Notes on Eventual Consistency

Francesc D. Muñoz-Escoí, José Ramón González de Mendívil,
Juan Salvador Sendra-Roig, José-Ramón García-Escrivá, José M. Bernabéu-Aubán

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
46022 Valencia (SPAIN)

fmunyoz@iti.upv.es, mendivil@unavarra.es, {jsendra,rgarcia,josep}@iti.upv.es

Technical Report TR-IUMTI-SIDI-2016/002

Notes on Eventual Consistency

Francesc D. Muñoz-Escoí, José Ramón González de Mendívil,
Juan Salvador Sendra-Roig, José-Ramón García-Escrivá, José M. Bernabéu-Aubán

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
46022 Valencia (SPAIN)

Technical Report TR-IUMTI-SIDI-2016/002

e-mail: fmunyoz@iti.upv.es, mendivil@unavarra.es, {jsendra,rgarcia,josep}@iti.upv.es

October 11, 2016

Abstract

Eventual consistency is demanded nowadays in geo-replicated services that need to be highly scalable and available. According to the CAP constraints, when network partitions may arise a distributed service should choose between being strongly consistent or being highly available. Since scalable services should be available, a relaxed consistency (while the network is partitioned) is the preferred choice. Eventual consistency is not a common data-centric consistency model, but only a state convergence condition to be added to a relaxed consistency model.

There are still two aspects of eventual consistency that have not been analysed in depth in previous works: which have been the oldest replication proposals providing eventual consistency, and which data-centric consistency models are not convergent by default (and, thus, provide the basis for building eventually consistent services). This paper provides some notes on these important topics.

KEYWORDS: Eventual consistency, Consistency model, CAP theorem, Data-centric consistency, Client-centric consistency, Data replication

1 Introduction

Eventual consistency [70] has received a lot of attention in the last decade due to the emergence of elastic distributed services. Elastic services [22, 33] need to be both scalable and adaptive, ensuring good levels of functionality, performance and responsiveness (i.e., QoS) –combined with a low cost– to their users, and of economical profit to their providers. In order to reach those levels of performance and responsiveness when the incoming workload being supported is high, the consistency among server replicas might be relaxed and this explains why eventual consistency has become so popular.

Elastic services are commonly deployed onto multiple datacentres and they may use thousands of computers. In those environments network partitions may arise. According to the CAP theorem, that was first stated by Fox and Brewer (1999) [26] and later proven by Gilbert and Lynch [29], when a network partition happens, there is a tradeoff between strong consistency and service availability. Since elastic services must guarantee availability in order to comply with their QoS requirements, consistency needs to be relaxed in those situations. This is another reason for the success of eventually consistent services. However, as we will see in Section 2, the compromises stated in the CAP theorem were already known 40 years ago.

There have been many recent research works about eventual consistency [60, 70, 50, 66, 2, 6, 8, 7, 15, 56] and some of them, e.g. [60, 6, 8, 15], present tutorials on this subject. In spite of this, some aspects of this concept have not yet been discussed in depth. Therefore, in order to provide the missing pieces for building a complete picture on this subject, our paper is focused on those other topics.

The first point of interest is a historical review. Although most recent research works cite a paper from Werner Vogels [71] (or its former copy in ACM Queue [70]) as the most well-known reference explaining this kind of consistency, there are many papers older than [70] that have either explained this same concept or implemented this kind of consistency. Some of them are revised in Section 2.

Section 3 revises the specifications of some data-centric replica consistency models in order to set the border between those models that are inherently convergent and those others that are too much relaxed to be convergent *per se*. Since eventual consistency means that replica convergence is reached when no new updates are received for a sufficiently long interval, this border identifies those *relaxed* models as the ones that must be taken as a basis to develop eventually consistent services. Moreover, as it has been proven in other papers [58], the inherently convergent replication models cannot be globally maintained when the network is partitioned. Therefore, both reasons compel us to use those relaxed models (e.g., FIFO/PRAM [48] or causal [1]) as the base for eventual consistency.

Section 4 provides some considerations about how to implement an eventually consistent replicated service. Four complementary aspects are considered: (1) replication protocol, (2) ordering requirements for service requests, (3) synchrony in agent interaction, and (4) state merging strategy for reaching convergence. There are multiple alternatives in each aspect and, not surprisingly, the best combinations regarding performance and convergence were already known long time ago.

Therefore, the principles that are needed to manage eventually consistent services had been already proposed and used in several of the oldest distributed services. The challenges at that time were centred in providing acceptable response times (and an acceptable consistency) using very limited computers and networks. Nowadays, current hardware resources are much more powerful, but the services being provided have also much more demanding requirements. They must be highly scalable and immediately adaptive. So, although those old principles may guide our research efforts in this kind of dynamic consistency, new other contributions are still required in this area.

2 Historical Review

Let us assume that we have listened something about eventual consistency and that, at the moment, only have read reference [70] about this concept. Once paper [70] is known, and quoting its contents, we may tell that eventual consistency is...

> *... a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme.*

That definition is informal, but clear and concise. Indeed, as we will see in forthcoming sections, eventual consistency cannot be defined in a formal way as a regular consistency condition, since it is a liveness condition (eventual state convergence) that may be added to other consistency models. Besides defining it, Vogels also mentions a widely known service that implements eventual consistency: the domain name system [51, 52]. Because of this, the reader may realise that traditional scalable distributed services have regularly been eventually consistent. Vogels provides another pointer to a former publication on this subject: Lindsay et al (1979) [47]. Therefore, it seems that eventual consistency has been a *classical* mechanism for achieving high performance in the distributed systems arena.

Taking a look at Section 1.4 from [47], the reader may observe that it describes a distributed relational database system that may use different replication protocols (primary-backup or majority voting) where two kinds of consistency may be managed. In the regular case, providing one-copy equivalence, transaction updates are forwarded and applied to all database replicas before that transaction is ended. On the other hand, with relaxed consistency (i.e., with eventual consistency), those updates may be forwarded afterwards, in a lazy way. This reduces the degree of synchronisation being demanded by transactions, allowing their fast completion. Thus, this is one of the first examples of consistency-performance tradeoff.

Assuming that the usage of relaxed consistency has been a *regular* solution for improving the scalability of distributed services, it will be difficult to find the oldest research work that provided the first example of

service or replication protocol specifically intended for ensuring that kind of consistency. There had been many old systems using replicated components and not all of them described their replication approaches in detail. Any way, let us go on in this backward look for that possible first eventually consistent service.

To this end, we may start considering the first references on replication models. Two classical replication models exist: active or state-machine replication [63] and passive or primary-backup replication. Primary-backup replication was proposed before, and its classical first reference is a paper from Alsberg and Day [3]. In that paper, strong consistency is assumed. Both read and write requests are served by the primary replica. Write operations, once processed, forward their updates to the first backup replica. When this backup applies those updates, it sends the reply to the client process plus an acknowledgement to the primary and an update forwarding message to the next backup replica. The consistency being perceived when this algorithm is followed is strong (indeed, linearisable [34]). In spite of this, that paper also outlines some variations of its basic algorithm. Thus, in page 569, it explains what can be done in multi-master scenarios. The general rule is to reach a consensus on the requests service order among all those master replicas before processing those incoming requests. But that general rule admits an exception that is described in this way in [3]...

> *There may be specific applications where the nature of the service permits the out of order processing of requests. An example is an inventory system where only increments and decrements to data fields are permitted and where instantaneous consistency of the data base is not a requirement.*

...; i.e., some applications may provide an updating interface consisting of multiple commutative operations (e.g., increments and decrements in this example). In that case, multiple master replicas are allowed, serving their requests concurrently. Consistency is eventually achieved when every replica receives and applies all the updates generated (in any order) in the remaining replicas. This is a valid sample of an eventually consistent service and it was described in 1976; i.e., three years before the relaxed algorithm found in [47].

Moreover, Alsberg and Day [3] cite two related papers to look for additional information [14, 36]. Bunch [14] describes a preliminary version of the primary-backup supporting algorithm discussed in [3]. In that version, backup replicas do not need to be linearly ordered and they do not propagate the updates following a chain forwarding approach. Instead, updates are logically multicast to all backups, allowing any kind of multicast implementation. Besides this first difference, there is another one: there are two classes of read operations referred to as critical and noncritical. Critical reads are directly managed by the primary replica. Noncritical read requests are forwarded to the *cheapest* replica (e.g., the one minimising transmission delay). In spite of their name, both read classes are strongly consistent since write operations do not return control to their client until all existing copies have been updated and have acknowledged the update completion. Note that with this solution it is impossible that once a read request returned a value V for a given element, any subsequent read request could return a different value older than V; i.e., there cannot be any read old-new inversions [5]. However, these noncritical reads settled the basis for the relaxed consistency algorithm described in [47], once the synchronous update propagation was replaced with a lazy forwarding.

