Computing and Informatics, Vol. 37, 2018, 1037–1072, doi: 10.4149/cai_2018_5_1037

EVENTUAL CONSISTENCY: ORIGIN AND SUPPORT

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

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
46022 Valencia, Spain
e-mail: {fmunyoz, rgarcia, jsendra, josep}@iti.upv.es

José Ramón González de Mendívil

Departamento de Ingeniería Matemática e Informática
Universidad Pública de Navarra
31006 Pamplona, Spain
e-mail: mendivil@unavarra.es

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 several aspects of eventual consistency that have not been analysed in depth in previous works: 1. which are the oldest replication proposals providing eventual consistency, 2. which replica consistency models provide the best basis for building eventually consistent services, 3. which mechanisms should be considered for implementing an eventually consistent service, and 4. which are the best combinations of those mechanisms for achieving different concrete goals. This paper provides some notes on these important topics.

Keywords: Eventual consistency, consistency model, CAP theorem, data replication

Mathematics Subject Classification 2010: 68-03, 68M14, 68N01, 68U35, 68W15

1 INTRODUCTION

Eventual consistency [64] has received a lot of attention in the last decade due to the emergence of elastic distributed services. Elastic services [31] 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 [28], when a network partition happens, there is a trade-off 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 [56, 64, 60, 8, 13, 54], but 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 aspects. They will be thoroughly analysed in the following sections that are introduced hereafter.

The first point of interest is a historical review. Although recent research works cite a paper from Werner Vogels [64] as the most well-known reference explaining this kind of consistency, there are many papers older than [64] which have either explained this same concept or implemented this consistency. Indeed, the oldest eventually consistent systems were proposed in the 70s of the past century. Those proposals tried to solve some problems that are close to those being solved nowadays: how to improve the performance and availability of scalable services, providing an acceptable level of consistency among their server instances. Some of these systems and solutions are revised in Section 2.

Section 3 sets the border between models that are inherently convergent and those others that are too much relaxed to be convergent *per se*. Inherently convergent replication models cannot be globally maintained when the network is partitioned. Eventual consistency requires that replica convergence is achieved when no new updates are received for a sufficiently long interval. This implies that *relaxed* models (e.g., FIFO/PRAM [45] or causal [1]) must be taken as a basis to develop eventually consistent services.

Section 4 explains how to implement an eventually consistent replicated service. Four complementary aspects are considered:

1. replication protocol,
2. operation ordering,
3. synchrony in agent interaction, and
4. state merging strategy for reaching convergence.

There are multiple alternatives in each aspect and, not surprisingly, some of the best combinations regarding performance and convergence were already known long time ago. However, depending on the concrete goals of an eventually consistent service, other new approaches may be needed and Section 4.2 revises those requirements.

The principles that are needed to manage eventually consistent services had been already used in several of the oldest distributed services. The challenges at that time were centred in providing acceptable response times and consistency using very limited computers and networks. Current hardware resources are much more powerful, but current services have also much more demanding requirements; e.g., they should be immediately adaptive [50]. So, although those old principles may guide our research efforts in this kind of dynamic consistency, other new contributions are still required in this area.

2 HISTORICAL REVIEW

Quoting Vogels [64], 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, 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 [47, 48], proposed in 1983. Because of this, the reader may realise that traditional scalable distributed services have usually been eventually consistent. Vogels provides another pointer to a former publication on this subject: Lindsay et al. (1979) [44]. 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 [44], 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 syn-

chronisation being demanded by transactions, allowing their fast completion. Thus, this is one of the first examples of consistency-performance trade-off.

Assuming that the usage of relaxed consistency has been a common 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, analysing the challenges that were solved by each one of those proposals.

To this end, we may start considering the first reference on primary-backup replication [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. When this algorithm is followed, the consistency being perceived is strong (indeed, linearisable [32]). In spite of this, Alsberg and Day also outline some variations of their basic algorithm. Thus, they also explain 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 their 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.*"

It assumes that 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, three years before the relaxed algorithm found in [44].

Moreover, Alsberg and Day [3] cite two related papers to look for additional information [12, 34]. Bunch [12] describes a preliminary version of the primary-backup 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 non-critical. Critical reads are directly managed by the primary replica. Non-critical 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. However, these non-critical reads

settled the basis for the relaxed consistency algorithm described in [44], once the synchronous update propagation was replaced with a lazy forwarding.

On the other hand, Johnson and Thomas [34] 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 [16]). Each database copy is held by a *database management process* (DBMP). The updates applied in a given master replica should be transferred to the remaining replicas. A list of pending replicas is maintained and the algorithm tolerates lazy propagation. 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 [41], 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. Indeed, Johnson and Thomas mention the following regarding service continuity in case of network partitions: "*. . . 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 trade-off 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, 28].

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 [65] distributed file system described by Parker et al. [52] in 1981. Its designers took care of handling network partitions, allowing

progress in disconnected subgroups of nodes. They also mentioned the trade-off 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 depend on the semantics of the operations being applied. 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 [34].

Fischer and Michael [25] describe an evolution of the algorithms presented in [34] 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.

Davidson (1984) [17] 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 (1985) [4] 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, transactions are classified as:

1. unconditionally committable (UC), when their execution cannot violate any precondition of other transactions run in other partitions,
2. conditionally committable (CC), when their acceptance cannot be guaranteed but their possible afterwards rejection will not introduce other consistency problems, and
3. non-committable (NC), when their possible afterwards 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, when a system is partitioned, of managing semantic correctness criteria at partition reconnection time and of using conditional criteria for accepting some classes of transactions.

Other semantic criteria for managing state merging at partition reconnection time were proposed by Sarin et al. (1985) [57]. 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.

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 rumour 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 (Kawell et al. [35]) was based on lazy update propagation and on the "last writer wins" policy for dealing with concurrent updates. The database being managed was unconven-

tional: 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). On each update, a complete document should be transferred among replicas. Fortunately, documents were small (usually 1 or 2 KB). Those updates were transferred following a pull policy. Notes assumed that computers are not continuously connected to the network. 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.

Kumar and Stonebraker (1988) [37] describe how to apply the *escrow* [51] 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.

In the *lazy replication* (1990) approach [39] proposed by Ladin et al., 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, the client specifies 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, and it defines a border for ensuring that 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 de-

lays 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 [58] (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 [38].

In the scope of replicated relational databases, Krishnakumar and Bernstein (1991) [36] 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, with N being the 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 [55] proposed 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 [55]: 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 repli-

cation protocol does not allow any divergence among the states of replicas. Therefore, ORDUP is inherently convergent.

Bayou (1994) [62] 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; and

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 [62] for complying with the consistency guarantees required in sessions.

