new leaf and T hang (Alg. 3, line 10). We could iterate until generating all possible trees of N leaves. Nevertheless, in order to avoid a combinatorial explosion (for nine datacenters there would already be 2,027,025 possible trees), the algorithm selects at each iteration the most promising trees and discards the rest. In order to rank the trees at each iteration, we use the constraint solver. Therefore, given a totally ordered list of ranked trees, if the difference between the rankings of two consecutive trees T_1 and T_2 is greater than a given threshold, T_2 and all following trees are discarded (Alg. 3, line 18). At the last iteration, among all trees with N leaves, we pick the one that produces the smallest global mismatch from the optimal visibility times by relying on the constraint solver.

Note that Algorithm 3 always returns a binary tree. Nevertheless, SATURN does not require the tree to be binary. One can easily fuse two *serializers* into one if both are directly connected, placed in the same location, and the artifical propagation delays among them are zero. Any of these

[2] A reduction from the Steiner tree problem [35] can be used to prove this.

Algorithm 3 Find the best configuration.

```
 1: function FIND_CONFIGURATION(V, Threshold)
 2:     ⟨First, Second⟩ ← PICK_TWO(V)
 3:     InitTree = rooted tree with First and Second as leaves
 4:     Trees ← {InitTree}
 5:     V ← V \ {First, Second}
 6:     while V ≠ ∅ do
 7:         NextDC ← HEAD(V)
 8:         NewTrees ← ∅                                  ▷ ordered set
 9:         for all Tree ∈ Trees do
10:             NTree ← NEW_ROOTED(NextDC, Tree)
11:             NTree.ranking ← SOLVE(NTree)
12:             NewTrees ← NewTrees ∪ {NTree}
13:             for all Edge ∈ Tree do
14:                 NTree ← NEW_TREE(NextDC, Tree, Edge)
15:                 NTree.ranking ← SOLVE(NTree)
16:                 NewTrees ← NewTrees ∪ {NTree}
17:         V ← V \ {NextDC}
18:         Trees ← FILTER(Threshold, NewTrees)
19:     return HEAD(Trees)
```

fusions would cause the tree to change its shape without reducing its effectiveness.

6. Fault-tolerance and Adaptability

6.1 Fault-tolerance

In the following discussion, we focus on the failures that may disrupt SATURN's operation. We disregard failures in datacenters, as the problem of making data services fault-tolerant has been widely studied and is orthogonal to the contributions of this paper. Although SATURN's metadata service is instrumental to improve the global system performance (in particular, to speedup update visibility and client migration), it is never an impairment to preserve data availability. The fact that the global total order of labels defined by timestamps respects causality makes SATURN robust to failures. Thus, even if the metadata service suffers a transient outage, and stops delivering labels, updates can be still applied based on the timestamp order (we recall that labels are also piggybacked in the updates delivered by the bulk-data service).

A transient outage may be caused by a *serializer* failure or a network partition among *serializers*. Both situations lead to a disconnection in the *serializers* tree topology, possibly preventing the metadata service from delivering each label to all interested datacenters. Failures in *serializers* can be tolerated using standard replication techniques. Our current implementation assumes a fail-stop fault model [47], as *serializers* are made resilient to failures by replicating them using chain replication [51]. Nevertheless, SATURN's design does not preclude the use of other techniques [20, 48] in order to weaken the fault assumptions that we have made when building the current prototype. Connectivity problems in the tree may be solved by switching to a different tree, using the online reconfiguration procedure described next.

6.2 On-line Changes in the Configuration

Configuring SATURN is an offline procedure performed before the system starts operating. Substantial changes in the workload characterization may require changes in the *serializers* tree topology. A change to a new tree may be also required if connectivity issues affect the current tree (backup trees may be pre-computed to speedup the reconfiguration).

We have implemented a simple mechanism to switch among configurations without interrupting SATURN's operation. Let C_1 denote the configuration currently being used. Let C_2 denote the tree configuration to which we have decided to switch. SATURN switches configurations as follows:

- All datacenters input a special label, namely *epoch* change, in the system through the C_1 tree.
- At each datacenter, labels produced after the epoch change label are sent via the C_2 tree.
- A datacenter can start applying labels arriving from the C_2 tree as soon as it has received the *epoch* change label for every datacenter and all previously received labels delivered by the C_1 tree have been applied locally.
- During the transition phase, labels delivered by the C_2 tree are buffered until the *epoch* change is completed.

