Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication

(extended abstract of the MSc dissertation)

Chathuri Lanchana Rubasinghe Gunawardhana
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—Geo-replication is a requirement of most cloud applications. A fundamental problem that geo-replicated systems need to address is how to ensure that remote updates are applied and made visible to clients in a consistent order. For that purpose, both clients and servers are required to maintain some form of metadata. Unfortunately, there is a tradeoff between the amount of metadata a system needs to maintain and the amount of concurrency offered to local clients. Given the high costs of managing large amounts of metadata, many practical systems opt to serialise some updates, using some form of sequencer, which, as we will show, may significantly reduce the throughput of the system. In this paper we advocate an alternative approach that consists in allowing full concurrency when processing local updates and using a deferred *local* serialisation procedure, before shipping updates to remote datacenters. This strategy allows to implement inexpensive mechanisms to ensure system consistency requirements while avoiding intrusive effects on update operations, a major performance limitation. We have implemented and extensively evaluated our approach. Experimental data shows that we outperform sequencer-based approaches by almost an order of magnitude in the maximum achievable throughput. Furthermore, unlike proposed sequencer-free solutions, our approach reaches nearly optimal remote update visibility latencies without limiting throughput.**

I. Introduction

Geo-replication is a requirement for modern internet-based services. Unfortunately, due to the long network delays among geographically remote datacenters, synchronous replication (where an update is applied to all replicas before the operation returns to the client) is prohibitively slow for most practical purposes. Therefore, many approaches require some form of asynchronous replication strategy.

A fundamental problem associated with asynchronous replication is how to ensure that updates performed in remote datacenters are applied and made visible to clients in a consistent manner. Many different consistency guarantees that allow for asynchronous replication have been defined [1]. Among them, causal consistency [2] has been identified as the strongest consistency model an always-available system can implement [3], becoming of practical relevance in geo-replicated settings. In fact, most weak consistency criteria require that updates are causally ordered, and only differ among each other on when it is safe to make remote updates visible [4].

Not surprisingly, the metadata problem has been extensively studied in the literature [5], [6], [7], [8], [9] (in fact, most solutions have roots in the seminal works of [10], [11]). A common solution to reduce the cost of implementing causal consistency consists in serialising all updates that are executed at a given datacenter [7]. In this case the state observed by a client when accessing a datacenter can be deterministically identified by the sequence number of the last update that has been applied locally. Using this approach, it is possible to ensure that clients always observe values that are consistent with causality by maintaining the metadata as small as a vector clock, with one entry for each datacenter.

Unfortunately, serialising all updates that are applied to a given datacenter may severely limit the concurrency among independent updates. Even if different data items are stored in different physical machines, all updates are still synchronously ordered by some sequencer (operating in the critical path of local clients), typically executed by a small set of machines. Such serialisation may become a bottleneck in the system. Therefore, the implementation of efficient sequencers is, by itself, a relevant research topic. CORFU [12] is a prominent example of an efficient sequencer implementation for datacenters.

Recent systems such as GentleRain [8], and Cure [9] have proposed an alternative technique to circumvent the tradeoff between the metadata size and the concurrency allowed in the system. In these systems, updates are tagged with a small-sized timestamp value, that can either be a single scalar [8] or a vector clock [9]. When a remote update is received, it is hidden until one is sure that all updates with a smaller timestamp have been locally applied. For this, servers have to periodically run among them a *global stability checking* procedure. Interestingly, each of this stabilisation rounds generates N^2 of intra-datacenter messages and $N * M^2$ of inter-datacenter messages, where N is the number of partitions per datacenter and M is the total number of datacenters. Thus, in order to reduce the amount of messages that are exchanged among servers to avoid overloading them, one is forced to either limit the number of partitions per datacenter, throttling the concurrency inside a datacenter; or to reduce the frequency at which the stabilisation rounds occur, deteriorating the quality-of-service provided to clients.

In this paper, we advocate, implement, and evaluate a novel approach to address the metadata versus concurrency tradeoff in weakly causal consistent geo-replicated systems. Our approach has some similarities with the systems that rely on global stability checking but also significant differences. As with [8], [9], we let local updates proceed without any a priory synchronization. However, unlike previous systems, we totally order all updates, in a manner consistent with causality, before shipping updates to remote datacenters. As a result, expensive global stabilisation is avoided, as it is trivial for a datacenter to check whether all updates subsumed in the timestamps piggybacked by remote updates have been locally applied (similarly to sequencer-based solutions).

We have implemented our approach as a variant of the open source version of Riak KV [13]. We have augmented the system with a service that totally orders all the updates, before shipping them, that we have called *Eunomia*[1]. Our experimental results show that our system outperforms sequencer-based systems by almost an order of magnitude while serving significantly better quality-of-service to clients compared with systems based on *global stabilisation checking* procedures.

The contributions of this paper are the following:

- The introduction of *Eunomia*, a new service for unobtrusively ordering updates, and the techniques behind it (§II).
- A fault tolerant version of *Eunomia* (§II-C).
- Sound experimental comparison of the maximum load that traditional sequencers and the newly introduced *Eunomia* can handle, and their potential bottlenecks (§V-A).
- Integration of *Eunomia* into an always-available geo-replicated data store (§III) and its performance comparison to state-of-the-art solutions (§V-B).

II. *Eunomia*: Unobtrusive Ordering

In this section, we first describe *Eunomia* and the techniques behind it. Then, we show how *Eunomia* can effectively sequence the updates that have occurred concurrently in a given datacenter, hiding the complexity introduced by such concurrency from remote geo-locations. We finally discuss important aspects in the design of *Eunomia* and the protocols used in its implementation.

