A Policy-Based Characterization Model and a Comprehensive Survey of Database Replication Systems

M. I. Ruiz-Fuertes[1], A. Schiper[2], R. C. Oliveira[3], F. D. Muñoz-Escoí[1]

[1]Instituto Tecnológico de Informática
Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain
[2]École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland
[3]HASLab / INESC TEC, Universidade do Minho, Braga, Portugal

miruifue@iti.upv.es, andre.schiper@epfl.ch, rco@di.uminho.pt, fmunyoz@iti.upv.es

Technical Report ITI-SIDI-2012/002

A Policy-Based Characterization Model and a Comprehensive Survey of Database Replication Systems

M. I. Ruiz-Fuertes[1], A. Schiper[2], R. C. Oliveira[3], F. D. Muñoz-Escoí[1]

[1]Instituto Tecnológico de Informática
Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain
[2]École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland
[3]HASLab / INESC TEC, Universidade do Minho, Braga, Portugal

Technical Report ITI-SIDI-2012/002

e-mail: miruifue@iti.upv.es, andre.schiper@epfl.ch, rco@di.uminho.pt, fmunyoz@iti.upv.es

January, 2012

Abstract

Since the appearance of the first distributed databases until the current modern replication systems, the research community has proposed multiple protocols to manage data distribution and replication, along with concurrency control algorithms to handle transactions running at every system node. Many protocols are thus available, each one with different features and performance, and guaranteeing different consistency levels. To know which replication protocol is the most appropriate, two aspects must be considered: the required level of consistency and isolation (i.e., the correctness criterion), and the properties of the system (i.e., the scenario), which will determine the achievable performance. In order for the administrator to select a proper replication protocol, the available protocols must be fully and deeply known. A good description of each candidate is fundamental, but a common ground is mandatory to compare the different options and to estimate their performance in the given scenario. This paper proposes a precise characterization model that allows us to decompose algorithms into individual interactions between significant system elements, as well as to define some underlying properties, and to associate each interaction with a specific policy that governs it. We later use this model as basis for a historical study of the evolution of database replication techniques, thus providing an exhaustive survey of the principal existing systems.

1 Introduction

Since traditional stand-alone database systems started to become distributed and replicated in the mid seventies, many different algorithms for concurrency and replica control have appeared thanks to the contributions of many authors. These proposals came from different communities, each one based on different assumptions and focused on the achievement of different goals. Each new distributed or replicated system defined its own methods, followed its own naming conventions and presented its algorithms in different ways: from descriptions in plain textual form to more or less detailed specifications in its own pseudocode language. In the midst of this abundance and disparity, it was difficult to find an appropriate solution for a given problem or to compare two apparently similar options to choose the best one for a given scenario.

Some authors from the distributed systems community performed different surveys and classifications [26, 65, 67]. Gray et al. [26] made the first step to study the existing systems for database replication, distinguishing between eager and lazy propagation strategies, and group and master ownership strategies, which combine between them to produce four types of replication systems. Wiesmann et al. [67] proposed a classification based on three parameters, where replication techniques are characterized with regard to their

server architecture (either primary-backup or update-everywhere), their server interaction (either constant or linear) and their transaction termination (either voting or non-voting). Both contributions aimed to classify a broad set of non-related systems, according to some criteria, generally coarse-grained in order to reduce the complexity and the number of equivalence classes. Each one of these criteria thus agglutinated several individual pieces of behavior that were observed together in the studied systems. Another approach was focusing on one set of similar systems, e.g., those mainly based on a given general technique, and characterize and classify them into disjoint subsets, according to other used techniques. This is the case of a work by Wiesmann and Schiper [65], which is focused on replication systems based on total order broadcast –namely, the three most relevant: active, certification-based and weak voting replication– and provides a performance comparison between them and two other widely used techniques –primary copy and lazy replication– that do not rely on group communication.

However, more and more systems appeared and those coarse-grained criteria turned to be insufficient when proposals became hybrid or explored new methodologies not yet categorized. A finer grain is thus necessary to better characterize replication systems and to provide a common ground to compare them all. More than a set of disjoint equivalence classes, what is needed is a common and general framework where different replication systems could be examined and compared. This approach was followed by Bernstein and Goodman in 1981 [8], when they surveyed almost all concurrency control algorithms for distributed databases published until then. In order to do so, they first proposed a framework consisting of a common terminology and a problem decomposition. By unifying concepts and splitting a complex process into several subproblems, a rich characterization and comparison among systems is possible.