This mechanism provides fast reconfigurations, namely, in the order of the larger latency among the metadata paths in C_1 (in our experiments, always less than *200ms*).

When reconfiguring because C_1 has failed, or if C_1 breaks during the reconfiguration, the following (slower) switching protocol is used:

- During the transition phase, updates are delivered in timestamp order and labels delivered by the C_2 tree are buffered.
- A datacenter can start applying labels arriving from the C_2 tree as soon as the update associated with the first label delivered by C_2 is stable in timestamp order.

In this case, the reconfiguration time is bound by the time it takes to stabilize updates by timestamp order.

7. Evaluation

Our primary goal is to determine if SATURN, unlike previous work (see §7.3.1), can simultaneously optimize throughput and remote update visibility latency under both full and partial geo-replication. For this, we run SATURN and other relevant competing solutions under:

(i) Synthetic workloads that allow us to explore how the different parameters that characterize a workload impact the performance (§7.3); and

(ii) A benchmark based on the Facebook's dataset and access patterns, to obtain an assessment of SATURN under complex realistic workloads (§7.4).

Our secondary goal is to understand the impact of SATURN's configuration. For this, we first experiment with alternative configurations of SATURN (§7.1). We then study the impact of latency variability in SATURN (§7.2).

	NC	O	I	F	T	S
NV	*37 ms*	*49 ms*	*41 ms*	*45 ms*	*73 ms*	*115 ms*
NC	-	*10 ms*	*74 ms*	*84 ms*	*52 ms*	*79 ms*
O	-	-	*69 ms*	*79 ms*	*45 ms*	*81 ms*
I	-	-	-	*10 ms*	*107 ms*	*154 ms*
F	-	-	-	-	*118 ms*	*161 ms*
T	-	-	-	-	-	*52 ms*

Table 1: Average latencies (half RTT) among Amazon EC2 regions: N. Virginia (NV), N. California (NC), Oregon (O), Ireland (I), Frankfurt (F), Tokyo (T), and Sydney (S)

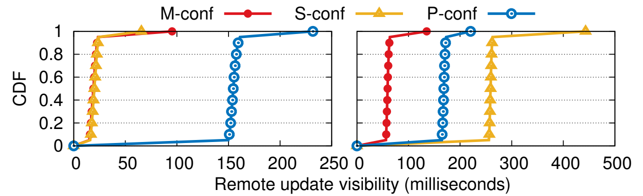


Figure 4: Left: Ireland to Frankfurt (*10ms*); Right: Tokyo to Sydney (*52ms*)

In order to compare SATURN with other solutions from the state-of-the-art (data services), we attached SATURN to an eventually consistent geo-replicated storage system we have built. Throughout the evaluation, we use this eventually consistent data service as the baseline, as it adds no overheads due to consistency management, to better understand the overheads introduced by SATURN. Note that this baseline represents a throughput upper-bound and a latency lower-bound. Thus, when we refer to the optimal visibility latency throughout the experiments, we are referring to the latencies provided by the eventually consistent system.

Implementation. Our SATURN prototype implements all functionality described in the paper. It has been built using the Erlang/OTP programming language. In our prototype, gears rely on physical clocks to generate monotonically increasing timestamps. To balance the load among frontends at each datacenter, we use Riak Core [14], an open source distribution platform. The optimization problem (Definition 2) used to configure SATURN is modeled using OscaR [44], a Scala toolkit for solving Operations Research problems.

Setup. We use Amazon EC2 `m4.large` instances running Ubuntu 12.04 in our experiments. Each instance has two virtual CPU cores, and 8 GB of memory. We use seven different regions in our experiments. Table 1 lists the average latencies we measured among regions. Our experiments simulate one datacenter per region. Clients are co-located with their preferred datacenter in separate machines. Each client machine runs its own instance of a custom version of Basho Bench [13], a load-generator and benchmarking tool. Each client eagerly sends requests to its preferred datacenter with zero thinking time. We deploy as many clients as necessary in order to reach the system's maximum capacity, without overloading it. Each experiment runs for more than 5 minutes. In our results, the first and the last minute of each experiment are ignored to avoid experimental artifacts. We measure the visibility latencies of remote update operations by storing the physical time at the origin datacenter when the update is applied locally, and subtracting it from the physical time at the destination datacenter when the update becomes visible. To reduce the errors due to clock skew, physical clocks are synchronized using the NTP protocol [43] before each experiment, making the remaining clock skew negligible in comparison to inter-datacenter travel time.