A. Overview

Even though sequencers can simplify the cause effect relationship among events, they can limit the concurrency of the system as they falls into the critical path of the clients. We propose *Eunomia*, a new service conceived to replace sequencers in these settings. *Eunomia* aims at simplifying the design of weakly consistent geo-replicated data stores without compromising intra-datacenter concurrency. Unlike traditional sequencers, *Eunomia* lets local client operations to execute without synchronous coordination, an essential characteristic to avoid limiting concurrency and increasing operation latencies. Then, in the background, *Eunomia* establishes a serialization of all updates occurring in the local datacenter in an order consistent with causality, based on timestamps generated locally by the individual servers that compose the datacenter. We refer to this process as *site stabilization procedure*. Thus, *Eunomia* is capable of abstracting the internal complexity of a multi-server datacenter without limiting the concurrency. Therefore, *Eunomia* can be used to improve any existing sequencer-based solution to enforce causal consistency across geo-locations [4], [7], as shown in §III.

[1]Greek goddess of law and legislation, her name can be translated as "good order".

N	Number of partitions
$Clock_c$	Client c clock
p_n	Partition n
$Clock_n$	Current physical time at p_n
Ops	Set of unstable operations at *Eunomia*
PartitionTime	Vector with an entry per partition at *Eunomia*
$u_j.ts$	Timestamp assigned to update u_j

Table I
Notation used in the protocol description.

B. Eunomia Into Play

In order to better illustrate how *Eunomia* works, we now present a detailed protocol of the interaction between *Eunomia* and nodes that constitute a datacenter. In the following exposition, we assume that updates to individual items are serialized by the native update protocol. In fact, we generalize this assumption for the case where multiple items are managed by a given set of servers that serialize all updates of those items. We call each of these set of servers a *partition*. We assume FIFO links among partitions and *Eunomia*. Table I provides a summary of the notation used in our protocols.

Eunomia assumes that each individual partition can assign a timestamp to each update without engaging in synchronous coordination with other partitions, or with *Eunomia* servers. We will explain below how this can be easily achieved. These timestamps have to satisfy two properties.

Property 1: If an update u_j causally depends on a second update u_i, then the timestamp assigned to u_j ($u_j.ts$) is strictly greater than $u_i.ts$.

Property 2: For two updates u_i and u_j received by *Eunomia* from partition p_n, if u_i is received before u_j then $u_j.ts$ is strictly greater than $u_i.ts$.

These two properties imply that updates are causally ordered across all partitions and that once *Eunomia* receives an update coming from a partition p_n, no update with smaller timestamp will be ever received from p_n.

In order to provide the above properties, clients play a fundamental role. A client c maintains during its session the largest clock seen ($Clock_c$). This clock aggregates its causal dependencies and it is included in the client's update

Algorithm 1 Operations at client c

```
1:  function READ(Key)
2:      send READ(Key) to server
3:      receive ⟨Value, Ts⟩ from server
4:      Clock_c ← MAX(Clock_c, Ts)
5:      return Value

6:  function UPDATE(Key, Value)
7:      send UPDATE(Key, Value, Clock_c) to server
8:      receive Ts from server
9:      Clock_c ← Ts
10:     return ok
```

Algorithm 2 Operations at partition p_n

```
1:  function READ(Key)
2:      ⟨Value, Ts⟩ ← KV_GET(Key)
3:      return ⟨Value, Ts⟩

4:  function UPDATE(Key, Value, Clock_c)
5:      MaxTs_n ← MAX(Clock_n, Clock_c + 1, MaxTs_n + 1)
6:      KV_PUT(Key, ⟨Value, MaxTs_n⟩)
7:      Op ← ⟨ Key, Value, MaxTs_n, p_n⟩
8:      send ADD_OP(Op) to Eunomia
9:      return MaxTs_n

10: function HEARTBEAT                          ▷ Every Δ time
11:     if Clock_n ≥ MaxTs_n + Δ then
12:         send HEARTBEAT(p_n, MaxTs_c) to Eunomia
```

requests. As described below, partitions compute update timestamps considering client clocks, which is key for the protocol's correctness.

The protocol assumes that each partition p_n is equipped with a physical clock. Clocks are loosely synchronized by a time synchronization protocol such as NTP [14]. The correctness of the protocol does not depend on the clock synchronization precision—namely on the clock drift. However, as discussed later, large clock drifts may have some negative impact on the protocol's performance (in particular, on how fast the datacenter can ship updates to other remote datacenters). To circumvent this limitation, our protocol uses hybrid clocks [15], which have been shown to overcome some of the limitations of simply using physical time. The use of physical time is fundamental for the efficiency of the *site stabilization procedure* performed by *Eunomia*.

We now proceed to describe how events are handled by clients, servers and *Eunomia* (Algorithms 1, 2, and 3 respectively).

Read A client c sends a read request operation on a data item (identified by *Key*) to the server(s) that host the partition (p_n) responsible for *Key* (Alg. 1, line 2). When p_n receives the request, it fetches the *Value* and the timestamp *Ts* that is locally stored for *Key* and returns both to the client. *Ts* is the clock used when the update operation was issued. After receiving the pair $\langle Value, Ts\rangle$, the client computes the maximum between $Clock_c$ and *Ts* (Alg. 1, line 4) to ensure that the read operation is included in its causal history.

Update A client c sends an update request operation to the server(s) hosting the responsible partition p_n for the object being updated. Apart from the *Key* and *Value*, the request includes client's clock $Clock_c$ (Alg. 1, line 7). When p_n receives the request, it first computes the timestamp of the new update (Alg. 2, line 5). This is computed by taking the maximum between $Clock_n$ (physical time), the maximum timestamp ever used by p_n ($MaxTs_n$) plus one and $Clock_c$ (client's clock) plus one. This ensures that the timestamp is greater than both $Clock_c$ and any other update timestamped by p_n. Then, p_n stores the *Value* and the recently computed timestamp in the local key-value store and asynchronously sends the operation to the *Eunomia* service. *Eunomia* adds the operation to the set of non-stable operations *Ops* and updates the p_n entry in the *PartitionTime* vector with operation's timestamp (Alg. 3, lines 2–4). Finally, p_n returns the update's timestamp to the client who updates $Clock_c$ with it, since it is guaranteed to be greater than its previous one.