On the other hand, Johnson and Thomas [36] propose a replication algorithm that is more general than that of [3]. It allows multi-master replication for a given kind of database (a key-value store that maintains users data in a user authentication and accounting system [18]). Each database copy is held by a *database management process* (DBMP). This paragraph is given for stating the consistency to achieve in that system...

> *The extent to which the copies of the database can be kept "identical" must be examined. Because of the inherent delay in communications between DBMPs, it is impossible to guarantee that the data bases are identical at all times. Rather, our goal is to guarantee that the copies are "consistent" with each other. By this we mean that given a cessation of update activity to any entry, and enough time for each DBMP to communicate with all other DBMPs, then the state of that entry (its existence and value) will be identical in all copies of the database.*

The updates applied in a given master replica should be transferred to the remaining replicas. A list of pending replicas is maintained. However, nothing is said about the degree of synchrony of those interactions nor about at which time a reply is sent to the requesting client. In spite of this, the algorithm being described tolerates lazy propagation. Therefore, this could be the first proposal of eventual consistency, since the overall meaning of that paragraph is identical to that presented in [70]. Since multiple writers may exist and they all may concurrently apply conflicting updates in different replicas, some rules were needed to reach a convergent state once those updates were forwarded to the remaining replicas. To this end, Johnson and Thomas designed a solution based on update timestamping. In order to define that timestamping approach some local clock is used in every server, combined with node identifiers to break ties, defining a total order on all system events. The authors assume that those clocks could be sufficiently synchronised by default; otherwise, they suggest the usage of event counters in every node. Indeed, this was a solution that inspired the definition of logical clocks [44], as Lamport acknowledges at the end of his paper. This conflict resolution rule (i.e., "the last writer wins") was also applicable in case of network partitions. In spite of this, Johnson and Thomas mention the following regarding service continuity in case of network partitions...

> *For example, a completely general system must deal with the possibility of communication failures which cause the network to become partitioned into two or more sub-networks. Any solution which relies on locking an element of the database for synchronized modification must cope with the possibility of processes in non-communicating sub-networks attempting to lock the same element. Either they both must be allowed to do so (which violates the lock discipline), or they both must wait till the partition ceases (which may take arbitrarily long), or some form of centralized or hierarchical control must be used, with a resulting dependency of some DBMPs on others for all modifications and perhaps accesses as well.*

Thus, they already identified in 1975 that in case of network partitions there is a tradeoff between service availability and service consistency (since locking was assumed in that paper as a means for ensuring strong consistency); i.e., part of what is known nowadays as the CAP theorem [26, 12, 29]. However, it is worth noting that the specific database being assumed in [36] provided an advantageous scenario for its authors. Its data are seldom updated and value convergence only depends on the update time. No semantic consistency constraints need to be considered and the database schema is very simple. As a result, the rules to be followed to recover data convergence are almost trivial. This will not happen in many other cases.

In our humble opinion, the paper from Johnson and Thomas can be considered the first key reference about eventual consistency. It was able to describe an efficient way to implement that kind of consistency (combining lazy update propagation with a general rule to reach convergence in case of conflicting updates, tolerating disconnected operation). Besides their data convergence rule, Johnson and Thomas defined specific mechanisms for detecting and managing delete-update and delete-create conflicts that might be hard to manage in a distributed deployment.

Let us now come back to our days, following a chronological order, to find additional contributions from other relevant papers in this historical review.

A first example is the LOCUS [72] distributed file system partially described by Parker et al [55] in 1981. Its designers took care of handling network partitions, allowing progress in disconnected subgroups of nodes. They also mentioned the tradeoff between strong consistency and service availability when network partitions arise. In LOCUS, consistency was relaxed while disconnected nodes went on and *version vectors* were proposed in order to detect state conflicts, applying *reconciliation protocols* at reconnection time. Those reconciliation protocols, according to Parker et al, depend on the semantics of the operations being applied to the replicated resources. Note that the maintenance of version vectors at each replica implies that the updates being propagated comply with causal consistency.

A second example is the Grapevine system developed at Xerox by Birrell et al [9]. Grapevine was an electronic mail service that also provided support for resource location, authentication and access control. The communication mechanisms being managed by the Grapevine servers were asynchronous (i.e., the sender was able to continue once the message was sent, without waiting for any kind of acknowledgement) and persistent (i.e., the communication servers were able to maintain the messages until their intended receivers were ready to get them). Grapevine needed a registration database where it maintained data about

its users and its groups. A *group* maintained a collection of users addresses, thus allowing that a single e-mail message could be delivered to a set of users specifying their group name. In the Grapevine deployment (1981) described in [9], this system was spread through the Xerox sites at USA, Canada and United Kingdom. There were five registration (and message) servers and around 1500 users defining 500 user groups. The registration database was fully replicated in those registration servers using a multi-master approach. Database updates were forwarded in a lazy way through the asynchronous and persistent communication channels regularly used for electronic mail propagation. The replication algorithms tolerated network partitions, merging any conflicting updates using timestamps and the "last writer wins" principle already described in [36].

Fischer and Michael [25] describe an evolution of the algorithms presented in [36] for managing a distributed directory service. In this new solution, no explicit update operation is provided. Instead, the programmer should apply first a delete operation followed by a new insert. Additionally, the system remembers which objects have been inserted and which others have been deleted. With those sets, it is able to find out whether a given object is still active or not. A criterion for purging removed elements from both sets is also given. It is based on how many servers have already known that information. With all these variations on the Johnson and Thomas algorithm, the result is much simpler (indeed, no delete-update nor delete-create conflicts may arise) and it is still able to tolerate network partitions and unreliable communication, guaranteeing eventual consistency and high availability.

Davidson (1984) [19] provides some rules for allowing service continuity in case of network partitions in a replicated database system. This means that the consistency among replicas is lost while the network remains partitioned, but service availability is guaranteed. However, once the partitioned groups rejoin, Davidson proposes several criteria for detecting serialisability violations and for choosing the set of transactions to be rolled back in order to build a global history that respects all serialisability requirements.

Apers and Wiederhold [4] propose some extensions a bit later (1985). They also study the network partition problem in replicated database systems. However, instead of focusing only on serialisable order, they also consider semantic pre- and post-conditions on each kind of transaction. As a result of this, transactions are classified as: (1) unconditionally committable (UC), when their execution cannot violate any pre-condition of other transactions run in other partitions, (2) conditionally committable (CC), when their acceptance cannot be guaranteed but their possible afterward rejection will not introduce other consistency problems, and (3) non-committable (NC), when their possible afterward rejection leads to consistency problems that will not be solvable. Only UC and CC transactions are accepted in case of a network partition. NC transactions are immediately rejected in that case. Algorithms are presented for merging partitions and for rebuilding their serialisation graphs, applying compensating transactions onto previously accepted CC transactions when needed. This is one of the first examples on managing semantic correctness criteria at partition reconnection time and on using conditional criteria for accepting some classes of transactions while the system is partitioned.

Other semantic criteria for managing state merging at partition reconnection time were proposed by Sarin et al (1985) [61]. They base their solution in timestamping all operations that modify the application or database state, defining in this way a total order for all those operations. However, no detail is given on how such update propagation should be made nor on how those timestamps are globally generated, allowing multiple kinds of implementations, even lazy propagation. When partitions rejoin, they forward and receive any missed updates. Conceptually, when a missed update is received in this reconnection stage, it leads to the roll back of all the operations that had been previously accepted and applied with a higher timestamp, reapplying them later on, in their appropriate order. However, multiple semantic optimisations are described for avoiding both the operation rollback and its reexecution once the missed update has been applied. Some examples mentioned in [61] that do not need those compensating actions are: commutative operations, conflicting operations applied on disjoint sets of data, ...

Demers et al (1987) describe the Clearinghouse system in [21]. In that system "the effect of every update is eventually reflected in all replicas". The system consists of several thousand nodes where a multi-master replication strategy is used. Each update request may be received and processed by a different replica and its effects will be lazily propagated to the remaining sites. The paper proposes and compares different lazy update propagation mechanisms in order to minimise network traffic: direct mail, anti-entropy and rumor mongering. With the latter two approaches the resulting system becomes highly scalable.

The first *computer-supported cooperative work* (CSCW) applications were developed in the middle eighties. The Lotus/Iris Notes project (that later became the Lotus Notes product) was described by Kawell et al in [37]. It was based on lazy update propagation and on the "last writer wins" policy for dealing with concurrent updates. The database being managed was unconventional: it was a collection of documents and a "transaction" consists in an update to one of those documents (multiple documents cannot be updated using a single action). As a result of this, on each update a complete document should be transferred among replicas. Fortunately, documents were small (usually 1 or 2 KB) in that preliminary version of Notes. Those updates were transferred following a pull policy. Notes assumed that computers are not continuously connected to the network. As a result of this, when a computer contacts others they exchange their documents lists, with the versions and IDs for each document. When those lists were compared, the computer that missed any update requested the other to transfer those updates. Following this strategy, all Notes replicas became eventually consistent, but those replicas might had been inconsistent for quite long intervals (e.g., Kawell et al [37] comment that in most cases those intervals exceeded 24 hours).