Fekete et al. (1996) [22] provide the first formal specification of an eventually consistent system. Their proposal formalises the algorithms described in [39]. 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) [66] 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) [56] provide a survey on *optimistic replication*; i.e., replication techniques that relax their concurrency control and consistency in order to achieve greater efficiency 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: 1. at any moment, for each replica, there is a (committed) prefix of the schedule that is equivalent to a prefix of the schedule of every other replica; 2. the committed prefix of each replica grows monotonically over time; 3. all non-aborted operations in the committed prefix satisfy their preconditions; and 4. for every submitted operation* α*, either* α *or* $\not\alpha$ *will eventually be included in the committed prefix.*"

The equivalence of *committed prefixes* among replicas allows the modelling of state convergence when no new updates are received. Their monotonic 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. The last condition means that in order to reach convergence and reconciliate from existing conflicts, some of the submitted operations may be discarded (i.e., $\not\alpha$); for instance, applying other compensating actions 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. Additionally, operations should be ordered in a way expected by users. Five ordering alternatives exist:

1. 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),
2. commutative operations, allowing any execution order (no conflict appears but it has a limited applicability),
3. canonical ordering (concurrent operations are ordered following an application-dependent set of rules),
4. 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
5. semantic optimisation (again, too complex).

Conflicts refer to how state conflicts are dealt with and resolved. Two alternatives:

1. 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
2. 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 nondeterministic decisions. There are three alternatives:

1. implicit (no operation is ever explicitly rejected; this may be supported using the "last writer wins" approach in case of conflicts),
2. background agreement (nodes send piggybacked information about their accepted updates in their update propagation messages; e.g., using version vectors), and
3. consensus (this is a complex and potentially blocking solution, usually needed in strongly consistent systems but not recommended in eventual ones).

One of the conclusions that may be extracted from [56] is that concurrency control could be avoided in eventually consistent systems. To this end, convergent and commutative replicated data types (CRDTs) [60] were proposed by Shapiro et al. as a set of replicated data types able to easily converge by themselves without needing any concurrency control mechanism. CRDTs are based either on commutative operations (CmRDTs), giving some rules, advices and examples for transforming regular operations into other commutative variants, or on a *merge* operation for state-based replication protocols (CvRDTs), introducing commutativity at update application time in those protocols. Thus, in both cases, the system achieves convergence ensuring that all incoming client requests are eventually executed by every replica.

This historical review would end here. Its goal has been to show that some research work on eventual consistency existed before Vogels' paper [64] was written. We have centred our discussion on the papers written before 1994, since subsequent papers have been thoroughly reviewed in other works; e.g., by Saito and Shapiro [56]. A summary of the contributions found in our 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: a formal specification for eventual consistency. There have been a few papers covering that goal [22, 13, 10] and the two latter deserve some comments since they have provided valuable contributions.

Bouajjani et al. [10] and Burckhardt [13] criticise that almost all previous definitions of eventual consistency had been only centred in state convergence at quies-

Reference	Contributions	Year
Johnson and Thomas [34]	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. [44]	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. [52]	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 [17]	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. [57]	Service continuity in partitioned databases. Semantic criteria for reordering or avoiding compensating actions at partition reconnection.	1985
Demers et al. [21]	Analysis of three types of lazy update propagation.	1987
Kawell et al. [35]	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 [37]	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. [58]	Support for client disconnected operation in distributed filesystems. Manual multi-conflict resolution may be demanded at reconnection time.	1990
Ladin et al. [39]	Specification of update ordering requirements, depending on application semantics. Explicit causal dependences as a base for building eventual consistency. Global order for reaching convergence on the set of processed operations in every replica.	1990
Krishnakumar and Bernstein [36]	N-ignorance as an eventually consistent example in replicated relational databases. Lazy writeset propagation with causal order.	1991
Pu and Leff [55]	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. [62]	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. [22]	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 [39].	1996
Yu and Vahdat [66]	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 [56]	Survey on optimistic replication techniques, including eventual consistency. Thorough classification of eventually consistent systems.	2005
Bouajjani et al. [10]	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

cence 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 [10] 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 [49] assume that multiple processors share a given memory and that processes directly run in those processors. Thus, those models are focused on how data is shared by processes. Those classical memory consistency models are known nowadays as data-centric consistency models and they are also used in distributed systems [61]. In this section, the term *consistency model* will always refer to those data-centric models.

The state convergence assumed in eventual consistency is a liveness property [23, 13, 54]. Therefore, the conditions required in several definitions of eventual consistency [56, 64] could be respected adding a convergence property to a relaxed consistency model. However, there is no agreement on where to place the borderline between strong and relaxed consistency models. Depending on the problem being considered, that frontier separates different sets of strong and relaxed models. For instance, the *linearisable* [32] and *sequential* [42] models are in the strong set when database one-copy equivalence with serialisable isolation is being considered [23]. On the other hand, Attiya [6] 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, two frontiers should be defined separating these three sets of models [54]:

Strong models: A *strong* model guarantees state convergence 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. The linearisable [32] (or atomic [43]) model is in this strong group [6].

Convergent models: A model is inherently *convergent* when the conditions that define the model imply state convergence. In this case write actions may overlap and read old-new inversions may happen when those read actions occur in different replicas. However, once all value propagations for concurrent write actions are delivered, the state in all replicas reaches convergence again.

Relaxed models: A model is *relaxed* if it does not guarantee convergence when all its consistency conditions are respected. 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.

Section 2 has shown that eventual consistency was generated as a reaction to the constraints of the CAP theorem [28]. Being eventually consistent, a service may remain available to all its clients even when the network that interconnects its replicas gets partitioned. Thus, inherently convergent models are all those on which the CAP constraints may apply; i.e., those that require consensus on the order of writes among all participating processes [54], since that consensus cannot be reached in an asynchronous distributed system when its processes may fail or get disconnected [24]. That requirement already applies to the *cache* [30] consistency model and all those stricter than cache. However, no need of consensus exists in the *slow* [33], *FIFO/PRAM* [45] and *causal* [1] models.

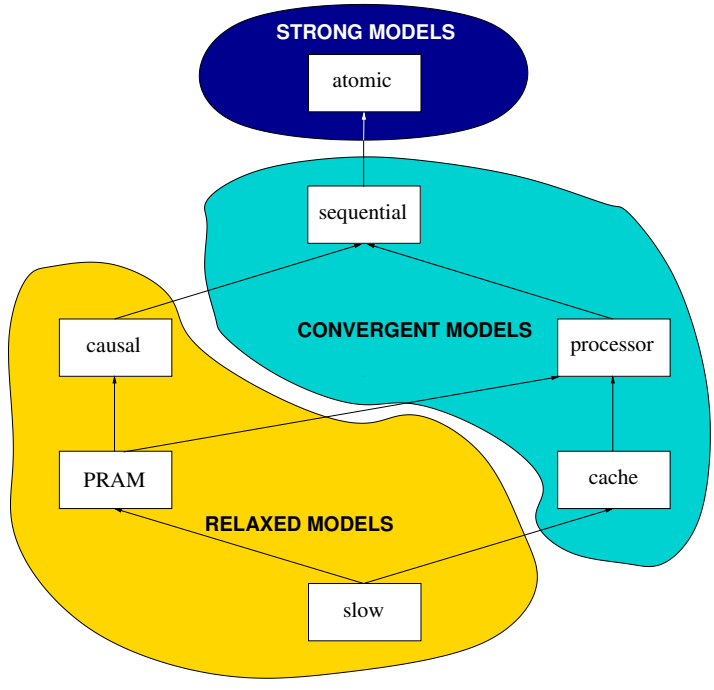

Figure 1. Strong, convergent and relaxed models

Figure 1 shows the *weaker-than* ($\longrightarrow$) relations among data-centric models. They were already identified by Mosberger [49] in 1993. Taking those relations as a base, it is easy to select which consistency models belong to each set, defining in this way the intended borders.