Timestamp Stability We say that a timestamp *Ts* is *stable* at the *Eunomia* servers when one is sure that no update with lower timestamp will be received from any partition (i.e., when *Eunomia* is aware of all updates with timestamp *Ts* or lower). Periodically, *Eunomia* computes the value of the maximum stable timestamp (*StableTime*), which is computed as the minimum of the *PartitionTime* vector (Alg. 3, line 8). Property 2 implies that no partition will ever timestamp an update with an equal or smaller timestamp than *StableTime*. Thus, *Eunomia* can confidently serialize all operations tagged with a timestamp smaller than or equal to *StableTime* (Alg. 3, line 9). *Eunomia* can serialize them in timestamp order, which is consistent to causality (Property 1), and then send them to other geo-locations (Alg. 3, line 10). Note that non-causally related updates coming from different partitions may have been timestamped with the same value. In this case, operations are known to be concurrent and *Eunomia* can process them in any order. For instance, it could use the identifier of the partition that sent the update to break ordering ties.

Heartbeats If a partition p_n does not receive an update for a fixed period of time, it will send a heartbeat including its current time to *Eunomia* (Alg. 2, lines 10–12). This is fundamental to ensure progress of the *site stabilization procedure*. Thus, if a partition p_n receives updates at a slower pace than others, it will not slow down the processing of other partitions updates at *Eunomia*. When *Eunomia* receives a heartbeat from p_n, it simply updates its entry in the *PartitionTime* vector (Alg. 3, line 6).

C. Fault-Tolerance

In the description above, for simplicity, we have described the *Eunomia* service as if implemented by a single non-replicated server. Naturally, as any other service in a datacenter, *Eunomia* must be made fault-tolerant. In fact, if *Eunomia* fails, the *site stabilization procedure* stops, and thus, local updates can no longer be propagated to other geo-locations.

Algorithm 3 Operations at *Eunomia*

```
1: function ADD_OP(Op)
2:     Ops ← Ops ∪ Op
3:     ⟨Key, Value, Ts, p_n⟩ ← Op
4:     PartitionTime[p_n] ← Ts

5: function HEARTBEAT(p_n, Ts)
6:     PartitionTime[p_n] ← Ts

7: function PROCESS_STABLE                    ▷ Every θ time
8:     StableTime ← MIN(PartitionTime)
9:     StableOps ← FIND_STABLE(Ops, StableTime)
10:    PROCESS(StableOps)
11:    Ops ← Ops \ StableOps
```

Algorithm 4 Operations at *Eunomia* replica e_f

```
1: function PROCESS_STABLE                    ▷ Every θ time
2:     if Leader_f == e_f then
3:         StableTime ← MIN(PartitionTime_f)
4:         StableOps ← FIND_STABLE(Ops_f, StableTime)
5:         PROCESS(StableOps)
6:         Ops_f ← Ops_f \ StableOps
7:         send STABLE(StableTime) to Replicas_f \ {e_f}

8: function STABLE(StableTime)
9:     StableOps ← FIND_STABLE(Ops_f, StableTime)
10:    Ops_f ← Ops_f \ StableOps

11: function NEW_LEADER(e_g)
12:    Leader_f ← e_g
```

In order to avoid such limitation, we now propose a fault-tolerant version of *Eunomia*.

In this new version, *Eunomia* is composed by a set of *Replicas*. Algorithm 4 shows the behaviour of a replica e_f of the fault-tolerant *Eunomia* service. We assume the initial set of *Eunomia* replicas is common knowledge: every replica knows every other replica and all servers (that implement data partitions) know the full set of replicas. Partition servers send operations and heartbeats (Alg. 2, lines 8 and 12 respectively) to the whole set of *Eunomia* replicas. Each replica processes operations and heartbeats exactly as in Algorithm 3. Note that the algorithm is deterministic, and its output does not depend on the order of inputs. Thus, if the system would become quiescent, all replicas of *Eunomia* would eventually reach the same state and output the same stream of (serialized) operations.

To avoid unnecessary redundancy when exchanging metadata among datacenters, a leader-based strategy is used to select the replica in charge of propagating this information. The existence of a unique leader is not required for the correctness of the algorithm; it is simply a mechanism to save network resources. Thus, any leader election protocol designed for asynchronous system can be plugged into our implementation (such as Ω [**?**]). A change in the leadership is notified to a replica e_f through the NEW_LEADER function (Alg. 4, line 12).

The notion of a leader is used to optimize the service's operation as follows. When the PROCESS_STABLE event is triggered, only the leader replica computes the new stable time and processes stable operations (Alg. 4, lines 2–5). Then, after operations have been processed, the leader sends the recently computed *StableTime* to the remaining replicas (Alg. 4, line 7). When replica e_f receives the new stable time, it removes the operations already known to be stable from its pending set of operations, since it is certain that those operations have been already processed (Alg. 4, lines 9–10).

D. Discussion

Correctness: We provide an informal proof that our protocol satisfies the two properties required by *Eunomia* (Properties 1 and 2).

Property 2 is trivial to prove. We need to ensure that updates handled by a partition p_n are tagged with strictly increasing timestamps and that heartbeats do not break the monotonicity. By Algorithm 2 line 5, p_n ensures that consecutive updates are tagged with increasing timestamps. On the other hand, heartbeats are only sent when the physical clock at p_n is greater or equal to the lastest timestamp used to tag an update plus a fixed time Δ (Alg. 2, line 10). This ensures that a heartbeat message is always tagged with a larger timestamp that all previously processed updates. Finally, an update happening right after a heartbeat is always tagged with a larger timestamp than the heartbeat's timestamp since the physical clock ($Clock_n$) is used to compute update's timestamp and this is assumed to increase monotonically (Alg. 2 line 5).

In order to prove Property 1 we need to prove that the partial order derived from update timestamps is consistent with causality. The three properties of causality ($\rightsquigarrow$) are:

(i) *Execution thread:* for two operations a and b issued during the same client session, if a happens before b then $a \rightsquigarrow b$;
(ii) *Read from:* for an update operation a and a read operation b, $a \rightsquigarrow b$ if b reads the state written by a;
(iii) *Transitivity:* for operations a, b and c, if $a \rightsquigarrow c$ and $c \rightsquigarrow b$, then $a \rightsquigarrow b$.