Kumar and Stonebraker (1988) [39] describe how to apply the *escrow* [54] method to replicated databases, assigning complementary parts of the escrow to each replica; i.e., distributing the escrow. The original escrow mechanism allowed the management of concurrent transactions that use commutative operations to update a given relation field even when the value of such field should respect some constraints (e.g., to be positive or exceed some minimal threshold). Escrow distribution allows the management of some concurrent transactions without exchanging messages among replicas in some cases. This enhances performance and increases the tolerable degree of concurrency. As a result of this, inter-replica consistency is relaxed and transactions serialisability is lost, but transaction correctness is still preserved.

Ladin et al (1988) [41] proposed a directory (or map) service with gossip update propagation. Those updates are only accepted when they comply with the "last writer wins" rule, but now using version vectors instead of timestamps or Lamport clocks. Two years later, in the *lazy replication* approach [42] proposed by the same authors, client requests are forwarded to a single replica that processes the operation and later propagates its updates in a lazy way. The operations being processed may be ordered according to the application semantics, selecting one of these approaches: client ordering, server ordering or global ordering. In the client-order case, a client process may specify which previously initiated operations precede the operation to be sent. To this end, each update operation returns an update identifier (uid) when it is completed and both queries and updates may specify as their arguments a set of precedent uids. In the server-order case, every server-ordered operation is totally ordered by the servers against every other server-ordered operation. Finally, in the global-order case, each global-ordered operation must be totally ordered by the servers against every other operation, independently on the type of the latter. This third type may be used in case of system reconfigurations, defining a border for ensuring that when it is delivered all server replicas must have delivered the same set of previous requests. Indeed, this is placing a convergence point for eventually consistent replicated services.

The systems to be implemented using the *lazy replication* technique find several advantages when they are compared to previous works. To begin with, they are basing their eventual consistency on an explicit (instead of potential, as when a regular causal multicast mechanism is being used for propagating updates) causal consistency. This reduces the amount of dependencies to be considered among the updates being propagated, enhancing performance and reducing delivery delays. Those delays may arise, e.g., when a precedent causal message is lost and is resent. A second advantage is the careful management of application semantics for specifying how concurrent operations should be observed by every replica.

The Coda distributed file system [62] (1990) is an example of distributed environment allowing disconnected operation. In this case, consistency is relaxed among clients and servers. Clients get a cache image of each demanded file and may operate on them even when no server may be reached. Using version vectors and file update identifiers, servers may later identify and accept the updates applied by those clients while they were disconnected. When clients remain connected to servers, they forward the updates to every reachable server in a synchronous way. Therefore, relaxed (eventual) consistency only arises at disconnection intervals and does not depend on the workload being supported. Note that other systems relied on lazy propagation (and its resulting eventual consistency) in order to shorten regular operation service time, but that was not the case in Coda. If a previously disconnected client introduces conflicting updates at reconnection time, those state divergences are reported to the user and must be manually merged. No automation is provided by default for managing these conflicts. In spite of this, subsequent releases of

Coda introduced an automated reconciliation process when the application update semantics allows this. Kumar and Satyanarayanan describe an example of this kind applied to directory management [40].

In the scope of relational databases where one-copy serialisability is the regular consistency requirement when replication is used, Krishnakumar and Bernstein (1991) [38] propose a system with lazy writeset propagation (but respecting causal order) where transactions may be accepted although up to N previous transactions executed in other replicas may be missing in the local node. The resulting system is not serialisable, and the resulting correctness criterion is known as N-ignorance. This eventually consistent system ensures enough guarantees for multiple distributed applications (e.g., a flight reservation system, making N equal to the highest overbooking tolerated in that system) and improves concurrency and performance up to N times when it is compared with strictly serialisable systems.

In the same field and year, Pu and Leff [59] propose the ε-serialisability concept based on asynchronous writeset propagation. The resulting executions may be one-copy serialisable for writes but only ε-serialisable for reads, being ε a bound on data divergence. To this end, and due to the asynchrony in write propagation, a query (i.e., read-only) transaction may tolerate to be overlapped with up to ε conflicting concurrent update transactions without becoming aborted. Since the value of ε is configurable, this technique ranges from strict one-copy serialisability to a very relaxed system with eventual replica consistency. Four replication protocols are described in [59]: ORDUP (ordered updates, demanding a total order of updates shared by all sites), COMMU (commutative updates), RITU (read-independent timestamped updates, allowing order freedom in non-conflicting updates) and COMPE (optimistic service, looking later for conflicts and using compensating transactions in order to reach convergence). Eventual replica consistency might be implemented using, for instance, large values for ε combined with a COMMU or COMPE replication protocol. Note, however, that the ORDUP replication protocol does not allow any divergence among the states of replicas. Therefore, ORDUP is inherently convergent.

Bayou (Terry et al 1994) [69] was a replicated storage system to be used in a mobile computing environment where disconnections may frequently arise. It uses lazy propagation of updates providing eventual consistency. However, Bayou introduced *sessions* in order to give a better consistency image to its users. At the server domain the consistency is very relaxed and only eventually convergent, but on each user session the consistency could be stronger depending on the properties being enforced. To this end, Bayou proposed four user-centric consistency guarantees:

- *Read your writes* (RYW): read operations reflect previous writes from the same process.
- *Monotonic reads* (MR): successive reads see a non-decreasing collection of writes.
- *Writes follow reads* (WFR): writes are propagated after the reads they depend on.
- *Monotonic writes* (MW): writes are propagated after writes that precede them.

The combination of WFR and RYW ensures a consistency that is similar to the data-centric causal model. When all these four guarantees are attained, the resulting image perceived by a user session is equivalent to one-copy consistency, but the actual data-centric consistency might still be very relaxed. Note that each operation being executed in a given session may be forwarded to a different server replica. Specific protocols based on version vectors were used in [69] for complying with the consistency guarantees required in sessions.

Fekete et al (1996) [23] provide the first formal specification of an eventually consistent system we are aware of. Their proposal formalises the algorithms described in [42]. To this end, I/O automata [49] are used. Although the specification is tailored to [42], several of the principles that have guided other modern specifications were given in [23]. Thus, it tolerates that operations were executed defining a partial order, but that order progressively tends towards a total order that is needed for reaching state convergence; i.e., operations may be reordered once run. Once the total order is decided for a sequence of operations, those operations are considered *stable* and the state in all replicas should have converged.

Yu and Vahdat (2000) [73] describe an implementation of the TACT middleware that is able to measure the current level of divergence among service replicas and to specify replica consistency requirements considering several aspects. This allows a precise control of replica divergence, based on three complementary dimensions: (1) numerical error (limits the total weight of writes applied across all replicas before being

propagated to a given replica), (2) order error (limits the amount of tentative writes, subject to reordering, that may be pending at a replica), and (3) staleness (places a real-time bound on the delay of write propagation). When all three dimensions have a zero bound, the system ensures linearisable consistency. On the other hand, when no bound is set, eventual consistency is used. TACT may use several algorithms for ensuring that the requested bounds are respected. This defines a continuous space of replica consistency from which the user may choose the adequate level for each deployed replicated service. Indeed, different replicas from the same service may have different bounds depending, for instance, on the characteristics of the hosting computer or on the network bandwidth and delay.

Saito and Shapiro (2005) [60] provide a survey on *optimistic replication*; i.e., replication techniques that relax their concurrency control and consistency in order to achieve greater efficiency and performance since synchronisation is avoided or, at least, minimised. Section 5 of that survey discusses eventual consistency. In that part, Saito and Shapiro provide one of the best definitions of this kind of consistency:

> *A replicated object is eventually consistent when it meets the following conditions, assuming that all replicas start from the same initial state:*
>
> - *At any moment, for each replica, there is a prefix of the schedule that is equivalent to a prefix of the schedule of every other replica. We call this a committed prefix for the replica.*
> - *The committed prefix of each replica grows monotonically over time.*
> - *All non-aborted operations in the committed prefix satisfy their preconditions.*
> - *For every submitted operation α, either α or $\not\alpha$ will eventually be included in the committed prefix.*

It is not a mathematical or formal specification of such concept, but it encompasses all the scenarios that we have depicted in this section. The equivalence of *committed prefixes* among replicas allows the modelling of state convergence when no new updates are received. Their monotonical growth expresses that such convergence is (and will be) reached multiple times but it is not continuously preserved. Precondition accomplishment models the semantic correctness of these eventually consistent systems. Consideration of $\not\alpha$ means that in order to reach convergence and reconciliate from existing conflicts, some of the submitted operations may be discarded, applying other compensating actions (i.e., $\not\alpha$) that eliminate their effects.

Besides this, Saito and Shapiro classify eventually consistent systems depending on how they deal with three relevant problems related to the operations to be executed: ordering, conflicts and commitment.