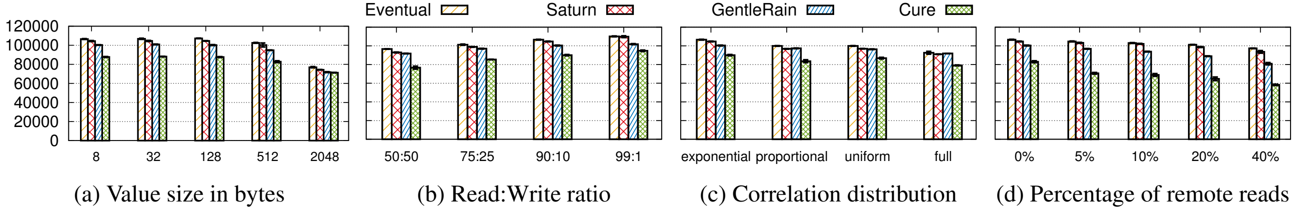
7.1 SATURN Configuration Matters

We compare different configurations of SATURN to better understand their impact on the system performance. We compare three alternative configurations: (i) a single-*serializer* configuration (S), (ii) a multi-*serializer* configuration (M), and (iii) a peer-to-peer version of SATURN that relies on the conservative label's timestamp order to apply remote operations (P). We focus on the visibility latencies provided by the different implementations.

For the S-configuration, we placed the *serializer* in Ireland. For the M-configuration, we build the configuration tree by relying on Algorithm 5.4. We run an experiment with a read dominant workload (90% reads). Figure 4 shows the cumulative distribution of the latency before updates originating in Ireland become visible in Frankfurt (left plot) and before updates originating in Tokyo become visible in Sydney (right plot). Results show that both the S and M configurations provide comparable results for updates being replicated in Frankfurt. This is because we placed the *serializer* of the S-configuration in Ireland, and therefore, the propagation of labels is done efficiently among these two regions. Unsurprisingly, when measuring visibility latencies before updates originating in Tokyo become visible in Sydney, the S-configuration performs poorly because labels have to travel from Tokyo to Ireland and then from Ireland to Sydney. Plus, results show that the P-configuration, that relies on the label's timestamp order, is not able to provide low visibility latencies in these settings. This is expected as, when tracking causality with a single scalar, latencies tend to match the longest network travel time (161ms in this case) due to false dependencies. In turn, the M-configuration is able to provide significantly lower visibility latencies to all locations (deviating only 8.2ms from the optimal on average).

7.2 Impact of Latency Variability on SATURN

The goal of this section is to better understand how changes in the link latency affect SATURN's performance. We have just seen that the correct configuration of the *serializers*' tree has an impact on performance. Therefore, if changes in the link latencies are large enough to make the current configuration no longer suitable, and these changes are permanent, a reconfiguration of SATURN should be triggered. In practice, transient changes in the link latencies are unlikely to



(a) Value size in bytes (b) Read:Write ratio (c) Correlation distribution (d) Percentage of remote reads

Figure 5: Dynamic workload throughput experiments.

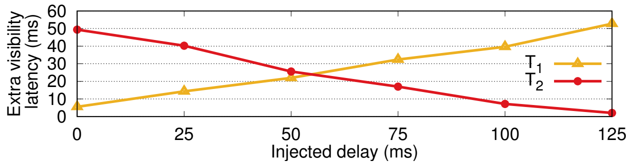


Figure 6: Impact of latency variability on remote update visibility in SATURN.

justify a reconfiguration; therefore we expect their effect on performance to be small.

To validate this assumption, we set a simple experiment with three datacenters, each located in a different EC2 region: N. Carolina, Oregon and Ireland. For the experiment, we artificially inject extra latency between N. Carolina and Oregon datacenters (averaged measured latency is 10ms). From our experience, we expect the latency among EC2 regions to deviate from its average only slightly and transiently. Nevertheless, to fully understand the consequences of latency variability, we also experimented with unrealistically large deviations (up to 125ms).

Figure 6 shows the extra remote visibility latency that two different configurations of SATURN add on average when compared to an eventually consistent storage system which makes no attempt to enforce causality. Both configurations, T_1 and T_2, use a single *serializer*: configuration T_1 places the *serializer* in Oregon, representing the optimal configuration under normal conditions and configuration T_2, instead, places the *serializer* in Ireland.