Important advances have emerged in the last 30 years since the work of Bernstein and Goodman [8] was published, such as the development of group communication systems with more complex communication primitives, leading to the appearance of new techniques. A new framework for comparing replication techniques was then proposed by Wiesmann et al. [66]. In this framework, five generic phases are identified within a replication protocol: request, server coordination, execution, agreement coordination, and response. According to this, authors then describe different techniques, analyzing how they perform each phase.

Following a similar approach and trying to help researchers and practitioners to make their way through the assorted plethora of database replication systems, this paper proposes a new characterization model that provides even more detailed descriptions than the framework by Wiesmann et al. [66], by splitting a replication system into a group of policies. This model allows us to describe in detail the nature of the interaction between significant system elements: the underlying local database, the clients and their transactions, and the group of system servers or components. Every time these elements interact, a specific policy regulates the way on which this interaction is performed. Thanks to the fine grain achieved by this model, almost all existing systems can be fully characterized. The resulting detailed descriptions allow an easier comparison between different database replication systems.

The second contribution of this paper is an extensive historical survey of database replication systems, based on the common description framework previously proposed. This chronological survey describes each proposal by detailing, for each policy, the followed strategy, as well as the enforced correctness criterion. As a result, it is highly valuable in order to compare and understand different proposals. This survey provides not only an empirical proof of the usefulness of our model proposal but also a study of the evolution of database replication systems and a reference manual for readers interested in this field, regardless of their background. It will allow beginners to obtain a global and precise idea of the state of the art of the database replication research field and will also provide a historical vision of the evolution of these systems, allowing us to detect which strategies are the most used and which combinations guarantee each correctness criterion. Moreover, this study makes it easier to identify combinations of strategies that are seldom used but might make sense if new goals are set for replication protocols, such as the support of either more relaxed or stricter consistency models, or the increase of the system scalability. Furthermore, this survey may enable us to identify which advances at which fields (in database management systems, in group communication systems, in isolation levels and correctness criteria specifications, in replication protocols, etc.) allowed the appearance of each described proposal, as well as to foresee which other advances could have a relevant effect on this evolution.

The rest of the paper is structured as follows. Section 2 presents all background concepts and definitions. Section 3 proposes the characterization model for database replication systems. The survey is

presented in Section 4. Finally, Section 5 concludes the paper.

2 Background: Concepts and Definitions

Server, clients and failures. A distributed (and possibly replicated) database is a distributed system composed of database *servers*, also called *nodes* or *sites*, $N_1, N_2, \ldots, N_n$, which store data, and *clients* or *users*[1] that contact these servers to access the data. Servers only communicate by message passing, as they do not have shared memory. Consequently, they neither have a global clock. Depending on the *failure model* assumed, failures are considered at different degrees. In the *crash-stop* model, when servers fail they permanently stop all their processing. In this case, a site is *correct* if it never crashes, otherwise it is *faulty*. The *crash-recovery* model allows failed servers to eventually recover after a crash. A more complex failure model, *Byzantine failures*, assumes that sites and their environment can behave in an arbitrary way.

Distribution and replication. From the point of view of the client, a *database* is a collection of logical data items. Each logical data item is physically stored at the servers. If the set of database items is partitioned and distributed among the sites, the system constitutes a *purely distributed database*. If some replication is introduced, so that different physical copies of the same logical data item are stored at different sites, the system is a *replicated database*. Replication is managed by a *replication protocol*. The number of physical copies of data item x is the *degree of replication* of x. Depending on the degree of replication of the data items and on the number of system nodes, a complete copy of the database may be stored at each site. This is called *full replication*. When not all sites store the complete set of data items, but only a subset, the replication is *partial*. Each of the copies of a given data item is called a *replica*. In full replication, the term replica is also used to refer to any server, as they contain a copy of all data items. When the system is fully replicated, we also denote sites as $R_1, R_2, \ldots, R_n$, to highlight that they are replicas.

Advantages of replication. Replication is a common solution to achieve *availability*: by storing multiple copies, clients can operate even if some sites have failed. Moreover, and despite the drawback of having to update all copies of each data item, replication can also improve performance, as clients are more likely to access a copy close by. For these reasons, replication is mostly preferred over pure distribution. Even initial systems, which were fundamentally distributed, introduced replication at some degree.