Ordering refers to the scheduling policy being used for ordering the updates that define the committed prefix at each replica. Additionaly, operations should be ordered in a way expected by users. Five ordering alternatives exist: (i) syntactic ordering, i.e., all nodes should follow the same operation order independently on the semantics of each operation (easy to implement but rises too many conflicts among nodes), (ii) commutative operations, allowing any execution order (no conflict appears but it has a limited applicability), (iii) canonical ordering (limited applicability), (iv) operational transformation, implying the transformation of some operations in order to adapt their results for reaching convergence in a committed prefix (complex procedure that depends on the application semantics), and (v) semantic optimisation (again, too complex).

Conflicts refer to how state conflicts are dealt with and resolved. Two alternatives: (i) syntactic (differences –either in the state values or in the operation order– are used for detecting conflicts and a deterministic criterion is used for resolving them), and (ii) semantic (conflicts are resolved considering the semantics of the involved operations; this is a complex and application-specific solution).

Finally, *commitment* refers to the protocols being used for deciding when an executed operation can be considered stable (i.e., it has been accepted and belongs to a common committed prefix in all replicas) or for reaching agreement on non-deterministic decisions. There are three alternatives: (i) implicit (no operation is ever explicitly rejected; this may be supported using the "last writer wins" approach in case of conflicts), (ii) background agreement (nodes send piggybacked information about their accepted updates in their update propagation messages; e.g., using version vectors), and (iii) consensus (this is a complex and potentially blocking solution, usually needed in strongly consistent systems but not recommended in eventual ones).

Reference	Contributions	Year
Johnson and Thomas [36]	Algorithms for managing a highly-available directory service. Multi-master replication as a way for implementing eventual consistency. Description of a basis for logical clocks. Mechanism for totally ordering the events in a system. Network partition tolerance. "Last writer wins" principle for reaching convergence. Management of delete-update and delete-create conflicts.	1975
Alsberg and Day [3]	Multi-master replication with commutable operations as a way for implementing eventual consistency.	1976
Lindsay et al [47]	Identification of the level of update propagation synchrony as a key aspect for replica convergence: strong consistency with synchronous propagation and eventual consistency with lazy propagation.	1979
Parker et al [55]	Version vectors for detecting inconsistencies in disconnected operation. Potential causal consistency as a base for implementing eventual consistency. Need of semantic reconciliation protocols at reconnection time.	1981
Birrell et al [9]	Deployment of a WAN system supporting eventual consistency.	1982
Fischer and Michael [25]	Avoidance of delete-update and delete-create conflicts in eventually consistent services using multi-master replication.	1982
Davidson [19]	Service continuity in partitioned databases with serialisable histories. Criteria for choosing which transactions to roll back at reconnection time.	1984
Apers and Wiederhold [4]	Service continuity in partitioned relational databases. Consideration of correctness invariants (stated as pre- and post-conditions) at partition reconnection time.	1985
Sarin et al [61]	Service continuity in partitioned databases. Semantic criteria for reordering or avoiding compensating actions at partition reconnection time.	1985
Demers et al [21]	Analysis of three types of lazy update propagation.	1987
Kawell et al [37]	Proposal of a CSCW application (Lotus Notes) with eventual consistency. Pull-based strategy for update propagation. Document propagation for reaching convergence.	1988
Kumar and Stonebraker [39]	Extension of the *escrow* method (management of commutative transactions respecting value constraints) to replicated databases. Serialisability is sacrificed and inter-replica consistency is relaxed, but transaction correctness is maintained.	1988
Satyanarayanan et al [62]	Support for client disconnected operation in distributed filesystems. Manual multi-conflict resolution may be demanded at reconnection time.	1990
Ladin et al [42]	Specification of update ordering requirements, depending on application semantics. Explicit causal dependences (i.e., real causal consistency) as a base for building eventual consistency. Global order as a way for reaching convergence on the set of processed operations in every replica.	1990
Krishnakumar and Bernstein [38]	N-ignorance as an eventually consistent example in replicated relational databases. Lazy writeset propagation with causal order.	1991
Pu and Leff [59]	ϵ-serialisability (relaxed consistency for query transactions in relational databases). Proposal of four replication protocols using asynchronous writeset propagation. Some protocols (COMMU and COMPE) may implement eventual consistency.	1991
Terry et al [69]	Specification of user-centric consistency conditions, instead of the traditional data-centric ones. Sessions for providing user-centric consistency. Users perceive a consistency that is stronger than that maintained by servers.	1994
Fekete et al [23]	Characterisation of eventual consistency based on stable operations (that force state convergence) and reorderable operations, using I/O automata. First formal specification for eventual consistency. Inspired in the replication protocols proposed in [42].	1996
Yu and Vahdat [73]	Evaluation of replica divergence. Selection of a per-replica level of consistency. Specification of consistency based on three axes: (1) numerical error, (2) order error, and (3) staleness.	2000
Saito and Shapiro [60]	Survey on optimistic replication techniques, including eventual consistency. Thorough classification of eventually consistent systems.	2005
Bouajjani et al [11]	Complete formal specification of eventual consistency. Definition of weak eventual consistency, centred only in convergence. Consideration of local program correctness in its consistency specification. Proposal of verification tools for eventually consistent systems.	2014

Table 1: Contributions outlined in this historical review.

This historical review would end here. Its goal has been to show that some research work on eventual consistency existed before Vogels' paper [70] was written. A summary of the contributions found in this review is given in Table 1. However, as it has been said in Section 1, there have been many recent papers on this subject and there is still an aspect that had not been completely dealt with before 2008: the formal specification of what is eventual consistency. There have been a few papers covering that goal [23, 10, 16, 11] and the paper from Bouajjani et al [11] deserves some comments since it has provided the best contributions.

Bouajjani et al [11] criticises that almost all previous definitions of eventual consistency (excepting [16]) had been only centred in state convergence at quiescence intervals. However, those definitions and specifications did not state anything about traces where no quiescent interval exists. In those cases, apparently, no state convergence effort is required. In spite of this, most eventually consistent services actually consider and make those efforts. Therefore, the specification given in [11] considers both: (1) the correctness of the operations being executed by each process, and (2) the conditions to be satisfied when the arrival of new updating requests never stops. Additionally, that paper qualifies quiescent convergence as *weak eventual consistency* and proposes some verification tools for evaluating the correctness of (both weak and non-weak) eventually consistent systems.

3 Eventual Data-centric Consistency Models

Distributed shared memory (DSM) consistency models [27, 53] regularly assume that multiple processors share a given memory and that processes directly run in those processors. Thus, they are centred in what is directly applied to the shared data by those server processes. Those classical memory consistency models are known nowadays as data-centric or server-centric consistency models [68]. In this section, the term *consistency model* will always refer to those data-centric models.

Some recent research works (e.g., [24, 56, 15]) consider that eventual consistency (i.e., state convergence) is a liveness property. Because of this, the requirements stated in many definitions of eventual consistency [60, 70] could be achieved adding a convergence property to a relaxed consistency model. Unfortunately, there is no agreement on where to place the borderline between strong and relaxed consistency models. Depending on the problem being considered, that frontier could identify different sets of strong and relaxed models. For instance, the *linearisable* [34] and *sequential* [45] models are in the strong set (while the remaining models are considered weak) when database one-copy equivalence with serialisable isolation is being considered [32, 24]. On the other hand, Steinke and Nutt [67] qualify the linearisable model as too strong to be specified as a composition of the consistency properties they identify in their work, while the sequential model may be specified without problems (i.e., it is weak enough to be specified). In a similar way, Attiya [5] states that only the linearisable model is strong enough to avoid old-new inversions in read accesses onto shared variables. The sequential model does not avoid that kind of inconsistency in its reads.

In the scope of eventual consistency characterisation, we believe that two frontiers may be defined, separating these three sets of models:

- *Strong models*: A *strong* model is one where state convergence is guaranteed among replicas on each write action. This means that write actions cannot overlap. As a result of this, no read old-new inversion is possible. According to Attiya (2010) [5] the linearisable [34] (or atomic [46]) model is in this strong group.

- *Convergent models*: We consider that a model is inherently *convergent* when it will be able to ensure that once the effects of each write have been propagated to the remaining processes, if no new write action is received in a sufficiently long interval, then the consistency model will ensure (without any other external mechanism) that all replicas have the same state.

  The sequential [45] consistency model is convergent but not strong. For instance, the following execution complies with the requirements of the sequential model, but different readers have been able to read different values between two consecutive writes:

$$W2(x)3, W1(x)2, R3(x)3, W1(y)5, R4(x)3, R4(x)2, R3(x)2, R4(y)5, R2(x)2, R2(y)5, R3(y)5 \quad (1)$$

The first character in each action states the type of action ("R" for reads and "W" for writes), the second character is the process identifier. The argument of the action is the variable on which the action is applied. Finally, the last character expresses the value being written in a write action or returned in a read action. So, W1(x)2 means that process P1 has written value 2 onto variable "x". Read actions express the instant at which a given process has received the effect of a previous write action generated at another process.