As expected, under normal conditions, T_1 performs significantly better than T_2, confirming the importance of choosing the right configuration. As we add extra latency, T_1 degrades its performance, but only slightly. One can observe that, in fact, slight deviations in the averaged latency have no significant impact in SATURN: even with an extra delay of 25ms (more than twice the average delay), T_1 only adds 14ms of extra visibility latency on average. Interestingly, it is only with more than 55ms of injected latency that T_2 becomes the optimal configuration, exhibiting lower remote visibility latency than T_1. Observing a long and sustained increase of 55ms of delay on a link that averages 10ms is highly unlikely. Indeed, this scenario has the same effect of migrating the datacenter from N. Carolina to São Paulo. Plus, if such large deviation becomes the norm, system operators can always rely on SATURN's reconfiguration mechanism to change SATURN configuration.

7.3 SATURN vs. the State-of-the-art

We compare the performance of SATURN against eventual consistency and against the most performant causally consistent storage systems in the state-of-the-art.

7.3.1 GentleRain and Cure

We consider GentleRain [26] and Cure [3] the current state-of-the-art. We have also experimented with solutions based on explicit dependency checking such as COPS [39] and Eiger [40]. Nevertheless, we concluded that approaches based on explicit dependency checking are not practical under partial geo-replication. Their practicability depends on the capability of pruning client's list of dependencies after update operations due to the transitivity rule of causality [39]. Under partial geo-replication, this is not possible, causing client's list of dependencies to potentially grow up to the entire database.

At its core, both GentleRain and Cure implement causal consistency very similarly: they rely on a background *stabilization mechanism* that requires all partitions in the system to periodically exchange metadata. This equips each partition with sufficient information to locally decide when remote updates can be safely–with no violation of causality—made visible to local clients. In our experiments, GentleRain and Cure's stabilization mechanisms run every 5ms following the authors' specifications. The interested reader can find more details in the original papers [3, 26]. We recall that SATURN does not require such a mechanism, as the order in which labels are delivered to each datacenter already determines the order in which remote updates have to be applied.

The main difference between GentleRain and Cure resides in the way causal consistency is tracked. While GentleRain summarizes causal dependencies in a single scalar, Cure uses a vector clock with an entry per datacenter. This enables Cure to track causality more precisely—lowering remote visibility latency—but the metadata management increases the computation and storage overhead—harming throughput. Concretely, by relying on a vector, Cure remote update visibility latency lower-bound is determined by the latency between the originator of the update and the remote datacenter. Differently, in GentleRain, the lower-bound is

determined by the latency to the furthest datacenter regardless of the originator of the update [3, 26, 32].

7.3.2 Throughput Experiments

In the following set of experiments, we aim at understanding how different parameters of the workload characterisation may impact SATURN's throughput in comparison to state-of-the-art solutions. We explore the workload space varying a single parameter, setting the others to a fixed value. We play with multiple aspects (default values within the parenthesis): values size (2B), read/write ratio (9:1), the correlation among datacenters (exponential), and the percentage of remote reads (0%). Figure 5 shows the results.

Value Size. We vary the size of values from 8B up to 2048B. Sizes have been chosen based on the measurement study discussed in the work of Armstrong et al. [7]. Results (Figure 5a) show that all solutions remain unaffected up to medium size values (128B). Nevertheless, as we increase the value size up to 2048B, solutions exhibit, as expected, similar behavior handling almost the same amount of operations per second. This shows that, with large value sizes, the performance overhead introduced by GentleRain and, above all, Cure, is masked by the overhead introduced due to the extra amount of data being handled.

R/W Ratio. We vary the read/write ratio from a read dominant workload (99% reads) to a balanced workload (50% reads). Results (Figure 5b) show that solutions are similarly penalized as the number of write operations increases.

Correlation. We define the correlation between two datacenters, as the amount of data shared among them. The correlation determines the amount of traffic generated in SATURN due to the replication of update operations. We define four patterns of correlation: *exponential*, *proportional*, *uniform*, and *full*. The exponential and proportional patterns fix the correlation between the datacenters based on their distance. Thus, two datacententers closely located (e.g., Ireland and Frankfurt) have more common interests than distant datacenters (e.g., Ireland and Sydney). The exponential pattern represents a more prominent partial geo-replicated scenario by defining very low correlation among distant datacenters. The proportional pattern captures a smoother distribution. The uniform pattern defines an equal correlation among all datacenters. Lastly, the full pattern captures a fully geo-replicated setting. Results show that the more prominent the partial geo-replication scenario is, the better results SATURN presents when compared to GentleRain and Cure, that are required to send heartbeats constantly, adding an overhead when compared to SATURN. Interestingly, even in the full geo-replicated scenario, the best case scenario for GentleRain and Cure, SATURN still provides a throughput comparable to GentleRain and significantly outperforms Cure (15.2% increase).