These properties, applied to our protocol, imply that an update u_j issued by client c has to be tagged with a timestamp strictly greater than all its previous updates and than any version previously read. $Clock_c$, which is the clock maintained by the client, aggregates the client's causal history in a single scalar. By Algorithm 2 line 5, we know that the timestamp assigned to a client update is strictly greater than $Clock_c$. Thus, we only need to prove that $Clock_c$ is always equal or greater than all previously read versions, ensured by Algorithm 1 line 4, and that it is always greater or equal to the timestamp assigned to its last update, ensured by Algorithm 1 line 9.

Hybrid Clocks Hybrid clocks [15] are a combination of physical time and logical time. The use of physical time is fundamental for *Eunomia*. Although *Eunomia* could simply use logical clocks, and still be correct, the rate at which

clocks from different partitions progress would depend on the workload. This fact could cause the *site stabilization procedure* to progress slowly, delaying remote update visibility. On the other hand, physical clocks from different partitions always move at similar speed independently of the workload characterization. This fact significantly stimulates the *site stabilization procedure*, making update processing faster in comparison to logical clocks.

On the other hand, the logical part of the hybrid clock is meant to improve the protocol's efficiency. One problem of using loosely synchronized physical clocks is that in order to ensure correctness, partitions may add waiting periods to allow clocks of different partitions to catch up due to clock drifts. In fact, protocols such as [8], [9] suffer from this problem. The logical part of the hybrid clock avoids artificial delays due to clock synchronization uncertainties. It represents the largest clock seen by the process maintaining the hybrid clock. In order to ensure that the hybrid clock increases monotonically, the maximum between the physical time and the logical time is returned when the clock is read. Our current prototype implements a simple version of hybrid clocks; a more elaborate implementation, able to bound the divergence between the physical and the logical part of the clock can be derived following [15].

Optimizing the Communication Patterns *Eunomia* constantly receives operations and heartbeats from partitions. This is an all-to-one communication schema and, if the number of partitions is large, it may not scale in practice. In order to overcome this problem and efficiently manage a large number of partitions, two simple techniques have been used: *(i)* build a propagation tree among partition servers; and *(ii)* batch operations at partitions, and propagate them to *Eunomia* only periodically. Both techniques (that can be combined) aim at reducing the amount of messages received by *Eunomia* per unit of time at the cost of a slight increase in the stabilization time.

III. Geo-replication

In our previous protocol we have shown how to unobtrusively timestamp local updates in a partial order consistent with causality. Still, we have not shown how to enforce causal consistency across geo-locations. In this section, we complete our previous protocol with the necessary mechanisms to ensure that remote updates—coming from other geo-locations—are made visible locally without violating causality. We assume a total of M geo-locations, each of them replicating the full set of objects. Each of these geo-locations uses the *Eunomia* service and thus propagates local updates in a total order consistent to causal consistency. We assume FIFO links between datacenters.

A. Protocol Extensions

We proceed to explain how the metadata is enriched and the changes we need to apply to our previous algorithms. Table II provides a summary of the notation used in this section.

M	Number of datacenters
$VClock_c$	Client c vector (M entries)
r_m	Receiver at datacenter m
$SiteTime_m$	Applied updates vector at r_m
$Queue_m$	Queues of pending updates at r_m
$u_j.vts$	Update u_j timestamp vector (M entries)

Table II
NOTATION USED IN THE GEO-REPLICATED PROTOCOL EXTENSION.

Updates are now tagged with a vector with an entry per datacenter, capturing inter-datacenter dependencies. The client clock is consequently also extended to a vector ($VClock_c$).

Update When a client c issues an update operation, it piggybacks its $VClock_c$ summarizing both local and remote dependencies. A partition p_n computes u_j vector timestamp ($u_j.vts$) as follows. First, the local entry of the vector $u_j.vts[m]$ is computed as the maximum between $Clock_n$, $MaxTs_n + 1$ and $VClock_c[m] + 1$, similarly to Algorithm 2, line 5. This permits *Eunomia* to still be able to causally order local updates based on $u_j.vts[m]$. Second, the rest of the entries (remote datacenter entries) are assigned to its sibling entries in $VClock_c$. When the operation is completed, p_n returns $u_j.vts$ to the client who can directly substitutes its $VClock_c$ since $u_j.vts$ is known to be strictly greater than $VClock_c$.

Read Read operations execute as in Algorithms 1 and 2. The only difference is that the returned timestamp is a vector instead of a scalar. Thus, in order to update $VClock_c$, a client c applies the MAX operation per entry.

Update Propagation The *site stabilization procedure* proceeds as before, totally ordering local updates based on the local entry of their vector timestamp ($u.vts[m]$). *Eunomia* propagates local updates to remote datacenters in $u.vts[m]$ order. Each update piggybacks its $u.vts$.

Remote Update Visibility Algorithm 5 shows how a receiver handles a remote update arrival. When datacenter m receives a remote update u_j coming from datacenter k, two conditions have to be satisfied before applying it locally: *(i)* all previously received updates coming from k have already been applied locally; and *(ii)* u_j dependencies, which are subsumed in $u_j.vts$, are visible locally. Both conditions can be trivially checked. The first condition can be enforced by simply keeping a queue ($Queue_m[k]$) of pending updates per remote datacenter (Alg. 5, line 3). The second condition can be enforced by maintaining a vector with an entry per remote datacenter ($SiteTime_m[k]$) indicating the latest update operation applied from each of the remote datacenters (Alg. 5, line 4). If any of the two conditions is not satisfied, u_j is added to $Queue_m[k]$ and will be eventually applied when both conditions hold.

FLUSH_PENDING is a recursive function (Algorithm 5 line 9) that makes sure no pending operation satisfying the

Algorithm 5 Operations at r_m

```
1: function NEW_UPDATE(u_j, k)
2:     Queue_m[k] ← [Queue_m[k]|u_j]                    ▷ add to tail
3:     if HEAD(Queue_m[k]) == u_j then
4:         if ∀d ∈ M \ {m, k}, SiteTime_m[d] ≥ u_j[d] then
5:             send APPLY(u_j) to server
6:             receive ok from server
7:             SiteTime_m[k] ← u_j[k]
8:             POP(Queue_m[k])
9:             FLUSH_PENDING
```

above two conditions is left without being applied. When a pending operation u_j originating at k is applied, both $Queue_m[k]$ and $SiteTime_m[k]$ are updated consequently.