These identified frontiers have a first consequence: all inherently convergent models (i.e., sequential, processor, cache) are stronger than *eventual consistency*. Note that a model A is stronger than another model B if all executions accepted by model A comply with the constraints of model B and there is some execution that respects model B but does not respect the constraints imposed by model A [63]. Inherently convergent models are eventually consistent, since they reach convergence once a given set of write actions is known and applied by every process. However, not all eventually consistent executions are inherently convergent. For instance, the following execution:

$$W2(x)3, W1(x)2, R3(x)3, R4(x)2, R3(x)2, R2(x)2,$$
$$R4(x)3, R1(x)3, W4(x)5, R2(x)5, R3(x)5, R1(x)5 \quad (1)$$

... is eventually consistent, since at the end all its four processes converge to value $x = 5$; but it is not sequential, nor processor, nor cache. Its strongest respected model is causal. Processes P2 and P3 have seen the sequence 3, 2, 5 on x and P1 and P4 have seen the sequence 2, 3, 5. When the written values 2 and 3 had been delivered at all processes, before value 5 is written, the state of these four processes did not converge. Therefore, it is not inherently convergent.

Up to our knowledge, this relationship among inherently convergent models and eventual consistency has not been explicitly identified yet. Indeed, Viotti and Vukolić [63] (2016) revise many distributed consistency models, trying to identify their relations in order to build a global hierarchy. In that hierarchy, they distinguish multiple variants of eventual consistency: basic eventual [13], strong eventual [13], eventual serialisability [22] and eventual linearisability. They only state that the *strong eventual* and the *eventual linearisability* models, being the strongest in that set, are weaker than the *linearisable* model. No relation is identified among all those four variants of eventual consistency and other data-centric models (besides *linearisable* consistency). However, our Execution 1 illustrates that the *basic eventual* model is weaker than *cache* consistency. Additionally, *cache* executions comply with the requirements of the *strong eventual* model (to reach convergence once the same set of update operations are delivered to all processes), and Execution 2 is an example of execution based on commutative operations, assuming an initial value of zero for variable x in every process, that is *strong eventual* but not *cache* consistent (P2 and P3 have observed sequence 3, 5 since they have received the operations in the order $+3$, $+2$; while P1 and P4 have observed sequence 2, 5, since they received those operations in the order $+2$, $+3$). This means that the *strong eventual* model is weaker than *cache* consistency. Therefore, these relations exist: *basic eventual* $\longrightarrow$ *strong eventual* $\longrightarrow$ *cache* $\longrightarrow$ *processor* $\longrightarrow$ *sequential* $\longrightarrow$ *linearisable*

(atomic). Thus, there are several models between the *strong eventual* to *linearisable* weaker-than relation.

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

Regarding network partitions, our proposed frontier among convergent and relaxed models has another interesting consequence. Fox and Brewer [26] stated the CAP theorem assuming that "consistency" was atomic (i.e., strong). Gilbert and Lynch [28] proved that same theorem respecting that assumption. However, Pascual-Miret et al. [53] have proven that even cache consistency cannot be respected by available replicated services while the network is partitioned. Their proofs are quite complex. A more intuitive proving argument of that fact has been already given here for setting the borderline between convergent and relaxed models. *Cache* consistency requires agreement on the order of writes on each variable. That agreement requires consensus among all the processes that replicate shared variables. Consensus cannot be achieved in a system where processes may fail [24] (or where they remain unreachable due to a network partition). Therefore, the CAP theorem not only applies to atomically consistent services but to all replicated services that are strong or convergent according to our classification.

Depending on the consistency requirements of the service, two alternatives exist for implementing eventual consistency in a system that tolerates network partitions:

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 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 [56] because in this way it reduces overall overhead and tolerates network partitions. In that case, service replicas become eventually consistent.

At a glance, this means that a replication protocol should be chosen, and its optimism may rely on lazy update propagation, introducing some degree of asynchrony in the interactions among replicas. However, other characteristics of inter-replica management may be relevant. Thus, a careful evaluation of those characteristics is needed.

In order to provide a characterisation of the existing implementation approaches, Section 4.1 analyses which elements should be considered to define an implementation strategy for eventual consistency. With that base, Section 4.2 provides a performance analysis of those implementation strategies. Later, Section 4.3 summarises

which systems were the first using each one of the implementation strategies identified in Section 4.1, complementing in this way the historical review presented in Section 2.

4.1 Four Elements to Implement Eventual Consistency

Several aspects should be considered to implement any replication strategy. Let us revise them, focusing on their effect on performance and replica convergence. 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 that 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* [11] 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, operations may be served by different masters, increasing in this way the degree of concurrency. 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). 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 [27] for achieving strong consistency. To this end, assuming that there are N replicas in the system and that each replica has one vote, two rules should be respected:

1. $\mathrm{WQ} > N/2$, and
2. $\mathrm{WQ} + \mathrm{RQ} > N$ votes.

In this way, it is guaranteed that conflicting operations will have a non-empty intersection of replicas. These traditional quorum-based protocols may be tagged as *strict quorum* protocols (QB_S) [7].

Modern NoSQL scalable datastores (e.g., Cassandra [40]) admit multiple quorum sizes, allowing varying degrees of consistency. Indeed, they are able

to provide relaxed eventual consistency when quorums do not need to intersect for accepting an operation [29]. This second kind of quorum-based protocols can be tagged as *partial quorum* protocols (QB_P) [7].

Operation ordering. 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 operation semantics and ordering. Let us analyse which alternatives exist in this area:

None (NO). When all the operations in the 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 MM replication protocols), being the optimal solution for improving replication throughput. Commutative replicated data types (CRDTs) [59, 60], also known as *conflict-free replicated data types*, 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.

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, some degree of coordination among replicas is 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.

Synchronisation degrees 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, five 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; and
5. the master replica replies to the client.

Thus, the client agent remains blocked until the reply is returned to it. The master replica is also waiting for the acknowledgements sent by the slave replicas. This means that strongly synchronous communication is needed. 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 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 being involved in steps 4 and 5.