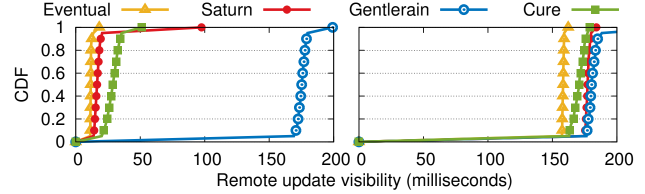


Figure 7: Left: Ireland to Frankfurt (*10ms*); Right: Ireland to Sydney (*154ms*)

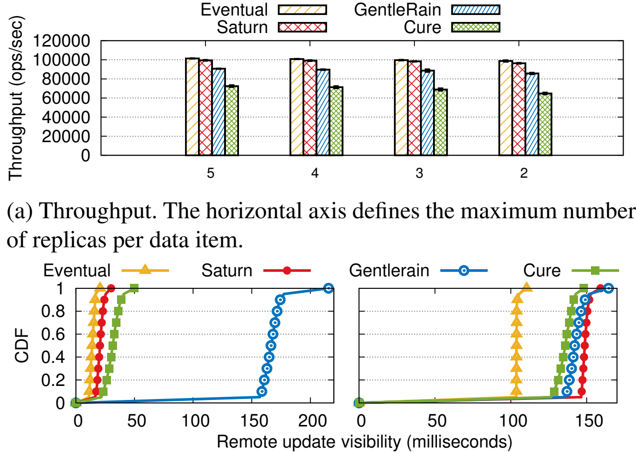
Remote Reads. We vary the percentage of remote reads from 0% up to 40% of the total number of reads. Interestingly, results show (Figure 5d) that GentleRain and, above all, Cure are significantly more disrupted than SATURN by remote reads. To better understand the cause of this behavior, we need to explain how remote reads are managed in our experiments by GentleRain and Cure. As in SATURN, a client requiring to read from a remote datacenter first needs to attach to it. An attach request is performed by providing the latest timestamp observed by that client (a scalar in GentleRain and a vector in Cure). The receiving datacenter only returns to the client when the stable time—computed by the stabilization mechanism—is equal or larger than client's timestamp. This significantly slows down clients. Results show that with 40% of remote reads, SATURN outperforms GentleRain by 15.7% and Cure by 60.5%.

We can conclude that SATURN exhibits a performance comparable to an eventually consistent system (2.2% of averaged overhead) while showing a slightly better throughput than GentleRain (4.8% average) and significantly better than Cure (24.7% on average). Cure overhead is dominated by the managing of a vector for tracking causality instead of a scalar as in GentleRain and SATURN. Regarding GentleRain, SATURN exhibits slightly higher throughput due to the overhead caused by GentleRain's stabilization mechanism.

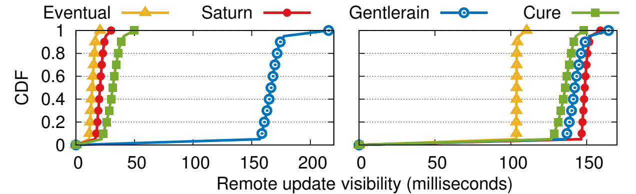
7.3.3 Visibility Latency Experiments

In the following experiment, we measure the visibility latency provided by each of the systems. We expect SATURN to exhibit lower visibility latencies on average than both Cure and GentleRain. We expect to slightly outperform Cure as we avoid the costs incurred by Cure's stabilization mechanism. We expect to significantly outperform GentleRain since it does not mitigate false dependencies and their latencies should theoretically tend to match the longest travel time among datacenters.

In addition to measuring the averaged visibility latencies provided by each solution, we analyze both the best and the worst case for SATURN. Given that *serializers* possible locations are limited, labels traversing the whole tree are likely to be delivered with some extra undesired delay. Concretely, in the following experiment, the major deviation from the optimal latencies is produced by the path connecting Ireland and Sydney (extra 20ms).



(a) Throughput. The horizontal axis defines the maximum number of replicas per data item.



(b) Left: Ireland to Frankfurt (*10ms*); Right: Ireland to Tokyo (*107ms*).

Figure 8: Facebook-based benchmark results.