The execution is sequentially consistent, since all processes have seen the same sequence of actions (W(x)3, W(x)2 and W(y)5) and such sequence is consistent with the writing order on each writer (in this case, P1 wrote value 2 on "x" before writing 5 on "y"). However, each process has seen the events of that sequence at different times. If we assume that the initial values of both variables are 0, then after the sixth event of that execution, each process holds the following values:

- P1: x=2, y=5.
- P2: x=3, y=0.
- P3: x=3, y=0.
- P4: x=2, y=0.

So, their states do not converge yet. Indeed, they do not converge until all the events in the execution have been considered. At that moment, all processes have got x=2 and y=5.

Additionally, if we considered that P1, P2, P3 and P4 are replicas of a given service, it might happen that a given client read (between the second and third actions) variable "x" from P1, receiving value 2 and later on (e.g., between the third and eighth actions) would read again "x" but from P2, obtaining at that moment value 3, that is older than 2. So, sequential consistency allows read old-new inversions. Therefore, this example shows that the sequential model is convergent but not strong (i.e., not enough strong in the sense stated in this section).

- *Relaxed models*: A model is *relaxed* if it does not ensure convergence when all its consistency requirements are respected. PRAM (also known as FIFO) consistency is an example of relaxed model since it only requires that the writes of each process are applied in writing order on the other replicas, allowing any interleaving of the writes made by different processes. Because of this, different receivers may see different values on a given set of variables when they have applied all their incoming updates.

  For instance, this other FIFO-consistent execution considers the same three write actions and four processes shown in execution 1. However, now the states of those four processes do not converge:

$$W2(x)3, W1(x)2, R3(x)3, W1(y)5, R4(x)2, R3(x)2, R4(y)5, R2(x)2, R2(y)5, R4(x)3, R1(x)3, R3(y)5 \quad (2)$$

  Note that the final state is x=2, y=5 in P2 and P3 but x=3, y=5 in P1 and P4.

Up to our knowledge, only Pascual-Miret [56, 57] has analysed where to place the frontier between relaxed and convergent models. In order to set that borderline we need to revise the consistency properties of all non-strong models. Steinke and Nutt [67] provide an appropriate composable specification of those properties. They are informally summarised as follows:

GPDO (Global Process-Data Order): There is a global agreement on the order of writes at each processor and on the same variable. Writes from different processors or by the same process but on different variables may be freely interleaved by each reader.

GPO (Global Process Order): There is a global agreement on the order of writes at each processor. Writes from different processors may be freely interleaved by each reader.

GDO (Global Data Order): There is global agreement on the order of writes on each variable.

GWO (Global Write-read-write Order): There is a global agreement on the order of potentially causal-related writes; i.e., write A globally precedes write B when the value written in A had been read by process p before it wrote B.

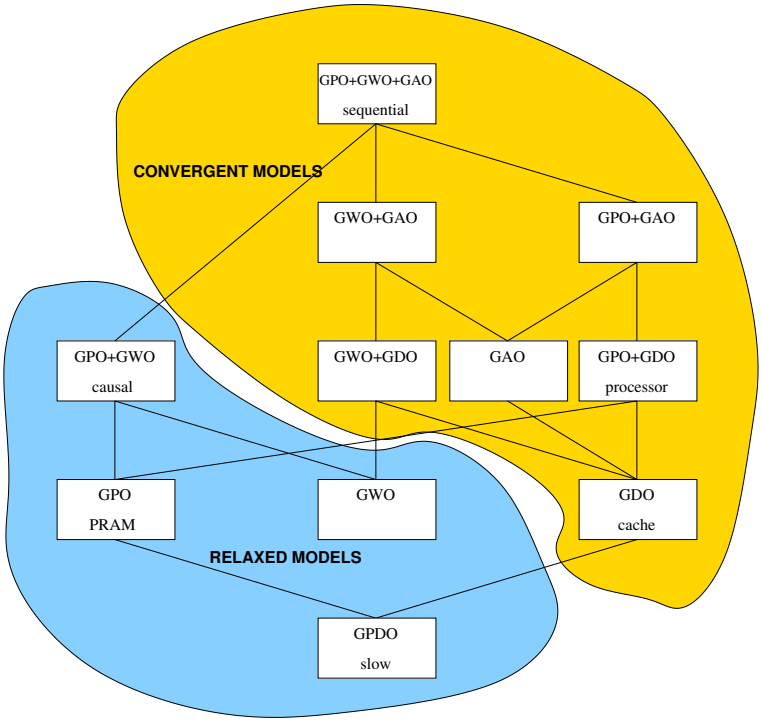

Figure 1: Convergent and relaxed models.

GAO (Global Anti Order): There is a global agreement on the order of any two writes when a process can prove that it read one before the other.

Steinke and Nutt prove [67] that: (1) GPDO defines slow [35] consistency, (2) GPO defines FIFO (or PRAM) [48] consistency, (3) GDO defines cache [31] consistency, (4) GPO+GDO define a model slightly stronger than processor [31] consistency, (5) GPO+GWO define causal [1] consistency, (6) GAO is stronger than GDO; i.e. GAO+GDO is equivalent to GAO, and (7) GPO+GWO+GAO define sequential [45] consistency.

At a glance, GDO is clearly convergent. It states that all processes agree on the order of writes onto each variable. Therefore, when all the updates have been propagated to every replica, the same value on each variable should be seen in all of the replicas. If GDO is convergent, then GAO is also convergent, since GAO is stronger than GDO. Besides, all consistency models that combine GDO (or GAO) with other consistency properties will be also convergent, since GDO implies convergence.

On the other hand, GPO is not convergent. Execution 2 has provided a counter-example for convergence in FIFO executions and GPO is equivalent to FIFO consistency. Additionally, GWO is neither convergent. Execution 2 trivially complies with GWO and it is not convergent. GPDO is not convergent, since it is strictly more relaxed than GPO.

Therefore, GDO is the most relaxed condition that implies convergence. This means that non-strong consistency models may be classified as follows:

- *Convergent*: sequential (GPO+GWO+GAO), processor (GPO+GDO) and cache (GDO).
- *Relaxed*: causal (GPO+GWO), FIFO (GPO) and slow (GPDO).

Figure 1 graphically shows this classification. Each box maintains a combination of properties. Those properties are shown in the top half of the box, while the bottom half shows the name of the resulting consistency model, if any. The basic properties are in the bottom of the figure. Combinations of properties are shown in layers above those taken as their base. The topmost layer corresponds to the sequential consistency model, which is the strongest one that may be built using those consistency properties.

With this, implementations of eventual consistency should choose between two alternatives: (1) to implement a replication protocol supporting a convergent consistency model, or (2) to support a relaxed model and extend its protocol with some data convergence mechanism when data divergences arise.

In the regular case, eventual consistency has been implemented using multi-master replication protocols with lazy update propagation and, in most cases, eventually consistent services are able to tolerate network partitions remaining available. With that kind of replication, convergent models cannot be supported. Note that in a multi-master algorithm multiple processes may concurrently write different values on the same variable (perhaps, in disjoint subgroups of a partition), propagating such values lazily to the other processes. It is impossible to reach an agreement on a common order for the writes applied on each variable (as GDO requires), since every process, besides writing, is also reading the values from other variables. In the end, any possible agreed write order would have been violated by those concurrent reads. Therefore, it is mandatory to take as a base a relaxed model, and complement it with some mechanism that fixes the data divergences that might occur. Additionally, we should also consider that other papers [50, 58] have proved that the causal model is the strongest one to be supported for sharing a general data resource in a consistent and available way in a partitionable network.

Thus, depending on the regular consistency requirements of the service to be implemented following an eventual convergence principle, two real alternatives exist for data-centric consistency: (1) to use slow or PRAM replica consistency (or even no consistency at all) when optimal throughput is the main goal, or (2) to use causal replica consistency when at least a causal behaviour at the client side is expected, partially sacrificing performance in this case. Both approaches should rely on supplementary data convergence mechanisms.

4 Implementation Approaches

The problem of guaranteeing state convergence only arises when a replicated service exists. The state being managed by that service may use an optimistic replication technique [60]. In that case, service replicas become eventually consistent.