On the other hand, when clients expect a reply from their updates, eventual consistency admits lazy update propagation. Thus, 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 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.

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 a collection of independent elements that are seldom updated. Additionally, the computational effort for those updates is minimal. In those cases, when conflicts arise, 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 [34]. 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, the reordering may be automated if there are deterministic criteria that rule those reordering decisions.

Manual convergence (MC). In some cases, there are no deterministic criteria to schedule conflicting operations that were rejected in the state merging procedure. Therefore a manual merging approach is needed in that case.

From the point of view of performance, the second alternative (OW) 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 missing requests should be run in those nodes that had not seen them. This might take a long time in case of prolonged network partitions.

4.2 A Guide for Performance Analysis

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 identified strategies, has been made 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) [19] presents a performance evaluation of different primary-copy algorithms depending on their degree of communication synchrony. A completely asynchronous 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. In the scenarios considered in [19], synchronous interactions may worsen communication time up to 20 times.

Golab et al. (2014) [29] propose the *gamma* client-centric metric for benchmarking consistency. To this end, the Cassandra scalable datastore is used in [29]. It uses a QB replication protocol. In their tested configurations with a "hotspot" distribution, only a "read one-write one" (RQ:1, WQ:1) quorum provided a higher proportion of consistency anomalies than strict quorums (e.g., "read all-write all" or "read a majority-write a majority"); 1.3 % vs. 0.6 %. On the other hand, with its "latest" distribution, the "read one-write one" quorum and the "read one-write a majority" (RQ:1, WQ>$N/2$) provided more anomalies than strongly consistent quorums (1.3 % and 0.6 %, respectively, versus 0.1 %). This means that the most relaxed QB protocols (RQ:1, WQ:1), as expected, introduce more inconsistencies than 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.

Bailis et al. (2014) [7] propose eventual consistency with probabilistic bounded staleness (PBS). Like [29], PBS uses QB_P protocols. Since quorum-based protocols use a read quorum for handling their read accesses, they may obtain up to RQ different values in each read. So, PBS provides (K, Δ, p)-*regular semantics* [63]. Bailis et al. evaluate the effects on both response time and consistency of different system configurations. Thus, when response time is assessed using a workload model that represents a peak interval in the LinkedIn servers (an example of IO-bound workload) in a system with three replicas with $p = 99.9\,\%$, the following values are obtained for different quorum sizes:

- $RQ = 1,\ WQ = 1$: Both read and write latencies of 0.66 ms, but requiring 1.85 ms for ensuring the intended (Δ,p)-regular semantics; i.e., this introduces a non-negligible staleness interval.
- $RQ = 2,\ WQ = 1$: Read latency of 1.63 ms, with write latency of 0.65 ms, but without any staleness interval in spite of being a QB_P configuration.
- $RQ = 2,\ WQ = 2$: Read latency of 1.62 ms, with write latency of 1.64 ms, and without staleness, since it is the smallest QB_S configuration.

- $RQ = 1$, $WQ = 3$: Read latency of 0.65 ms, with write latency of 4.09 ms, without staleness. This is the standard ROWA configuration, the recommended QB_S deployment for read-intensive applications.

That evaluation shows that those QB_P configurations which minimise response time are subject to returning stale values. However, there are other QB_P configurations that do not read stale values and still provide better response times than the most efficient QB_S configuration. Let us call *overall latency* the sum of both read and write latencies. QB_P configurations have shown an overall latency between 1.32 and 2.28 ms, while QB_S ones have demanded between 3.26 and 4.74 ms. This shows that QB_P has been, as a minimum, 43 % faster than QB_S (and more than N times faster in the best case) and, in some cases, they have avoided staleness in their read values.

These partial evaluations could be extended. Section 4.1 has identified the four aspects to consider in every implementation of eventual consistency. Each aspect provides multiple implementation approaches (3 in the general case, but there are 4 state convergence strategies) and this provides a total of $3^3 \cdot 4 = 108$ combinations.

Assuming this diversity of possible implementation approaches, developers of scalable services need some advice on the best choice in each implementation aspect. Those advices depend on which are the main goals for that service and on the priorities given for each objective. Although each application may have its specific goals, there are some common objectives for all of them. They are:

Minimal response time (mRT). Services that use eventual consistency try to optimise their service time and the response time being perceived by their clients.

Minimal convergence effort (mCE) when no disconnection arises. When no network partition happens, this goal measures the effort being required by the assessed mechanism in order to reach inter-replica convergence.

Minimal convergence recovery time (mCT) once a network partition is healed. This depends on the communication rounds being needed for transferring any missed state, and on the concrete protocol to be used for joining that missed state with the local state on each replica.

Minimal transfer size (mTS) once a network partition is healed. The size of the state to be transferred (or the amount of operations to be propagated) conditions the time being needed for recovering convergence. It is a critical goal for mobile computers; e.g., laptops using CSCW applications [35].

Depending on a given prioritised group of goals, programmers need a guide about which combination of implementation approaches on each aspect is the best one for achieving them. At the moment, that guide is missing but it could be written once some prototypes of the approaches identified in Section 4.1 were deployed and evaluated. In order to partially fill this gap, an overall evaluation of those approaches is given in Table 2. Four levels of achievement are considered: irrelevant (×), poor (–), moderate (+), and good (++). A mechanism is qualified as *irrelevant* (×) when

its effect is null on a given objective. This means that it is not directly related with that goal.

Axis and Mechanism	**Goals**			
	mRT	**mCE**	**mCT**	**mTS**
Replication protocol				
Primary copy	+	++	++	×
Multi-master	++	–/+	+	×
Quorum-based (strict)	–	–	×	×
Quorum-based (partial)	++	–/+	+	×
Operation ordering				
None	++	+	×	×
Partial order	+	+	×	×
Total order	–	+	×	×
Synchronisation degree				
Asynchronous	++	+	×	×
One-synchronous	+	+	×	×
Two-synchronous	–	+	×	×
Convergence strategy				
Unneeded	++	++	+	–
Overwrite	++	+	++	++
Reordering	–	×	–	–
Manual convergence	–	×	–	

Table 2. Goals achievement level for each implementation approach

Regarding the mCE goal, there is no difference among the operation ordering and synchronisation degree variants because in both cases there are two aspects to consider: the degree of achieved convergence and the effort being needed. Thus, in the operation ordering axis, when the achieved convergence is considered, the marks are: no order (–), partial order (+), total order (++), since when every replica applies all the operations in total order, convergence is trivially achieved; but when the needed effort is taken into account, the marks are just the contrary: no order (++), partial order (+), total order (–), since total order requires multiple rounds of message exchange for being achieved [18], while "no order" does not require any effort. So, all the alternatives receive a moderate (+) global mark. The same happens in the synchronisation degree axis.

Considering what is shown in Table 2, the most recommendable implementation approaches for covering each goal are:

1. *Minimal response time* (mRT): MM-NO-A-UN, MM-NO-A-OW, QB$_P$-NO-A-UN, QB$_P$-NO-A-OW.