B. Discussion

Vector Clocks: Our protocol relies on vector clocks to ensure causal consistency across different geo-locations. We could easily adapt our protocols to use a single scalar, as in [8]. Nevertheless, vector clocks make a more efficient tracking of causal dependencies introducing no false dependencies across datacenters, which reduces the update visibility latency, at the cost of slightly increasing the storage and computation overhead. Note that the lower-bound update visibility latency for a system relying on vector clocks is the latency between the originator of the update and the remote datacenter, while with a single scalar it is the time distance to the farthest datacenter regardless of the originator of the update.

Separation of Data and Metadata In the protocols described before, partitions send updates (including the update value) to the *Eunomia* service, which is responsible for eventually propagating them to remote datacenters. This can limit the maximum load that *Eunomia* can handle and become a bottleneck due to the potentially large amount of data that has to be handled. In order to overcome this limitation, we propose decoupling data and metadata.

Thus, in our prototype, for each update operation, partitions generate a unique update identifier ($u.id$), composed of the local entry of the update vector timestamp ($u.vts[m]$) and the object identifier (*Key*). We avoid sending the value of the update to *Eunomia*. Instead, partitions only send the unique identifier $u.id$ together with the partition id (p_n). *Eunomia* is only responsible for handling and propagating these lightweight identifiers, while the partitions itself are responsible for propagating the update values together with $u.id$ to its sibling partitions in other datacenters. A receiver r_m proceed as before, but a partition p_n can only execute the remote operation once it has received both the data and the metadata. This technique slightly increases the computation overhead at partitions, but it allows *Eunomia* to handle a significantly heavier load independently of update values.

IV. Implementation

The *Eunomia* service has been implemented in the `C++` programming language and integrated with a version of Riak KV [13], a weakly consistent datastore used by many companies offering cloud-based services including bet365 and Rovio. Since Riak KV is implemented in Erlang, we first attempted to build *Eunomia* using the `Erlang/OTP` framework, but unfortunately we rapidly reached a bottleneck in our early experiments due to the inefficiency of `Erlang` data structures. Note that for *Eunomia* to work, we need to store a potentially very large number of updates, coming from all logical partitions composing a datacenter, and periodically traversed them in timestamp order when a new stable time is computed. Inserting and traversing this (ordered) set of updates was limiting the maximum load that *Eunomia* could handle. The `C++` implementation does not suffer from these performance limitations.

Furthermore, in order to fully explore the capacities of *Eunomia* and experimentally demonstrate our hypothesis, we have integrated *Eunomia* with a causally consistent geo-replicated datastore implementing the protocol presented in §II-A and §III. Our prototype, namely *EunomiaKV*, is built as a variant of Riak KV [13], and includes the optimizations discussed in §II-D and §III-B. Since the open source version does not support replication across Riak KV clusters, we have also augmented the open source version of Riak KV with geo-replication support.

V. Evaluation

Our main goal with the evaluation is to show that *Eunomia* does not suffer from the limitations of the competing approaches. Therefore, we compare *Eunomia* with both *sequencer based* approaches and with approaches based on *global stabilization checking* procedures. We recall that the main disadvantage of sequencers is to throttle throughput, by operating in the critical path of local clients. Therefore, we aim at showing that *Eunomia* does not compromise the intra-datacenter concurrency and can reach much better throughput than sequencer based approaches. Conversely, the disadvantage of the global stabilization approach is to introduce long delays in update visibility at remote sites. Thus, we also aim at showing that *Eunomia* does not significantly delay remote update visibility.

Experimental Setup The experimental test-bed used is a private cloud composed by a set of virtual machines deployed over 20 physical machines (8 cores and 40 GB of RAM) connected via a Gigabit switch. Each VM, which runs Ubuntu 14.04, and is equipped with 2 (virtual) cores, 10GB disk and 9GB of RAM memory; is allocated in a different physical. Before running each experiment, physical clocks are synchronized using the NTP protocol [14] through a near NTP server.

Workload Generator Each client VM runs its own instance of a custom version of Basho Bench [16], a load-generator and benchmarking tool to conduct accurate and repeatable performance tests. For each experiment, we deploy as many client instances as possible without overloading the system.

In our experiments, unless specified, we use the following parameters. Values used in operations are a fixed binary of

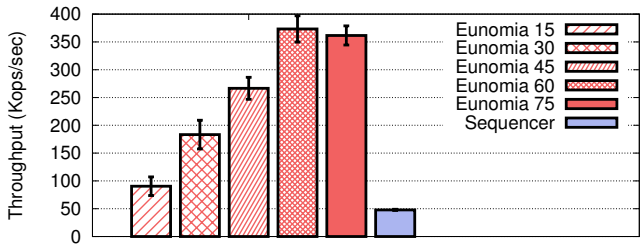

Figure 1. Maximum throughput achieved by *Eunomia* and an implementation of a sequencer. We vary the number of partitions that propagate operations to *Eunomia*.

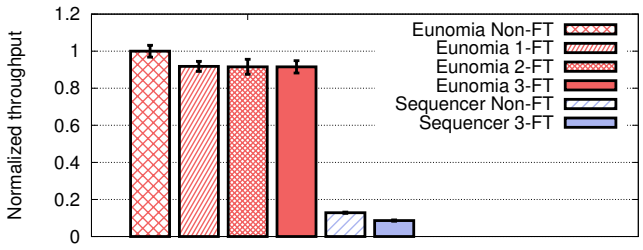

Figure 2. Maximum throughput achieved by a fault-tolerant version of *Eunomia* and sequencers. Non-FT denotes non fault-tolerant versions while 1-, 2-, and 3-FT denote fault-tolerant versions with 1, 2, and 3 replicas

100 bytes. We use a uniform key distribution across a total of 100k keys (objects). Each experiment runs for 10 mins and the first and the last minute of each experiment is ignored to avoid experimental artifacts.