In order to implement any replication strategy several aspects should be considered. Let us revise them, focusing on their effect on performance and replica convergence. Note that in each aspect, several implementation approaches are enumerated. After their name, in parentheses, we show an abbreviation in order to refer to them later on:

- *Replication protocol*. The replication protocol rules the steps to be followed by the service replicas in order to manage a given operation submitted by a client agent. Several types of replication protocols may be found:
  - *Primary copy* (PC): One of the server replicas is distinguished as the primary replica. All update operations must be forwarded to the primary, who is the unique replica that may process the updates. Once an operation has been processed by the primary, its effects are collected and forwarded to the remaining replicas that will accept and apply those updates in order to reach convergence with the state of that primary. Primary copy replication protocols follow the *primary-backup* [13] replication model.
  - *Multi-master* (MM): Each operation is processed by a single replica (also known as *master* for that operation service) who later propagates the resulting updates (if any) to the remaining replicas. However, in this case, each operation may be forwarded to any replica. No primary exists in this type of protocol. Thus, concurrent operations may be served by different masters, increasing in this way the degree of concurrency in the service of operations. This strategy might generate inconsistencies if the operations being served by different masters are conflicting.
  - *Quorum-based* (QB): Operations are classified as queries (i.e., read-only operations) or updates (i.e., those operations that create, delete or modify at least one data element in their execution). Queries need to be executed at as many replicas as stated in their *read quorum* (RQ) while updates should reach at least a *write quorum* (WQ). Quorum-based protocols were originally defined [28] for achieving strong consistency. To this end, assuming that there are n replicas

in the system and that each replica has one vote, write quorums should exceed $n/2$ votes and the write quorum plus the read quorum should exceed n votes. In this way, it is guaranteed that conflicting operations will have a non-empty intersection of replicas.

In case of a network partition, when the weight of the write quorum is minimised (i.e., it gets a value of $\lfloor n/2 \rfloor + 1$) [28], a quorum-based protocol is able to enforce strong consistency if a major subgroup exists, forcing in this way the adoption of a *primary partition* [17] model. We have not discussed quorum-based systems in Section 2 since they lead to service unavailability in minor subgroups, preventing eventual consistency techniques from being used in those situations. However, modern NoSQL scalable datastores (e.g., Cassandra [43]) admit multiple quorum sizes. With this, they may adopt varying degrees of consistency.

In order to reach the highest throughput multi-master protocols are the best choice, since they allow the highest concurrency. Non-intersecting quorums (e.g., 1R-1W) are also a good choice. Primary copy protocols might not be enough in case of heavy workloads in scalable systems. On the other hand, regular quorum-based protocols (i.e., those with intersecting quora) provide low throughput, and they have not been used in eventually consistent services.

- *Operation ordering requirements*. Eventually consistent services do not demand state convergence after processing each operation. Instead of this, the state in different replicas diverges at some intervals and will reach again convergence afterwards. This state convergence may depend on the semantics of the operation being provided in the replicated service interface. In some cases, those semantics allow that a given set of operations could be executed in any order at any replica, reaching convergence when the same set has been executed at every replica. Let us analyse which alternatives exist in this area (ordering requirements and operation semantics):
  - *None* (NO). When all the operations in the public interface of a given service are commutative, there is a complete freedom on the order of execution of the incoming requests. Once every replica has executed the same set (i.e., unordered collection) of operations, all those replicas will be convergent. This eliminates the need of inter-replica coordination (e.g., in multi-master replication protocols), being the optimal solution for improving replication throughput.

    CRDTs [64, 65, 66] have been proposed by Shapiro and Preguiça in order to eliminate operation ordering requirements in replicated data types. They provide a useful guide for designing data types with commutative operations and for avoiding conflicts when non-commutative operations exist.

    The *timestamp-based* conflict resolution protocol explained by Johnson and Thomas in [36] also belongs to this class. It uses multi-master replication. Every replica maintains a local clock that is used for building a two-part timestamp. The low-order part is the process identifier. When an updating request is served by a master and the remaining replicas receive its state updates, no ordering coordination is demanded by that protocol. Such propagated update is accepted, or not, depending on a local comparison of the state timestamps.
  - *Partial order* (PO). At a glance, non-commutative operations need to be executed in order to ensure replica convergence. However, even in that case, update operations that might seem conflicting may be executed in any order when they are updating disjoint parts of the shared state. As a result, the global order to be considered becomes partial and this means that concurrent (and unordered) service is tolerated for a subset of the operation requests. In spite of this, partial order means that the other subset of requests must respect a given order and this implies that some degree of coordination among replicas will be needed, reducing the service throughput.
  - *Total order* (TO). In some cases all updating operations are conflicting and they should be executed in a global total order by every replica. This is the regular behaviour in strong consistency protocols and should be avoided if high performance is the main goal.
- *Synchrony in agent interaction*. Distributed services usually follow a client/server interaction pattern. In a strongly consistent service, when a client agent requests an updating operation to a replicated server, these interaction steps may be distinguished:

1. The client sends the request to a master replica.
2. The master replica processes the request.
3. The master replica sends the state updates to every slave replica.
4. Slave replicas acknowledge the completion of the application of those updates on their local copies.
5. The master replica replies to the client.

In that scenario, the client agent remains blocked until the reply is returned to it and the master replica is also waiting for the acknowledgements sent by the slave replicas. This means that strongly synchronous communication is needed in that case. The first primary-backup [3] replication protocols followed that same pattern, ensuring linearisable consistency.

However, in eventually consistent services that level of synchrony is unneeded. For instance, query requests only demand steps 1, 2 and 5, but they can be served by any replica (not necessarily a master or the primary one) and pure updating requests (that return no result) may be completely asynchronous for the client, who will be only involved in step 1. Additionally, in this latter case, a master replica may only execute steps 2 and 3, without needing to wait for step 4 or to do step 5.

On the other hand, when clients expect a reply from their updates, eventual consistency admits lazy update propagation. This means that the serving replica may execute steps 2 and 5 as soon as possible, executing later step 3 in a lazy way (either followed by step 4 or not, depending on the reliability of the communication channels).

Therefore, multiple degrees of agent interaction synchrony are possible in replicated systems. Since query operations always demand a reply, let us centre our attention on how updating operations may be processed. The following alternatives exist:

- *Asynchronous* (A): When updates do not need any reply and server state propagation is done in a lazy way. This is the optimal solution regarding throughput.
- *One synchronous interaction* (1S): When either: (1) the update requires an answer and server state propagation is done in a lazy way, or (2) the update does not need any answer but server state propagation is synchronous.
- *Two synchronous interactions* (2S): When the five steps described above are done in a synchronous way. This only happens in strongly consistent services.

- *State convergence strategy*. The state of different replicas may become divergent from time to time. In those cases, some strategy is needed in order to fix those divergences. That strategy should be able to, first, identify the differences among multiple replicas and, later, decide how to merge those diverging states into a convergent one. The following general alternatives exist to this end:

  - *Unneeded (UN)*. In some applications, state divergences only arise while an updating request that has been processed at a replica is not yet known by the remaining replicas. Once the other replicas receive and process that request, state convergence is restored. This happens, for instance, when all service operations are commutative. Note that in that case, the replication protocol being used should be based on "operation transfer" instead of "state transfer".

    In case of network partitions, each subgroup should remember the set of operations that they have processed while the network was partitioned, in order to transfer that set to the remaining subgroups when connectivity is restored. This allows that every operation executed at every subgroup were considered and accepted in the resulting convergent state. However, that acceptance may depend on the existing semantic constraints in the distributed application.

    For instance, let us assume a banking account that is replicated at multiple sites. Its current balance is 1000$. At a given moment, a network partition occurs and three subgroups (i.e., partitions) of nodes exist. Each subgroup knows that the balance was 1000$. Therefore, all they accept one withdrawal (e.g., of 500$ each). Now, each subgroup thinks that the current balance

is 500$. However, the actual global balance is -500$ and its correct value will be computed when connectivity is recovered. In many cases, an application might have a constraint (e.g., negative balances are not allowed) that would not be respected by that resulting merged state.

This strategy does not need any divergence detection mechanism.

- *Overwrite (OW)*. Some types of simple applications (e.g., directories, calendars, ...) may use databases that hold entries that are seldom updated or that receive updates that need a minimal computation effort and that do not depend on other database fields. In those cases, in case of conflicts, the application is interested in the newest updating attempt. All previous ones may be overwritten by that latest one.

  These applications only need to tag the updating actions with a (logical) timestamp, as it was suggested in [36]. In case of conflicts, the merged state will only hold the newest update. The detection and resolution of conflicts may be easily automated with this strategy.

- *Reordering (RO)*. When the updating operations applied in a concurrent way at different master nodes or partitions depend on the previous state and are conflicting, only one of them might be accepted according to the application semantics. In that scenario, those other conflicting operations should be discarded (using backward recovery in the nodes where they had been previously applied) and restarted. This implies an operation reordering. In some cases, that reordering may be automated if there are deterministic criteria that rule those reordering decisions.

- *Manual convergence (MC)*. If the conflicting operations are discarded and need to be restarted during the state merging procedure but there are no deterministic criteria to schedule those rejected operations, user intervention may be needed for deciding an appropriate schedule order. Therefore a manual merging approach is needed in that case.

From the point of view of performance, the second alternative is the best one, since it only needs to find the newest state and apply it to the remaining replicas. Additionally, the mechanisms needed in that case may be simple (logical timestamps or version vectors). Unfortunately, that strategy is not generally applicable.

Commutative operations also simplify the convergence approaches. Nothing special is needed, although the still missing requests (that could be a large set in case of network partitions that have lasted a long time) should be run in those nodes that had not seen them. This might take a long time in some cases.