For these experiments, we have used the default values defined in §7.3. Results show that SATURN only increases visibility latencies by 7.3ms on average when compared to the optimal, outperforming GentleRain and Cure that add 97.9ms and 21.3ms on average respectively. Figure 7 shows the cumulative distribution of the latency before updates originating in Ireland become visible in Frankfurt (left plot) and Sydney (right plot). The former represents the best case scenario with no extra delay imposed by the tree. The latter represents the worst case scenario. Figure 7 shows that SATURN almost matches the optimal visibility latencies in the best case scenario (only 7ms of extra delay in the 90th percentile) and, as expected, adds an extra of 20.4ms (90th percentile) in the worst case. Results also show that SATURN is able to provide better visibility latencies than both GentleRain and Cure in the best case and to GentleRain in the worst case. As expected, GentleRain tends to provide visibility latencies equal to the longest network travel time, which in this case is between Frankfurt and Sydney. Interestingly, in SATURN's worst case, Cure only serves slightly lower latencies (3.6ms less in the 90th percentile). Although the metadata used by Cure to track causality theoretically allows them to make visible remote updates in optimal time, in practice, the stabilization mechanism results in a significant extra delay.

7.4 Facebook Benchmark

To obtain an assessment of SATURN's performance under complex realistic workloads, we experiment with a social networking workload we integrated into Basho Bench, our benchmarking tool. Our workload generator is based on the study of Benevenuto et al. [15]. The study defines a set of operations (e.g., browsing photo albums, sending a message, editing user settings among many others) with its corresponding percentage of occurrence. This serves us not only to characterize the workload in terms of the read/write ratio, but it also tells us whether an operation concerns user data (e.g., editing settings), a friend's data (e.g., browse friend updates), or even random user data (e.g., universal search), which relates to the number of remote operations.

We use a public Facebook dataset [52] of the New Orleans Facebook network, collected between December 2008 and January 2009. The dataset contains a total of 61096 nodes and 905565 edges. Each node represents a unique user, which acts as a client in our experiments. Edges define friendship relationships among users. In order to distribute and replicate the data among datacenters (seven in total), we have implemented the partitioning algorithm described in [46], augmented to limit the maximum number of replicas each partition may have, to avoid partitionings that rely extensively on full replication. As in [46], partitions are made to maximize the locality of data regarding a user and her friends, thus, minimize remote reads.

We examine the throughput of SATURN in comparison to an eventually consistent system, GentleRain, and Cure. Figure 8a shows the results of a set of experiments in which we fix the minimum of replicas to 2, and vary the maximum from 2 to 5, which indirectly varies the number of remote read operations. Results show that SATURN exhibits a throughput comparable to an eventually consistent system (only 1.8% of averaged overhead) and significantly better than GentleRain and Cure, handling 10.9% and 41.9% more operations per second on average respectively.

In a second experiment, we measure the remote update visibility latency exhibited by the solutions. SATURN increases visibility latencies by 16.1ms on average when compared to the optimal, outperforming GentleRain and Cure that add 79.2ms and 23.7ms on average respectively. In addition, we analyze the best and the worst case scenario for SATURN. Figure 8b shows the visibility latency of updates replicated from Ireland to Frankfurt (on the left) and Tokyo (on the right). The former represents the best case scenario for SATURN; the latter represents the worst. In the worst case scenario, SATURN introduces significant overheads when compared to the optimal (47.2ms in the 90th percentile). This is expected, as it has to traverse the whole tree. Nevertheless, it still exhibits a performance comparable to both GentleRain and Cure, only adding 0.9ms and 9.9ms respectively in the 90th percentile. Moreover, in the best case scenario, SATURN exhibits visibility latencies very close to the optimal (represented by the eventually consistent line), with only a difference of 8.7ms in the 90th percentile.

8. Related Work

Table 2 classifies the most relevant causally consistent systems previously proposed in the literature. In the following discussion, we focus on how each system implements causal consistency across datacenters and on the tradeoffs involved. We classify the systems based on three categories:

	Key Technique	Metadata Structure	Partial Replication
Bayou [45, 50]	*sequencer-based*	*scalar*	*no*
Practi [24]	*sequencer-based*	*scalar*	*yes*
ISIS[3] [16]	*sequencer-based*	*vector[dcs]*	*no*
Lazy Repl. [36]	*sequencer-based*	*vector[dcs]*	*no*
SwiftCloud [53]	*sequencer-based*	*vector[dcs]*	*no*[4]
ChainReaction [5]	*sequencer-based*	*vector[dcs]*	*no*
COPS [39]	*explicit check*	*vector[keys]*	*no*
Eiger [40]	*explicit check*	*vector[keys]*	*no*
Bolt-on [10]	*explicit check*	*vector[keys]*	*no*
Orbe [25]	*explicit check*	*vector[servers]*	*no*
GentleRain [26]	*global stabilization*	*scalar*	*no*
Cure [3]	*global stabilization*	*vector[dcs]*	*no*
Saturn	*tree-based dissem.*	*scalar*	*yes*

Table 2: Summary of causally consistent systems.

the *key technique* behind their implementation of causal consistency, the *metadata* used to identify causal dependencies, and the ability to support *partial replication*.

Sequencer-based approaches [5, 16, 24, 36, 45, 50, 53] simplify the implementation of causal consistency at the cost of limiting intra-datacenter parallelism. They rely on a per-datacenter centralized authority, commonly called sequencer, to order local updates. This allows them to effortlessly aggregate metadata and thus avoid metadata explosion. Nevertheless, the sequencer is contacted before the request is processed, limiting datacenter capacity. Other approaches avoid sequencers while tracking dependencies more precisely [10, 25, 39, 40]. Unfortunately, these systems may generate a very large amount of metadata, incurring a significant overhead due to its management costs [9, 26].

From our perspective, the most scalable and performant solutions of the literature are GentleRain [26] and Cure [3]. These systems rely on a background stabilization mechanism that runs periodically and that introduces a tradeoff among overhead and visibility latencies.

Interestingly, all solutions exchange metadata directly among datacenters, piggybacked with the data payload. Thus, the order of updates must be inferred exclusively from the metadata (unlike in SATURN, where metadata is served in the correct order). Thus, on one hand, when metadata is aggregated, such as in [24, 26, 45], false dependencies induce poor remote visibilities compared to systems tracking causality more precisely [3, 16, 25, 36, 39, 40]. On the other hand, when metadata is not aggregated, the associated computation and storage overhead reduces system's throughput.

As §7 demonstrates, SATURN operates on a sweet-spot among these approaches. It relies on a single scalar to track causal dependencies, incurring a negligible computational and storage overhead. Furthermore, unlike systems relying on a single scalar, SATURN mitigates the impact of false dependencies by using a tree-based metadata dissemination. This allows SATURN to achieve significantly lower visibility latencies. Finally, SATURN is optimized for partial geo-replication. Differently to previous solutions, it enables genuine partial replication, requiring datacenters to only manage the data and metadata of the items replicated locally.

Finally, two systems that do not fit in previous classification but are related to SATURN: Kronos [27] and Occult [42].

Kronos is a generic service that allows to precisely track any partial order, avoiding false dependencies. Kronos flexibility comes at the cost of a centralized implementation which, in geo-replicated settings, forces clients to pay the cost of a potentially large roundtrip to use the service. The ideas behind the design of SATURN may be used to derive an efficient distributed implementation of Kronos.

Occult has been developed concurrently with SATURN. Interestingly, it allows updates to be applied immediately, in an order that may violate causality. Nevertheless, causality is enforced when executing read operations, based on metadata that is kept by clients. Occult aims at avoiding slowdown cascades [2]. In order to mitigate the problem of false dependencies, while avoiding the metadata size to became very large, Occult is required to enforce a number of constraints on the system operation; for instance, updates can only be performed on a master replica. SATURN does not impose such constraints and allows clients to maintain and manage much less metadata. The combination of Occult and SATURN is an interesting research avenue that has the potential to address the metadata size, the impact of false dependencies, and also slowdown cascades.

9. Conclusions

We have presented SATURN, a distributed metadata service for efficiently implementing causal consistency under both full and partial geo-replication. We have shown that SATURN, when attached to a eventually geo-replicated storage system, exhibits throughput comparable (only 2% overhead on average) to systems providing almost no consistency guarantees. At the same time, SATURN mitigates the impact of false dependencies—unavoidably introduced when compressing metadata—by relying on a metadata dissemination service that can be configured to match optimal data visibility latencies. Finally, SATURN supports genuine partial replication, which makes it suitable for partial geo-replication.

Acknowledgments

We thank our shepherd Neeraj Suri, Marc Shapiro, Pierre Schaus, Vasco Manquinho, Rodrigo Rodrigues, Alejandro Tomsic, Nuno Preguiça, Christopher Meiklejohn, and the anonymous reviewers for their comments and suggestions. This work has been partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects PTDC/ EEI-SCR/ 1741/ 2014 (Abyss) and UID/ CEC/ 50021/ 2013; the FP7 project 609 551 SyncFree; and the Horizon 2020 project 732 505 LightKone.