A. *Eunomia Throughput*

We first report on a number of experiments that aim at: *(i)* experimentally measuring the maximum load that our efficient implementation of *Eunomia* can handle, varying the number of partitions connected to it; *(ii)* comparing *Eunomia* to traditional sequencers; and *(iii)* assessing how failures affect the performance of the *Eunomia* service.

For comparison, these experiments also show the maximum load that a traditional sequencer can handle. Our implementation of a sequencer is the simplest possible and mimics traditional implementations. In every update operation, data servers synchronously request a monotonically increasing number to the sequencer before returning to the client. We have also implemented a fault-tolerant version of the sequencer based on chain replication [**?**]: Replicas of the sequencer are organized in a chain. Partitions send requests to the head of the chain. Requests traverse the chain up to the tail. When the tail receives a request, it replies back to the data server, which in turn returns to the client. In our experiments the chain is composed of three replicas.

In order to stretch as much as possible the implementation, circumventing other potential sources of bottlenecks in the system, we directly connect clients to *Eunomia*, bypassing the data store. Thus, each client simulates a different server (in charge of a data partition) in a multi-server datacenter. This allowed us to emulate the use of very large datacenters, with much more data servers than the ones that were at our disposal for this experiments, and overload *Eunomia* in a way that would be otherwise impossible with our testbed.

Throughput Upper-Bound We first compare the non fault-tolerant version of the *Eunomia* against a non fault-tolerant implementation of a sequencer. The *Eunomia* implementation used for the experiment is configured to batch updates and only send them to *Eunomia* after 2ms. Figure 1 plots the maximum throughput achieved by both services.

As results show, *Eunomia* maximum throughput is reached when having 60 servers (data partitions) issuing operations eagerly (with zero waiting time between operations). We observe that *Eunomia* is able to handle almost an order of magnitude more operations per second than a sequencer (more precisely, 7.7 times more operations, exceeding 370kops while the sequencer is saturated at 48kops). Considering that according to our experiments, a single machine in a Riak cluster is able to handle approximately 3kops per second, These results confirm that sequencers limit intra-datacenter concurrency and can easily become a bottleneck for medium size clusters (i.e, for clusters above 150 machines, the sequencer would be the limiting factor of system performance), even assuming a read dominant (9:1) workload, a common workload for internet-based services. On the other hand, under the same workload assumptions, more than a thousand data servers could be used before saturating *Eunomia*.

Another advantage of *Eunomia* in comparison to sequencers is that, as discussed in §II-D, batching is not in the client critical path. Thus, *Eunomia*'s throughput can be further stretched by increasing the batching time (at the cost of slightly increasing the remote update visibility latency). Such stretching cannot be easily achieved with sequencers, as any attempt to batch requests at the sequencer blocks clients.

Fault-Tolerance Overhead In the following experiments we measure the overhead introduced by the fault-tolerant version of *Eunomia* and the impact of faults in the service. Figure 2 compares the maximum throughput achievable by *Eunomia* when increasing the number of replicas up to three. For completeness, the plot also includes the throughput for a non fault-tolerant sequencer and its fault-tolerant version with three replicas. We normalized the throughput against the non fault-tolerant version of *Eunomia*. As results show, the fault-tolerant version of *Eunomia* only adds a small overhead (roughly 9% penalty) independently on the number of replicas. We expect this overhead to increase as the number of replicas increases, but we consider three replicas to be a realistic number. On the other hand, adding fault-tolerance to the sequencer version adds a penalty of almost 33%, thus being more expensive proportionally. The reason for this difference is that, as explained before, *Eunomia* replicas do not need to coordinate as their results are

independent of relative order of inputs, while sequencer replicas need to coordinate to avoid providing inconsistent sequence numbers.

B. Experiments with Geo-Replication

We now report on a set of experiments offering evidence that a causally consistent geo-replicated datastore built using *Eunomia* is capable of providing higher throughput and better quality-of-service than previous solutions that avoid the use of local sequencers.

For this purpose, we have implemented GentleRain [8] and a variation of it that uses vector clocks instead of a single scalar to enforce causal consistency across geo-locations. The latter resembles the causally consistency protocol implemented by Cure [9]. Both approaches are sequencer-free implementations that rely on a global stabilization checking procedure in order to apply operations in remote locations consistently with causality. For this, sibling partitions across datacenters have to periodically send heartbeats, and each partition within a datacenter has to periodically compute its local-datacenter stable time. In our experiments, we set the frequency of this events to 10 ms and 5 ms respectively unless otherwise specified. Both approaches are implemented using the codebase of *EunomiaKV* and thus integrated with Riak KV. To simplify the implementation, we avoided building the propagation tree and used all to all communication to calculate the GST. This is an acceptable design choice as tree is only required if the number of partitions in the cluster is large.

In most of our experiments, we deploy 3 datacenters, each of them composed of 8 logical partitions balanced across 3 servers. The emulated round-trip-times across datacenters (DCs) are 80ms between DC1 and both DC2 and DC3, and 160ms between DC2 and DC3.

1) Throughput: In the following experiments, we measure the throughput provided by *EunomiaKV*, GentleRain, Cure, and an eventually consistent multi-cluster version of Riak KV. Note that the latter does not enforce causality, and thus partitions execute remote updates as soon as they are received. Therefore, the comparison of Eunomia with Riak KV allows to assess the overhead induced by Eunomia for providing causal consistency. As discussed below, this overhead is very small.