Considering only performance, the MM-NO-A-OW (multi-master, with no ordering requirements, asynchronous interactions and overwrite-based merging) or MM-NO-A-UN combinations of strategies seem to be the best ones. Up to our knowledge, no complete performance comparison, including all the identified strategies, has been performed yet. In spite of this, some research papers have analysed and compared some of the alternatives discussed in any of those aspects.

For instance, de Juan-Marín et al (2007) [20] presents a performance evaluation of different primary-copy algorithms depending on their degree of communication synchrony. A completely asychronous interaction (i.e., using the A case) was able to complete an update operation in less than 2 ms (the processing time at the server side was around 1 ms), while the same request demanded at least 25 ms in the 2S case. Therefore, synchronous interactions may worsen communication time up to 20 times in the scenarios being considered in [20].

Golab et al (2014) [30] propose the *gamma* client-centric metric for benchmarking consistency. To this end, the Cassandra scalable datastore is used in [30]. It uses a QB replication protocol. In their tested configurations with a "hotspot" distribution only a "read one-write one" quorum provided a higher proportion of consistency anomalies (1.3%) than the regular strongly consistent quorums (e.g., "read all-write all" or "read a majority-write a majority"), that only reached 0.6%. On the other hand, with its "latest" distribution, the "read one-write one" quorum (1.3% of accesses with anomalies) and the "read one-write a majority" (0.6%) provided more anomalies than regular strongly consistent quorums (0.1%). This means that the most relaxed QB protocols (1R-1W), as expected, introduce more inconsistencies than

Paper	Repl. prot.	Order	Sync.	Converg.	CAP mgmnt.		Year
					C	A	
Johnson and Thomas [36]	MM	NO, PO	A, 1S	OW	FIFO	Y	1975
Bunch [14]	PC	TO	2S	N/A	strong	N	1975
Alsberg and Day [3]	MM	NO	1S	UN	relaxed	Y	1976
Lindsay et al [47]	PC, QB	PO	2S	2PC	strong	N	1979
Parker et al [55]	MM	PO	1S	MC	causal	Y	1981
Apers and Wiederhold [4]	–	PO	1S	RO	strong	N	1985
Ladin et al [41]	MM	PO, TO	1S	UN	causal	Y	1988
Shapiro and Preguiça [64]	MM	NO	A	UN	relaxed	Y	2007

Table 2: Implementation strategies used in several proposals.

the traditional strongly consistent QB protocols. Such difference is up to 13 times greater in the worst case, but it is only twice greater in the common case.

With the aim of complementing our historical revision started in Section 2, Table 2 shows which papers were the first to apply some of the combinations of implementation strategies discussed in this section. In order to consider all possible combinations, some strongly consistent protocols are also in the table.

The table shows how each protocol deals with the CAP theorem when a network partition occurs. In that case, each protocol needs to decide whether consistency is relaxed or availability is sacrificed. In the A column, a Y (yes) means that availability is maintained in every replica, while an N (no) means that some replicas remain unavailable. The C column shows which is the strongest consistency model being supported by the available replicas while the network remains partitioned. The "relaxed" value means that any of the relaxed models identified in Section 3, or even none at all, is enough in that proposal since it is assuming commutative operations.

The entry for Alsberg and Day shows the information about the variant of its protocol that admits multi-master management with commutative operations. The 2PC abbreviation represents the two phase commit protocol explained in [47] in order to decide the fate of distributed transactions. That final protocol introduces two synchronous communication stages at the end of each distributed transaction. Despite tolerating lazy propagation, the algorithms described by Lindsay et al do not admit progress in all subgroups when the network is partitioned.

5 Summary

A discussion on several aspects of eventual consistency has been provided. In its weak form, eventual consistency is basically a liveness property (data convergence) to be added to any relaxed memory consistency model. Due to this, it cannot have a formal specification similar to other consistency conditions since none of them considers time. However, Bouajjani et al [11] have recently proposed a formal specification that carefully characterises safety (program correctness) and liveness (eventual state convergence, even when there are no quiescent intervals in a trace) correctness conditions for eventually consistent services. In the end, the correctness conditions to be achieved depend mainly on [60]: (1) the operation scheduling policy (indeed, the rules of that policy also define the data-centric consistency model to be used: causal, PRAM, slow...), (2) the conflict detection mechanisms, (3) the conflict resolution mechanisms, and (4) the operation commitment procedures. Note that (3) and (4) are defining the state convergence strategy to be followed, and both service network-partition tolerance and throughput will strongly depend on them.

Although eventual consistency has received a lot of attention when elastic and geo-replicated distributed services have been developed, it was already suggested in several papers 40 years ago. Therefore, it is not a new concept. A short historical review has been presented, describing some of the oldest works in this subject.

Finally, using the consistency model specification framework provided by Steinke and Nutt [67], the border between inherently convergent and relaxed models has been set. Those relaxed models (e.g., slow,

PRAM and causal) may be taken as a basis for implementing eventually consistent services. This shows that eventual consistency is quite a relaxed condition.

References

[1] Ahamad M, Burns JE, Hutto PW, Neiger G (1991) Causal memory. In: 5th International Workshop on Distributed Algorithms and Graphs (WDAG), Delphi, Greece, pp 9–30

[2] Almeida S, Leitão J, Rodrigues LET (2013) ChainReaction: a causal+ consistent datastore based on chain replication. In: 8th EuroSys Conference, Prague, Czech Republic, pp 85–98

[3] Alsberg P, Day JD (1976) A principle for resilient sharing of distributed resources. In: 2nd International Conference on Software Engineering (ICSE), San Francisco, CA, USA, pp 562–570

[4] Apers PMG, Wiederhold G (1985) Transaction classification to survive a network partition. Tech. rep., STAN-CS-85-1053, Dept of Comput Sc, Stanford Univ, Stanford, CA, USA

[5] Attiya H (2010) Robust simulation of shared memory: 20 years after. Bull EATCS 100:99–113

[6] Bailis P, Ghodsi A (2013) Eventual consistency today: limitations, extensions, and beyond. Commun ACM 56(5):55–63

[7] Baquero C, Almeida PS, Shoker A (2014) Making operation-based CRDTs operation-based. In: 14th IFIP International Conference on Distributed Application and Interoperable Systems (DAIS), Berlin, Germany, pp 126–140

[8] Bernstein PA, Das S (2013) Rethinking eventual consistency. In: ACM SIGMOD International Conference on Management of Data (SIGMOD), New York, NY, USA, pp 923–928

[9] Birrell A, Levin R, Needham RM, Schroeder MD (1982) Grapevine: An exercise in distributed computing. Commun ACM 25(4):260–274

[10] Bosneag A, Brockmeyer M (2002) A formal model for eventual consistency semantics. In: IASTED International Conference on Parallel and Distributed Computing Systems (PDCS), Cambridge, USA, pp 204–209

[11] Bouajjani A, Enea C, Hamza J (2014) Verifying eventual consistency of optimistic replication systems. In: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, CA, USA, pp 285–296

[12] Brewer EA (2000) Towards robust distributed systems. In: 19th ACM Symposium on Principles of Distributed Computing (PODC), Portland, Oregon, USA, p 7

[13] Budhiraja N, Marzullo K, Schneider FB, Toueg S (1992) Optimal primary-backup protocols. In: 6th International Workshop on Distributed Algorithms and Graphs (WDAG), Haifa, Israel, pp 362–378

[14] Bunch SR (1975) Automated backup. In: Alsberg P (ed) Research in Network Data Management and Resource Sharing; Preliminary Research Study Report, CAC Document Number 162, University of Illinois at Urbana-Champaign, USA, pp 71–106

[15] Burckhardt S (2014) Principles of eventual consistency. Foundations and Trends in Programming Languages 1(1-2):1–150

[16] Burckhardt S, Leijen D, Fähndrich M, Sagiv M (2012) Eventually consistent transactions. In: 21st European Symposium on Programming (ESOP), Tallinn, Estonia, pp 67–86

[17] Chockler G, Keidar I, Vitenberg R (2001) Group communication specifications: a comprehensive study. ACM Comput Surv 33(4):427–469

[18] Cosell BS, Johnson PR, Malman JH, Schantz RE, Sussman J, Thomas RH, Walden DC (1975) An operational system for computer resource sharing. In: 5th ACM Symposium on Operating System Principles (SOSP), The University of Texas at Austin, Austin, Texas, USA, pp 75–81

[19] Davidson SB (1984) Optimism and consistency in partitioned distributed database systems. ACM Trans Database Syst 9(3):456–481

[20] de Juan-Marín R, Decker H, Muñoz-Escoí FD (2007) Revisiting hot passive replication. In: 2nd International Conference on Availability, Reliability and Security (ARES), Vienna, Austria, pp 93–102

[21] Demers AJ, Greene DH, Hauser C, Irish W, Larson J, Shenker S, Sturgis HE, Swinehart DC, Terry DB (1987) Epidemic algorithms for replicated database maintenance. In: 6th ACM Symposium on Principles of Distributed Computing (PODC), Vancouver, BC, Canada, pp 1–12