   Regarding the replication protocol, MM and QB$_P$ are the best approaches since they do not require any coordination among concurrently served requests. Thus, they tolerate high concurrency without introducing any inter-request blocking.

Therefore, their mark is good (++). To this end [3], the operations being managed should be commutative. PC may be also combined with lazy replication but compel every operation request to be forwarded to the same primary replica. In the end, with heavy workloads, this might collapse such single replica, while that same workload could be shared out among multiple masters in MM. This explains why the PC mark is lower (+) than the MM and QB_P ones. QB_S totally order conflicting write operations, using to this end its quorum-based concurrency control algorithm. Those algorithms provide higher response time than QB_P (as shown in [7]) and are prone to deadlocks that may be avoided using some kind of priority management among the competing concurrent lock requestors. However, that management introduces either:

(a) the need of additional communication [46] for deciding whether a given low priority vote may still be changed or not, or
(b) the abortion of low priority requests in case of conflict [14].

In the end, this introduces a non-negligible overhead that penalises the response time being perceived by client processes.

The NO and A convenience in the second and third axes has already been explained.

Finally, considering the convergence strategy, both RO and MC have a poor (–) mark since the former needs to cancel and reapply operations to implement its reordering and the latter requires human intervention. In both cases, the time interval needed for those tasks becomes long, although RO automates its tasks, being slightly better than MC. On the other hand, UN is based on non-conflicting operations. So, no concurrency control mechanism is needed and no operation abortion or rollback may arise [59, 60, 2]. This deserves a good (++) mark. Such mark is shared by OW in case of objects of small size, since it also avoids operation re-execution and immediately determines whether an incoming state propagation should be applied or rejected in a receiver replica. The timestamps needed for handling this can be built without problems [34].

2. *Minimal convergence effort* (mCE): There is a tie between all alternatives in both the *operation ordering* and *synchronisation degree* axes. So, all their alternatives seem to be equally adequate in this goal. The other two axes do not have ties among their best choices, but it is worth noting that *convergence strategy* is the main axis to be considered regarding this mCE goal. The replication protocol to be used has a low impact in the final effort.

   PC is able to guarantee convergence without much effort, since all replicas accept what the primary propagates and no conflicts arise. It obtains a good (++) mark. QB_S also guarantees convergence, but requires a careful voting management that may lead to deadlocks in case of conflicts. Its mark is poor (–). Finally, both MM and QB_P do not take care of conflicts in their default protocols. This guarantees a very good performance and a minimal effort. However,

they do not guarantee convergence by default if any pair of conflicting concurrent update operations exist. Thus, they also receive a poor (–) mark, although it becomes moderate (+) if conflicts are rare.

Regarding convergence strategies, RO and MC are mechanisms needed to recover convergence once it has been lost due to a network partition when every resulting service subgroup may go on, maintaining their availability. Those techniques are not regularly applied when no disconnection has arised. UN is based on defining conflict-free data types. The effort should be applied at object definition time. Once its feasibility is confirmed, there are no implementation nor execution overheads. On the other hand, OW demands operation timestamping. Such timestamping may be based on global logical counters [34] that are trivially built, but introduce a non-negligible overhead on message size. This explains why the UN mark (++) is better than the OW one (+) in this goal. As a result, every *-*-*-UN combination is worth to be considered here.

3. *Minimal convergence recovery time* (mCT): This goal considers the time spent in replica state reconciliation once a network partition is healed. The synchronisation degree axis is irrelevant since it deals with inter-requests ordering and recovery messages do not participate in that order. For the same reason, operation ordering is also irrelevant.

   Regarding replication protocols, QB_S should be discarded, since it was defined for achieving strong consistency and adopts the *primary partition* model [15] in case of disconnection. Apparently, PC shares the same problem, but it has an important difference: the primary copy is an inherent leader process and behaves as a group representative for state transference once the network partition disappears [20], assuming that there is a primary process per partition [5]. Thus, PC becomes the best choice in this regard. On the other hand, MM and QB_P may need some intra-group coordination among their server processes in order to find out which are the set of updates to be transferred. That knowledge is immediately achieved in PC.

   Finally, among the existing convergence strategies, OW is the best since it selects the newest value in case of conflict. This means that once the network connectivity is recovered a single version of each updated object needs to be transferred. This minimises the amount of needed messages and their size. UN needs to transfer all missed operations among process groups. This demands more time and space than in OW. So, its mark (+) is worse than that of OW. Finally, RO demands operation cancellation and re-execution for merging the missed updates in order to define an agreed history sequence. This is still more time demanding than in UN. MC demands human intervention and this implies larger intervals than every other (automated) solution.

Taking into account all these aspects, every PC-*-*-OW combination is worth considering in this goal.

4. *Minimal transfer size* (mTS): This goal is highly related with the previous one, but the replication protocol is unimportant in this case, since that protocol does not condition what data needs to be transferred. As a result, only the convergence strategy should be assessed.

   OW provides an immediate criterion for reducing the size of the data to be transferred in the reconciliation stage. It only propagates the latest known version of each modified object. This deserves a good mark (++).

   UN and RO need to transfer to the reconnected system groups either their missed operations or their effects, since convergence is usually recovered once every operation is known by every system process. This usually means a larger transfer size than in OW, since OW has a clear criterion for selecting a single value per element, while these two other approaches have no transfer minimising criterion. This is the worst possible behaviour in this regard and it explains their poor mark (–).

   MC requires human intervention to solve conflicts. The amount of data to be applied may be minimal (the user decides, and such decision may be better than other automated ones, since users may consider many more aspects than those included in a program). However, human conflict resolution needs to manage at least a summary of the conflicting operations. That information should be transferred to the computer that manages user interaction for conflict resolution. Thus, in the end, this alternative needs two data transfer phases, since later on, such decision should be announced and the results or the code of the accepted operations should be propagated to the participating nodes that had missed their effects. The overall size and cost of these approaches strongly depend on the concrete mechanisms being used in each phase. Thus, its mark is left blank since it is system-dependent, but it could not be better than moderate (+) since it requires two data transfer phases.

   The best combinations in this goal are those that match the *-*-*-OW pattern. Although UN, RO and MC are not recommended, they only need some criterion to reduce the size of missed updates before transferring them in order to improve their mark. Some criteria examples have been proposed elsewhere. For instance, delta-state CRDTs [2] reduce the amount of elements to be modified by each pending operation, improving in this way the behaviour of UN. Also, they transfer only the latest value of the updated elements in case of long lasting partitions. With those extensions, the mark differences between OW and other mechanisms may vanish.

An exhaustive experimental evaluation of all existing alternatives for implementing eventual consistency is unapproachable, since there are too many combinations to be considered. Table 2 has shown a high level assessment of all those alternatives, providing a first guide for choosing the best basic approaches in each concrete goal.