We experiment with both uniform and power-law key distributions, denoted with U and P respectively in Figure 3. For each of them, we vary the read:write ratio (99:1, 90:10, 75:25 and 50:50). These ratios are representative of real large internet-based services workloads. As shown by Figure 3, the throughput of all solutions decreases as we increase the percentage of updates. Nevertheless, *EunomiaKV* always provides a comparable throughput to eventual consistency. Precisely, on average, *EunomiaKV* only drops 4.7% of throughput, being extremely close in read intensive workloads (1% drop). On the other hand, GentleRain and Cure are always significantly below both eventual consistency and *EunomiaKV*. This is due to the cost of the global stabilization checking procedure. Note that the throughput difference

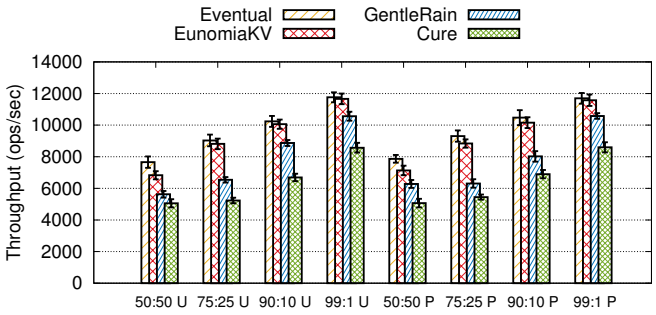

Figure 3. Throughput comparison between *EunomiaKV* and state-of-the-art sequencer-free solutions.

between GentleRain and Cure is caused by the overhead introduced by the metadata enrichment procedure of the latter (as discussed in §III-B). Based on our experiments, it is possible to conclude that the absolute number of updates per unit of time is the factor that has the largest impact in *EunomiaKV* (rather than key contention).

2) Remote Update Visibility: In order to compare the quality-of-service that can be provided by *EunomiaKV*, GentleRain, and Cure, we measure remote update visibility latency. In *EunomiaKV*, we measure the time interval between the data arrival and the instant in which the update is executed at the responsible partition. Note that, for an update to be applied, a datacenter needs to have access to the metadata (in our case, provided by *Eunomia*) and check that all of its causal dependencies have also been previously applied locally. In our implementation, partitions ship updates immediately to remote datacenters. Therefore, we have observed that updates are always locally available to be applied by the time metadata indicates that its causal dependencies are already satisfied locally. Although other strategies could be used to ship the payload of the updates, this has a crucial advantage for the evaluation of Eunomia: under this deployment the update visibility latency is exclusively influenced by the performance of the metadata management strategy, including the stabilization delay incurred at the originating datacenter.

On the other hand, for GentleRain and Cure, we measure the time interval between the arrival of the remote operation to the partition and when the global stabilization checking procedure allows its visibility. Note that all values presented in the figures already factor-out the network latencies among datacenters (which are the same for all protocols); thus numbers capture only the artifical artifacts inherent to the different approaches.

Figure 4 shows the cumulative distribution of the latency before updates originating at DC1 become visible at DC2. We observe that *EunomiaKV* offers, by far, the best remote update visibility latency. Unsurprisingly, GentleRain extra delay is larger than Cure's because of the amount of false dependencies added when aggregating causal dependencies into a single scalar. In fact, GentleRain is not capable of

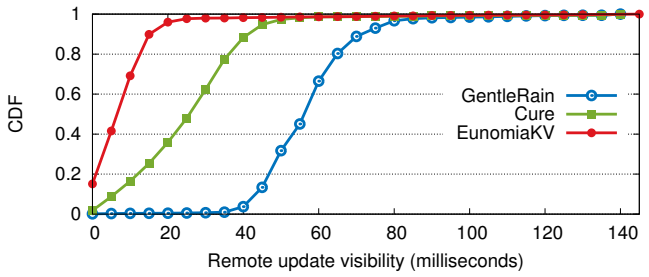

Figure 4. Visibility latency of remotes updates originating at DC1 measured at DC2 (40ms trip-time).

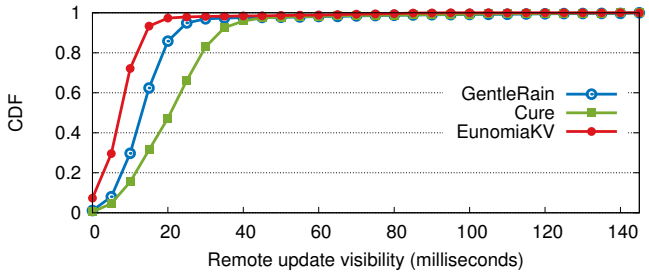

Figure 5. Visibility latency of remotes updates originating at DC2 measured at DC3 (80ms trip-time).

making updates visible without adding 40ms of extra delay. Again, the scalar is the cause of this phenomenon since the minimum delay will not depend on the originator of the update but on the travel time to the furthest datacenter. This confirms the rationale presented in the discussion of §III-B.

Finally, in order to isolate the impact of GentleRain's global stabilization checking procedure independently of the metadata size, we measure the remote update visibility latency at DC3 for updates originating at DC2. As one can observe in Figure 5, GentleRain exhibits better remote update latencies than Cure but still worse than *EunomiaKV*. In this setting, vector clocks does not help reducing latencies. Thus, the gap between Cure and GentleRain is exclusively due to the storage and computational overhead caused by vector clocks. Furthermore, the fact that *EunomiaKV* still provides better latencies is, once again, an empirical evidence that global stabilization checking procedures are expensive in practice.

3) Throughput vs Remote Visibility Tradeoff: In this last subsection, we discuss the tradeoff between throughput and remote update visibility latency posed by solutions relying on a global stabilization checking procedures. In our previous experiments, we set the local-datacenter stable time computation frequency to 5ms, which, according to our experiments, does the best tradeoff. In this section we explore this tradeoff in more detail, to provide a better insight on the cost of global stabilization checking procedures.