[3] ISIS can support partial geo-replication by extending its metadata to multiple *vector[dcs]* (one per communication group; a maximum of $2^{dcs} - 1$).

[4] SwiftCloud supports partial replication at the client-side, a challenge that has not been addressed by SATURN.

References

[1] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[2] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *Proceeding of the 15th Workshop on Hot Topics in Operating Systems*, HotOS '15, 2015.

[3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguia, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceeding of the IEEE 36th International Conference on Distributed Computing Systems*, ICDCS '16, pages 405–414, 2016.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, SIGCOMM '08, pages 63–74, Seattle, WA, USA, 2008.

[5] S. Almeida, J. a. Leitão, and L. Rodrigues. ChainReaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, Prague, Czech Republic, 2013.

[6] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, Big Sky, Montana, USA, 2009.

[7] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1185–1196, New York, New York, USA, 2013.

[8] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 385–394, Donostia-San Sebastián, Spain, 2015.

[9] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, pages 22:1–22:7, San Jose, California, 2012.

[10] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, New York, USA, 2013.

[11] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, Melbourne, Victoria, Australia, 2015.

[12] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the 10th European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, Bordeaux, France, 2015.

[13] Basho. Basho Bench. `http://github.com/basho/basho_bench`, .

[14] Basho. Riak core. `http://github.com/basho/riak_core`, .

[15] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 49–62, Chicago, Illinois, USA, 2009.

[16] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3), Aug. 1991.

[17] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. Rodrigues. On the use of clocks to enforce consistency in the cloud. *IEEE Data Engineering Bulleting*, 38(1):18–31, 2015.

[18] E. A. Brewer. Towards robust distributed systems. In *Keynote at the ACM Symposium on Principles of Distributed Computing*, PODC, 2000.

[19] A. Brodersen, S. Scellato, and M. Wattenhofer. Youtube around the world: Geographic popularity of videos. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 241–250, Lyon, France, 2012.

[20] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, New Orleans, Louisiana, USA, 1999.

[21] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1): 11–16, July 1991.

[22] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP '93, pages 44–57, Asheville, North Carolina, USA, 1993.

[23] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Hollywood, CA, USA, 2012.

[24] M. Dahlin, L. Gao, A. Nayate, A. Venkataramana, P. Yalagandula, and J. Zheng. Practi replication. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '06, 2006.

[25] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, Santa Clara, California, 2013.

[26] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks.

In *Proceedings of the 5th ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, Seattle, WA, USA, 2014.

[27] R. Escriva, A. Dubey, B. Wong, and E. G. Sirer. Kronos: The design and implementation of an event ordering service. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys '14, pages 3:1–3:14, Amsterdam, The Netherlands, 2014.

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

[29] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, SIGCOMM '09, pages 51–62, Barcelona, Spain, 2009.

[30] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1):297–316, 2001.

[31] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Trade-offs in replicated systems. *IEEE Data Engineering Bulletin*, 39: 14–26, 2016.

[32] C. Gunawardhana, M. Bravo, and L. Rodrigues. Unobtrusive deferred update stabilization for efficient geo-replication. *Arxiv preprint arXiv:1702.01786*, Feb. 2017.

[33] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[34] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM '13, pages 3–14, Hong Kong, China, 2013.

[35] R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[36] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), July 1978.

[38] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 265–278, 2012.

[39] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, Cascais, Portugal, 2011.

[40] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 313–328, 2013.

[41] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. *Technical Report TR-11-21, University of Texas at Austin, Austin, Texas*, 2011.

[42] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can't believe it's not causal! In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, 2017.

[43] NTP. The network time protocol. http://www.ntp.org.

[44] OscaR Team. OscaR: Scala in OR. https://bitbucket.org/oscarlib/oscar.

[45] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP '97, pages 288–301, Saint Malo, France, 1997.

[46] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM '10, pages 375–386, New Delhi, India, 2010.

[47] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, May 1984.

[48] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[49] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, Cascais, Portugal, 2011.

[50] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Sep 1994.

[51] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '04, 2004.

[52] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM Workshop on Online Social Networks*, WOSN '09, pages 37–42, Barcelona, Spain, 2009.

[53] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, Vancouver, BC, Canada, 2015.