No single combination is the best for all possible goals. In some proposals [2], this has led to the implementation of several mechanisms in a particular axis, using the best of them in each particular scenario.

4.3 Strategies along Time

With the aim of complementing our historical revision started in Section 2, Table 3 shows which papers were the first using some combinations of implementation strategies discussed in this section. In order to have a wider variety, 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 their protocol that admits multi-master management with commutative operations. The 2PC abbreviation represents the two phase commit protocol explained in [44] 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.

Paper	**Repl. prot.**	**Order**	**Sync.**	**Converg.**	**CAP mgmnt.**		**Year**
					C	**A**	
Johnson and Thomas [34]	MM	NO, PO	A, 1S	OW	FIFO	Y	1975
Bunch [12]	PC	TO	2S	N/A	strong	N	1975
Alsberg and Day [3]	MM	NO	1S	UN	relaxed	Y	1976
Lindsay et al. [44]	PC, QB	PO	2S	2PC	strong	N	1979
Parker et al. [52]	MM	PO	1S	MC	causal	Y	1981
Apers and Wiederhold [4]	–	PO	1S	RO	strong	N	1985
Ladin et al. [39]	MM	PO, TO	1S	UN	causal	Y	1990
Shapiro and Preguiça [59]	MM	NO,PO	A	UN	relaxed	Y	2007
Almeida et al. [2]	MM	NO	A	UN	relaxed	Y	2015

Table 3. Implementation strategies used in several proposals

Table 3 shows that one of the first implementations of eventual consistency, that of Johnson and Thomas (1975) [34], already provided one of the best combinations of implementation approaches regarding response time: MM-NO-A-OW.

However, its OW strategy for state convergence cannot be used in every application; it is adequate for small objects like those assumed in [34]. Until 2007, with the proposal of CRDTs [59], we were not able to find a solution of similar quality (MM-NO-A-UN). However, its NO approach for operation ordering and UN strategy for state convergence is based on commutative operations, that require their re-execution at every replica. Besides, CRDTs require partial order (causal propagation) in case of using their state-based variant. Delta-based CRDTs [2] fix that problem, since they use the same solution (MM-NO-A-UN) but relying on partial delta state transfers, avoiding in this way potentially costly re-executions at reception time. This guarantees an excellent performance. Due to this, delta-CRDTs are used in the implementation of some modern NoSQL databases, like Riak and Cassandra.

5 CONCLUSIONS

A discussion on several aspects of eventual consistency has been provided. Eventual consistency is basically a liveness property (data convergence) to be added to relaxed consistency models. Due to this, it had not had a formal specification similar to other consistency conditions since they have been usually specified as safety constraints. However, Bouajjani et al. [10] and Burckhardt [13] have recently proposed specifications that carefully characterise safety (program correctness) and liveness (eventual state convergence) correctness conditions for eventually consistent services.

Although eventual consistency has received a lot of attention nowadays 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 although relevant works in this subject.

The border between inherently convergent and relaxed models has been set. Those relaxed models (e.g., PRAM and causal) may be taken as a basis for implementing eventually consistent services. This shows that eventual consistency is quite a relaxed condition, and allows us to extend the classical CAP theorem constraints, since CAP's consistency not only encompasses the atomic model but also every consistency based on consensus.

Finally, four complementary aspects have been identified for implementing eventual consistency. In each aspect, several alternatives exist, and this means that there are many implementation strategies for developing eventually consistent services. A performance evaluation guide for assessing those strategies has been given, identifying the best combination of mechanisms for achieving some concrete application goals.

REFERENCES

[1] Ahamad, M.—Burns, J. E.—Hutto, P. W.—Neiger, G.: Causal Memory. Proceedings of the 5th International Workshop on Distributed Algorithms and Graphs (WDAG '91), Delphi, Greece, 1991, pp. 9–30.

[2] Almeida, P. S.—Shoker, A.—Baquero, C.: Efficient State-Based CRDTs by Delta-Mutation. In: Bouajjani, A., Fauconnier, H. (Eds.): Networked Systems (NETYS 2015). Springer, Cham, Lecture Notes in Computer Science, Vol. 9466, 2015, pp. 62–76.

[3] Alsberg, P.—Day, J. D.: A Principle for Resilient Sharing of Distributed Resources. Proceedings of the 2nd International Conference on Software Engineering (ICSE '76), San Francisco, CA, USA, 1976, pp. 562–570.

[4] Apers, P. M. G.—Wiederhold, G.: Transaction Classification to Survive a Network Partition. Technical report, STAN-CS-85-1053, Department of Computer Science, Stanford University, Stanford, CA, USA, 1985.

[5] Asplund, M.—Nadjm-Tehrani, S.—Beyer, S.—Galdámez, P.: Measuring Availability in Optimistic Partition-Tolerant Systems with Data Constraints. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07), Edinburgh, UK, 2007, pp. 656–665, doi: 10.1109/DSN.2007.62.

[6] Attiya, H.: Robust Simulation of Shared Memory: 20 Years After. Bulletin of the EATCS, Vol. 100, 2010, pp. 99–113.

[7] Bailis, P.—Venkataraman, S.—Franklin, M. J.—Hellerstein, J. M.—Stoica, I.: Quantifying Eventual Consistency with PBS. The VLDB Journal, Vol. 23, 2014, No. 2, pp. 279–302.

[8] Bernstein, P. A.—Das, S.: Rethinking Eventual Consistency. Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13), New York, NY, USA, 2013, pp. 923–928, doi: 10.1145/2463676.2465339.

[9] Birrell, A. D.—Levin, R.—Needham, R. M.—Schroeder, M. D.: Grapevine: An Exercise in Distributed Computing. Communications of the ACM, Vol. 25, 1982, No. 4, pp. 260–274, doi: 10.1145/358468.358487.

[10] Bouajjani, A.—Enea, C.—Hamza, J.: Verifying Eventual Consistency of Optimistic Replication Systems. Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14), San Diego, CA, USA, 2014, pp. 285–296, doi: 10.1145/2535838.2535877.

[11] Budhiraja, N.—Marzullo, K.—Schneider, F. B.—Toueg, S.: Optimal Primary-Backup Protocols. In: Segall, A., Zaks, S. (Eds.): Distributed Algorithms (WDAG '92). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 647, 1992, pp. 362–378.

[12] Bunch, S. R.: 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, 1975, pp. 71–106.

[13] Burckhardt, S.: Principles of Eventual Consistency. Foundations and Trends in Programming Languages, Vol. 1, 2014, No. 1–2, pp. 1–150, doi: 10.1561/2500000011.

[14] Carey, M. J.—Livny, M.: Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication. Proceedings of the 14th International Conference on Very Large Data Bases (VLDB '88), Los Angeles, CA, USA, 1988, pp. 13–25.