For this purpose, we vary the local-datacenter stable time computation frequency between 1ms and 100ms. We measure both the remote update visibility latency and the

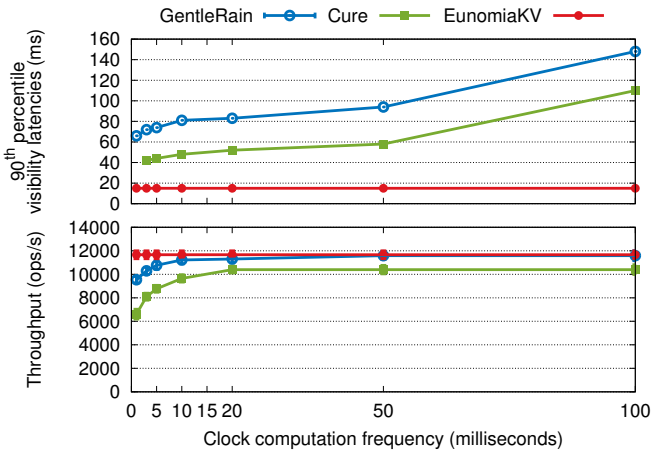

Figure 6. Tradeoff between throughput and remote update visibility latency posed by global stabilization checking procedures.

throughput. Again, latencies refer to the artificial delay added by the system at DC2 for updates originating at DC1. Figure 6 shows the results of this experiment. The graph at the bottom plots throughput numbers. The graph on the top plots the 90^{th} percentile of remote update visibility latencies. As one can observe, as we increase the frequency of the computation, lower artificial delays are added but also lower throughput is provided. We can observe that the throughput decreases more rapidly when the experiment moves towards higher frequencies. The figure also shows the throughput and update visibility latencies of *EunomiaKV*. Even when the computation is very frequent (1ms), GentleRain and Cure do not reach remote update latencies comparable to *EunomiaKV* while its drop in throughput is quite significant (more precisely 18.2% for GentleRain and 44.4% for Cure). We noticed that our version of Cure was slightly overloaded for 1ms frequency. Thus, artifical delays where drastically increased (90^{th} percentile of 75ms), even surpassing GentleRain's 90^{th} percentile (66ms) for the same frequency. We have not plotted this to avoid obstructing plot readability.

These results confirm that global stabilization checking procedures are quite expensive in practice and that solutions based on them are far from achieving remote update latencies close to optimal latencies.

VI. Related Work

Recently, and tackling scalability challenges close to ours, multiple weakly consistent geo-replicated data stores implementing causal consistency across geo-locations have been proposed. We group them into two categories: *(i)* sequencer-based solutions [7]; *(ii)* and sequencer-free solutions [5], [6], [8], [9].

Sequencer-based These solutions rely on a single sequencer per datacenter to enforce causal consistency. The sequencer is in charge of totally ordering local updates, in a causally consistent manner, and propagate them to remote locations.

The use of synchronous sequencers significantly limits the intra-datacenter concurrency, as demonstrated by our experiments. We have shown that sequencers may get easily saturated for medium-size clusters, while *Eunomia* is able to handle much heavier loads (up to 7.7 times more).

Sequencer-free There have been two major trends in this category: *(i)* solutions that rely on explicit dependency check messages [5], [6]; and *(ii)* solutions based on global stabilization checking procedures [8], [9].

COPS [5] finely track dependencies for each individual data item. Remote updates are tagged with a list of dependencies. When a datacenter receives a remote update, it issues an explicit dependency checking messages per dependency. This process is known to be expensive and to limit systems performance [8] due to the large amount of metadata generated. Orbe [6] only partially solves this problem by aggregating dependencies belonging to the same logical partition into one scalar.

As a solution to the metadata problem, research community has proposed alternatives that rely on a background *global stabilization checking* procedure [8], [9]. This procedure equips partitions with sufficient information to safely execute remote updates consistently with causality. As our extensive evaluation has empirically demonstrated, global stabilization checking procedures are expensive in practice. *EunomiaKV* exhibits significantly better throughput than these solutions (therefore much better than solutions based on explicit dependency check messages [8]). In addition, our evaluation have shown that *EunomiaKV* generates substantially smaller remote update visibility latencies than GentleRain and Cure, the two most performant solutions of the state-of-the-art.

VII. Conclusions

We have presented a novel approach for building weakly consistent geo-replicated data stores that require updates to be causally ordered. Our solution relies on a new service, namely *Eunomia*, that abstracts the internal complexity of datacenters, a key characteristic to inexpensively implement causal consistency across geo-locations. Furthermore, unlike sequencers, *Eunomia* does not limit the intra-datacenter concurrency by performing a cheap and unobtrusive ordering of updates.

The paper described the techniques behind *Eunomia* and a high performant causally consistent geo-replicated protocol that integrates it. Our experimental results demonstrate that *Eunomia*, unlike sequencers, is able to handle very heavy loads without becoming a performance bottleneck (up to 7.7 times more operations per second than a sequencer). Furthermore, we have built a data store, namely *EunomiaKV*, that implements the proposed geo-replicated protocol. Our prototype is built as a variant of the Riak KV data store. According to our experiments, *EunomiaKV* is currently the most performant causal consistency implementation, providing appreciably higher throughput and smaller update visibility delay than GentleRain and Cure, the two most performant solutions of the state-of-the-art.

Acknowledgement

This work has been performed in collaboration with Manuel Bravo, a member of the Distributed Systems Group at INESC-ID. We would like to thank Kuganesan Srijeyanthan for his help on the C++ version of *Eunomia*.

References

[1] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 181–192, Nov. 2013. [Online]. Available: http://dx.doi.org/10.14778/2732232.2732237

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[3] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," ser. PODC, 2015.

[4] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," ser. SOSP, 1997.

[5] 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," ser. SOSP, 2011.

[6] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," ser. SoCC, 2013.

[7] S. Almeida, J. a. Leitão, and L. Rodrigues, "Chainreaction: A causal+ consistent datastore based on chain replication," ser. EuroSys, 2013.

[8] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," ser. SoCC, 2014.

[9] D. D. Akkoorath, A. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," ser. ICDCS, 2016.

[10] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Trans. Comput. Syst.*, 1992.

[11] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Trans. Comput. Syst.*, vol. 9, no. 3, Aug. 1991.

[12] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, "Corfu: A shared log design for flash clusters," ser. NSDI, 2012.

[13] "Riak KV," https://github.com/basho/riak_kv.

[14] "The network time protocol," http://www.ntp.org.

[15] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," ser. OPODIS, 2014.

[16] "Basho Bench," http://github.com/basho/basho_bench.