[22] Dustdar S, Guo Y, Satzger B, Truong HL (2011) Principles of elastic processes. IEEE Internet Comput 15(5):66–71

[23] Fekete A, Gupta D, Luchangco V, Lynch NA, Shvartsman AA (1996) Eventually-serializable data services. In: 15th ACM Symposium on Principles of Distributed Computing (PODC), Philadelphia, PA, USA, pp 300–309

[24] Fekete AD, Ramamritham K (2010) Consistency models for replicated data. In: Charron-Bost B, Pedone F, Schiper A (eds) Replication: Theory and Practice, Lect Notes Comput Sc, vol 5959, Springer, chap 1, pp 1–17

[25] Fischer MJ, Michael A (1982) Sacrificing serializability to attain high availability of data. In: ACM Symposium on Principles of Database Systems (PODS), Los Angeles, CA, USA, pp 70–75

[26] Fox A, Brewer EA (1999) Harvest, yield and scalable tolerant systems. In: 7th USENIX Workshop on Hot Topics of Operating Systems, (HotOS), Rio Rico, Arizona, USA, pp 174–178

[27] Gharachorloo K, Adve SV, Gupta A, Hennessy JL, Hill MD (1992) Programming for different memory consistency models. J Parallel Distrib Comput 15(4):399–407

[28] Gifford DK (1979) Weighted voting for replicated data. In: 7th ACM Symposium on Operating System Principles (SOSP), Pacific Grove, CA, USA, pp 150–162

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

[30] Golab WM, Rahman MR, AuYoung A, Keeton K, Gupta I (2014) Client-centric benchmarking of eventual consistency for cloud storage systems. In: 34th IEEE International Conference on Distributed Computing Systems (ICDCS), Madrid, Spain, pp 493–502

[31] Goodman JR (1989) Cache consistency and sequential consistency. Tech. rep., Number 61, IEEE Scalable Coherent Interface Working Group

[32] Guerraoui R, Garbinato B, Mazouni K (1994) The GARF library of DSM consistency models. In: ACM SIGOPS European Workshop, pp 51–56

[33] Herbst NR, Kounev S, Reussner RH (2013) Elasticity in cloud computing: What it is, and what it is not. In: 10th International Conference on Autonomic Computing (ICAC), San Jose, CA, USA, pp 23–27

[34] Herlihy M, Wing JM (1990) Linearizability: A correctness condition for concurrent objects. ACM Trans Program Lang Syst 12(3):463–492

[35] Hutto PW, Ahamad M (1990) Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In: 10th IEEE International Conference on Distributed Computing Systems (ICDCS), Paris, France, pp 302–309

[36] Johnson PR, Thomas RH (1975) The maintenance of duplicate databases. RFC 677, Network Working Group, Internet Engineering Task Force

[37] Kawell L Jr, Beckhardt S, Halvorsen T, Ozzie R, Greif I (1988) Replicated document management in a group communication system. In: ACM Conference on Computer-Supported Cooperative Work (CSCW), Portland, Oregon, USA, pp 395–

[38] Krishnakumar N, Bernstein AJ (1991) Bounded ignorance in replicated systems. In: 10th ACM Symposium on Principles of Database Systems (PODS), Denver, Colorado, USA, pp 63–74

[39] Kumar A, Stonebraker M (1988) Semantics based transaction management techniques for replicated data. In: ACM SIGMOD International Conference on Management of Data (SIGMOD), Chicago, Illinois, USA, pp 117–125

[40] Kumar P, Satyanarayanan M (1993) Log-based directory resolution in the Coda file system. In: 2nd International Conference on Parallel and Distributed Information Systems (PDIS), San Diego, CA, USA, pp 202–213

[41] Ladin R, Liskov B, Shrira L (1988) A technique for constructing highly available services. Algorithmica 3:393–420

[42] Ladin R, Liskov B, Shrira L (1990) Lazy replication: Exploiting the semantics of distributed services. In: 9th ACM Symposium on Principles of Distributed Computing (PODC), Quebec City, Quebec, Canada, pp 43–57

[43] Lakshman A, Malik P (2010) Cassandra: a decentralized structured storage system. Operating Systems Review 44(2):35–40

[44] Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565

[45] Lamport L (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE T Comput 28(9):690–691

[46] Lamport L (1986) On interprocess communication. Distrib Comput 1(2):77–101

[47] Lindsay BG, Selinger PG, Galtieri CA, Gray JN, Lorie RA, Price TG, Putzolu F, Traiger IL, Wade BW (1979) Notes on distributed databases. Tech. rep., RJ2571(33471), IBM Research Laboratory, San Jose, CA, USA

[48] Lipton RJ, Sandberg JS (1988) PRAM: A scalable shared memory. Tech. rep., CS-TR-180-88, Princeton University, USA

[49] Lynch NA, Tuttle MR (1987) Hierarchical correctness proofs for distributed algorithms. In: 6th ACM Symposium on Principles of Distributed Computing (PODC), Vancouver, Canada, pp 137–151

[50] Mahajan P, Alvisi L, Dahlin M (2011) Consistency, availability and convergence. Tech. rep., UTCS TR-11-22, The University of Texas at Austin, USA

[51] Mockapetris PV (1983) Domain names - concepts and facilities. RFC 882, Network Working Group, Internet Engineering Task Force

[52] Mockapetris PV (1983) Domain names - implementation and specification. RFC 883, Network Working Group, Internet Engineering Task Force

[53] Mosberger D (1993) Memory consistency models. Operat Syst Review 27(1):18–26

[54] O'Neil PE (1986) The escrow transactional method. ACM Trans Database Syst 11(4):405–430

[55] Parker DS, Popek GJ, Rudisin G, Stoughton A, Walker BJ, Walton E, Chow JM, Edwards DA, Kiser S, Kline CS (1981) Detection of mutual inconsistency in distributed systems. In: Berkeley Workshop, pp 172–184

[56] Pascual-Miret L (2014) Consistency models in modern distributed systems. An approach to eventual consistency. Master's thesis, Depto. de Sistemas Informáticos y Computación, Univ. Politècnica de València, Spain

[57] Pascual-Miret L, Muñoz-Escoí FD (2016) Replica divergence in data-centric consistency models. In: XXIV Jornadas de Concurrencia y Sistemas Distribuidos (JCSD), Granada, Spain

[58] Pascual-Miret L, González de Mendívil JR, Bernabéu-Aubán JM, Muñoz-Escoí FD (2015) Widening CAP consistency. Tech. rep., IUMTI-SIDI-2015/003, Univ. Politècnica de València, Valencia, Spain

[59] Pu C, Leff A (1991) Replica control in distributed systems: An asynchronous approach. In: ACM SIGMOD International Conference on Management of Data (SIGMOD), Denver, Colorado, USA, pp 377–386

[60] Saito Y, Shapiro M (2005) Optimistic replication. ACM Comput Surv 37(1):42–81

[61] Sarin SK, Blaustein BT, Kaufman CW (1985) System architecture for partition-tolerant distributed databases. IEEE Trans Computers 34(12):1158–1163

[62] Satyanarayanan M, Kistler JJ, Kumar P, Okasaki ME, Siegel EH, Steere DC (1990) Coda: A highly available file system for a distributed workstation environment. IEEE Trans Computers 39(4):447–459

[63] Schneider FB (1990) Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput Surv 22(4):299–319

[64] Shapiro M, Preguiça NM (2007) Designing a commutative replicated data type. Tech. Rep. RR-6320, INRIA, Rocquencourt, France

[65] Shapiro M, Preguiça NM, Baquero C, Zawirski M (2011) Conflict-free replicated data types. In: 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Grenoble, France, pp 386–400

[66] Shapiro M, Preguiça NM, Baquero C, Zawirski M (2011) Convergent and commutative replicated data types. Bull EATCS 104:67–88

[67] Steinke RC, Nutt GJ (2004) A unified theory of shared memory consistency. J ACM 51(5):800–849

[68] Tanenbaum AS, van Steen M (2007) Distributed Systems - Principles and Paradigms (2nd ed.). Pearson Education

[69] Terry DB, Demers AJ, Petersen K, Spreitzer M, Theimer M, Welch BB (1994) Session guarantees for weakly consistent replicated data. In: 3rd International Conference on Parallel and Distributed Information Systems (PDIS), Austin, Texas, USA, pp 140–149

[70] Vogels W (2008) Eventually consistent. ACM Queue 6(6):14–19

[71] Vogels W (2009) Eventually consistent. Commun ACM 52(1):40–44

[72] Walker BJ, Popek GJ, English R, Kline CS, Thiel G (1983) The LOCUS distributed operating system. In: 9th ACM Symposium on Operating System Principles (SOSP), Bretton Woods, New Hampshire, USA, pp 49–70

[73] Yu H, Vahdat A (2000) Design and evaluation of a continuous consistency model for replicated services. In: 4th Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, USA, pp 305–318