[15] Chockler, G. V.—Keidar, I.—Vitenberg, R.: Group Communication Specifications: A Comprehensive Study. ACM Computing Surveys, Vol. 33, 2001, No. 4, pp. 427–469, doi: 10.1145/503112.503113.

[16] Cosell, B. P.—Johnson, P. R.—Malman, J. H.—Schantz, R. E.—Sussman, J.—Thomas, R. H.—Walden, D. C.: An Operational System for Computer Resource Sharing. Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP '75), Austin, Texas, USA, 1975, pp. 75–81, doi: 10.1145/800213.806524.

[17] Davidson, S. B.: Optimism and Consistency in Partitioned Distributed Database Systems. ACM Transactions on Database Systems, Vol. 9, 1984, No. 3, pp. 456–481, doi: 10.1145/1270.1499.

[18] Défago, X.—Schiper, A.—Urbán, P.: Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. ACM Computing Surveys, Vol. 36, 2004, No. 4, pp. 372–421, doi: 10.1145/1041680.1041682.

[19] de Juan-Marín, R.—Decker, H.—Muñoz-Escoí, F. D.: Revisiting Hot Passive Replication. Proceedings of the 2nd International Conference on Availability, Reliability and Security (ARES '07), Vienna, Austria, 2007, pp. 93–102.

[20] de Juan-Marín, R.—Decker, H.—Armendáriz-Íñigo, J. E.—Bernabéu-Aubán, J. M.—Muñoz-Escoí, F. D.: Scalability Approaches for Causal Multicast: A Survey. Computing, Vol. 98, 2016, No. 9, pp. 923–947.

[21] Demers, A. J.—Greene, D. H.—Hauser, C.—Irish, W.—Larson, J.—Shenker, S.—Sturgis, H. E.—Swinehart, D. C.—Terry, D. B.: Epidemic Algorithms for Replicated Database Maintenance. Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC '87), Vancouver, BC, Canada, 1987, pp. 1–12, doi: 10.1145/41840.41841.

[22] Fekete, A.—Gupta, D.—Luchangco, V.—Lynch, N. A.—Shvartsman, A. A.: Eventually-Serializable Data Services. Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96), Philadelphia, PA, USA, 1996, pp. 300–309, doi: 10.1145/248052.248113.

[23] Fekete, A. D.—Ramamritham, K.: Consistency Models for Replicated Data. In: Charron-Bost, B., Pedone, F., Schiper, A. (Eds.): Replication: Theory and Practice. Chapter 1. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 5959, 2010, pp. 1–17.

[24] Fischer, M. J.—Lynch, N. A.—Patterson, M. S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM, Vol. 32, 1985, No. 2, pp. 374–382, doi: 10.1145/3149.214121.

[25] Fischer, M. J.—Michael, A.: Sacrificing Serializability to Attain High Availability of Data. Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '82), Los Angeles, CA, USA, 1982, pp. 70–75, doi: 10.1145/588111.588124.

[26] Fox, A.—Brewer, E. A.: Harvest, Yield and Scalable Tolerant Systems. Proceedings of The Seventh Workshop on Hot Topics in Operating Systems (HotOS '99), Rio Rico, Arizona, USA, 1999, pp. 174–178, doi: 10.1109/HOTOS.1999.798396.

[27] Gifford, D. K.: Weighted Voting for Replicated Data. Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP '79), Pacific Grove, CA, USA, 1979, pp. 150–162, doi: 10.1145/800215.806583.

[28] Gilbert, S.—Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. ACM SIGACT News, Vol. 33, 2002, No. 2, pp. 51–59, doi: 10.1145/564585.564601.

[29] Golab, W. M.—Rahman, M. R.—Auyoung, A.—Keeton, K.—Gupta, I.: Client-Centric Benchmarking of Eventual Consistency for Cloud Storage Systems. 2014 34th IEEE International Conference on Distributed Computing Systems (ICDCS), Madrid, Spain, 2014, pp. 493–502.

[30] Goodman, J. R.: Cache Consistency and Sequential Consistency. Technical report, No. 61, IEEE Scalable Coherent Interface Working Group, 1989.

[31] Herbst, N. R.—Kounev, S.—Reussner, R. H.: Elasticity in Cloud Computing: What It Is, and What It Is Not. 10th International Conference on Autonomic Computing (ICAC), San Jose, CA, USA, 2013, pp. 23–27.

[32] Herlihy, M. P.—Wing, J. M.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems, Vol. 12, 1990, No. 3, pp. 463–492, doi: 10.1145/78969.78972.

[33] Hutto, P. W.—Ahamad, M.: Slow Memory: Weakening Consistency to Enchance Concurrency in Distributed Shared Memories. Proceedings of the 10th IEEE International Conference on Distributed Computing Systems (ICDCS), Paris, France, 1990, pp. 302–309, doi: 10.1109/ICDCS.1990.89297.

[34] Johnson, P. R.—Thomas, R. H.: The Maintenance of Duplicate Databases. RFC 677, Network Working Group, Internet Engineering Task Force, 1975, doi: 10.17487/rfc0677.

[35] Kawell Jr., L.—Beckhardt, S.—Halvorsen, T.—Ozzie, R.—Greif, I.: Replicated Document Management in a Group Communication System. Proceedings of the 1988 ACM Conference on Computer-Supported Cooperative Work (CSCW), Portland, Oregon, USA, 1988, p. 395, doi: 10.1145/62266.1024798.

[36] Krishnakumar, N.—Bernstein, A. J.: Bounded Ignorance in Replicated Systems. Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '91), Denver, Colorado, USA, 1991, pp. 63–74, doi: 10.1145/113413.113419.

[37] Kumar, A.—Stonebraker, M.: Semantics Based Transaction Management Techniques for Replicated Data. Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD '88), Chicago, Illinois, USA, 1988, pp. 117–125, doi: 10.1145/50202.50215.

[38] Kumar, P.—Satyanarayanan, M.: Log-Based Directory Resolution in the Coda File System. Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems (PDIS '93), San Diego, CA, USA, 1993, pp. 202–213, doi: 10.1109/PDIS.1993.253092.

[39] Ladin, R.—Liskov, B.—Shrira, L.: Lazy Replication: Exploiting the Semantics of Distributed Services. Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing (PODC '90), Quebec City, Quebec, Canada, 1990, pp. 43–57, doi: 10.1145/93385.93399.

[40] Lakshman, A.—Malik, P.: Cassandra: A Decentralized Structured Storage System. ACM SIGOPS Operating Systems Review, Vol. 44, 2010, No. 2, pp. 35–40, doi: 10.1145/1773912.1773922.

[41] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, Vol. 21, 1978, No. 7, pp. 558–565, doi: 10.1145/359545.359563.

[42] Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, Vol. 28, 1979, No. 9, pp. 690–691, doi: 10.1109/TC.1979.1675439.

[43] Lamport, L.: On Interprocess Communication. Distributed Computing, Vol. 1, 1986, No. 2, pp. 77–101.

[44] Lindsay, B. G.—Selinger, P. G.—Galtieri, C. A.—Gray, J. N.—Lorie, R. A.—Price, T. G.—Putzolu, F.—Traiger, I. L.—Wade, B. W.: Notes on Distributed Databases. Technical report, RJ2571 (33471), IBM Research Laboratory, San Jose, CA, USA, 1979.

[45] Lipton, R. J.—Sandberg, J. S.: PRAM: A Scalable Shared Memory. Technical report, CS-TR-180-88, Princeton University, USA, 1988.

[46] Maekawa, M.: A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. ACM Transactions on Computer Systems, Vol. 3, 1985, No. 2, pp. 145–159.

[47] Mockapetris, P. V.: Domain Names – Concepts and Facilities. RFC 882, Network Working Group, Internet Engineering Task Force, 1983, doi: 10.17487/rfc0882.

[48] Mockapetris, P. V.: Domain Names – Implementation and Specification. RFC 883, Network Working Group, Internet Engineering Task Force, 1983, doi: 10.17487/rfc0883.

[49] Mosberger, D.: Memory Consistency Models. ACM SIGOPS Operating Systems Review, Vol. 27, 1993, No. 1, pp. 18–26, doi: 10.1145/160551.160553.

[50] Muñoz-Escoí, F. D.—Bernabéu-Aubán, J. M.: A Survey on Elasticity Management in PaaS Systems. Computing, Vol. 99, 2017, No. 7, pp. 617–656.

[51] O'Neil, P. E.: The Escrow Transactional Method. ACM Transactions on Database Systems, Vol. 11, 1986, No. 4, pp. 405–430.

[52] Parker, D. S.—Popek, G. J.—Rudisin, G.—Stoughton, A.—Walker, B. J.—Walton, E.—Chow, J. M.—Edwards, D. A.—Kiser, S.—Kline, C. S.: Detection of Mutual Inconsistency in Distributed Systems. In: Berkeley Workshop, 1981, pp. 172–184.

[53] Pascual-Miret, L.—González de Mendívil, J. R.—Bernabéu-Aubán, J. M.—Muñoz-Escoí, F. D.: Widening CAP Consistency. Technical report, IUMTI-SIDI-2015/003, Universitat Politècnica de València, Valencia, Spain, 2015.

[54] Pascual-Miret, L.—Muñoz-Escoí, F. D.: Replica Divergence in Data-Centric Consistency Models. 2016 27th International Workshop on Database and Expert Systems Applications (DEXA Workshops), Porto, Portugal, 2016, pp. 109–112.

[55] Pu, C.—Leff, A.: Replica Control in Distributed Systems: An Asynchronous Approach. Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data (SIGMOD '91), Denver, Colorado, USA, 1991, pp. 377–386, doi: 10.1145/115790.115856.

[56] Saito, Y.—Shapiro, M.: Optimistic Replication. ACM Computing Surveys, Vol. 37, 2005, No. 1, pp. 42–81.

[57] Sarin, S. K.—Blaustein, B. T.—Kaufman, C. W.: System Architecture for Partition-Tolerant Distributed Databases. IEEE Transactions on Computers, Vol. C-34, 1985, No. 12, pp. 1158–1163.

[58] Satyanarayanan, M.—Kistler, J. J.—Kumar, P.—Okasaki, M. E.—Siegel, E. H.—Steere, D. C.: Coda: A Highly Available File System for a Distributed Workstation Environment. IEEE Transactions on Computers, Vol. 39, 1990, No. 4, pp. 447–459, doi: 10.1109/12.54838.

[59] Shapiro, M.—Preguiça, N. M.: Designing a Commutative Replicated Data Type. Technical report RR-6320, INRIA, Rocquencourt, France, 2007.

[60] Shapiro, M.—Preguiça, N. M.—Baquero, C.—Zawirski, M.: Convergent and Commutative Replicated Data Types. Bulletion of the EATCS, Vol. 104, 2011, pp. 67–88.

[61] Tanenbaum, A. S.—van Steen, M.: Distributed Systems – Principles and Paradigms. 2nd edition, Pearson Education, 2007.

[62] Terry, D. B.—Demers, A. J.—Petersen, K.—Spreitzer, M. J.—Theimer, M. M.—Welch, B. B.: Session Guarantees for Weakly Consistent Replicated Data. Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS '94), Austin, Texas, USA, 1994, Art. No. 19.

[63] Viotti, P.—Vukolić, M.: Consistency in Non-Transactional Distributed Storage Systems. ACM Computing Surveys, Vol. 49, 2016, No. 1, Art. No. 19.

[64] Vogels, W.: Eventually Consistent. ACM Queue, Vol. 6, 2008, No. 6, pp. 14–19, doi: 10.1145/1466443.1466448.

[65] Walker, B. J.—Popek, G. J.—English, R.—Kline, C. S.—Thiel, G.: The LOCUS Distributed Operating System. Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP '83), Bretton Woods, New Hampshire, USA, 1983, pp. 49–70, doi: 10.1145/800217.806615.

[66] Yu, H.—Vahdat, A.: Design and Evaluation of a Continuous Consistency Model for Replicated Services. Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00), San Diego, CA, USA, 2000, pp. 305–318.

Francesc D. Muñoz-Escoí received his Ph.D. in computer science from Universitat Politècnica de València (UPV) in 2001. He currently works as Associate Professor at UPV. He has published more than 100 papers in international conferences and journals. His research interests cover multiple distributed system areas: group communication services, distributed algorithms, replication protocols, recovery approaches, distributed data management, elastic services and cloud computing.

José-Ramón García-Escrivá received his final degree in computer science from UPV in 1987, where he currently works as Associate Professor. He has been involved as researcher in more than 20 projects. His interests cover both web technologies and distributed system areas.

Juan Salvador Sendra-Roig currently works as Associate Professor at UPV. His research interests cover string algorithms and several distributed system areas: distributed algorithms, replication protocols, distributed data management, and indexing and pre-processing of massive text data.

José M. Bernabéu-Aubán received his Ph.D. in computer science from Georgia Institute of Technology (USA) and currently works as Full Professor at UPV where he leads the Distributed Systems Research Group. He has led the Instituto Universitario Mixto Tecnológico de Informática at UPV from 1994 to 2004 and since 2014 up to now. From 2004 to 2011, he joined Microsoft Corp. working actively in the architecture, design and development of the Windows Azure platform, co-authoring some of its patents. He has written multiple papers in journals and conferences in different distributed system areas. He has led more than 30 research projects in those fields.

José Ramón González de Mendívil received his Ph.D. in sciences from the University of the Basque Country (Spain) in 1993 and currently works as Full Professor at Universidad Pública de Navarra where he leads the Distributed Systems research group. He has written several papers in journals and conferences in different distributed system areas: distributed algorithms, deadlock detection, deadlock resolution, replicated databases, and replication protocols. His current interest is designing elastic services for PaaS using fuzzy performance models.