Desirable properties of replicated systems. Traiger et al. [64] suggested the concepts of one-copy equivalence and location, replica, concurrency, and failure transparencies, as desirable features for distributed systems. A system that provides transparency is as easy to use as a stand-alone, one-copy system. This way, although the data is geographically distributed and may move from place to place, users can act as if all the data were in one node (location transparency). Moreover, although the same data item may be replicated at several nodes, users can treat the item as if it were stored as a single item at a single node (replication transparency). In addition, it appears as if there were no concurrency in the system (concurrency transparency). Finally, the effects of confirmed operations survive hardware and software failures, while all the effects of an in-progress operation are undone in case of failure (failure transparency).

Local databases, transactions and sessions. A local *Database Management System*, or DBMS for short, runs at each site and is responsible for the control of the data. The DBMS allows clients to access the data for reading and writing, through the use of *transactions*. Transactions are sequences of read and write operations (e.g., sequences of SQL sentences) followed by a commit or an abort operation, and maintain the ACID properties [28]: atomicity, consistency, isolation and durability. Atomicity requires the all-or-nothing policy: either all the operations of a transaction are performed or none of the operations is reflected in the final state of the database. Consistency requires that each successful transaction produces only legal results. Isolation requires that the effects of running transactions are hidden from other transactions running concurrently. Durability requires that all the data changes made by a successful transaction are persistently

[1] We employ the term *user* to specifically refer to a human agent.

applied, which is done by means of the *commit* operation. On the other hand, if a transaction *aborts*, all its changes are undone. A transaction is called a *query* or a *read-only transaction* if it does not contain any write operation; otherwise it is called an *update transaction*. The set of logical items a transaction reads is called the *readset*. Similarly, the *writeset* is the set of logical items written by a transaction, and usually it also includes the updated or inserted values. The *resultset* is compound by the results that will be returned to the client.

Database users are provided with the concept of *session*, in order to logically group a set of transactions from the same user. Transactions from different users belong to different sessions. However, it can be left to the user the decision of using one or multiple sessions to group their transactions.

Transactions may be executed concurrently in a DBMS. In this case, the concurrency control of the DBMS establishes which executions of concurrent transactions are correct or legal.

Workload, delegate and remote nodes. The database workload is composed of transactions, $T_1, T_2, \ldots$ Transactions from the same client session are submitted sequentially, but may be addressed to different servers, either by the client itself, by a load balancer, or by another component or server that redirects the request to another server. If the request is finally addressed to only one server, this server is called the *delegate* for that transaction and it is responsible for its execution. The rest of the system nodes are called *remote* nodes for that transaction. When the database is purely distributed (partitions of the data stored at different nodes) or partial replication is used, if the contacted server does not store all the items the transaction needs to access, other servers can be requested to execute different portions of the transaction, which is then called a *distributed transaction*. In this case, the concepts of delegate and remote nodes are no longer applicable: the client contacts to a server where a root transaction is started, and accesses to data stored in other server involve the creation of a subtransaction in that other node, which is also called *cohort*. This way, each operation may involve a communication with another node in the system. If accessed items are replicated, subtransactions must be executed in all copies. To coordinate the local subtransactions executed at each participating site, all accesses are managed by a distributed concurrency control. In order to commit these distributed transactions, atomic commit protocols, explained below, are used, acting the root transaction as coordinator.

Conflicts. Two transactions *conflict* if they have conflicting operations. Two operations conflict if they are issued by different transactions, access the same data item and at least one of the operations is a write. Conflicts among transactions should be treated somehow, ensuring that the conflicting operations are executed at the same order at every replica. There are two main approaches for treating conflicts. A system is *pessimistic* or *conservative* if it avoids conflicts by establishing some locks, mutexes or other barriers over items accessed by a transaction, so that they cannot be concurrently accessed by other transactions. On the other hand, a system is *optimistic* if it lets transactions freely access items, resolving possible conflicts only when they appear or at the end of the transaction, during termination.

Server layers. Inside a server, different layers can be identified: the network or communication layer, at the bottom of the stack; the data layer; the replication layer; and the application layer, on the top. These general layers may appear merged together at different systems. For instance, in a system that manages replication by embedding the necessary code into the DBMS internals, the data and the replication layer are merged. On the other hand, we say a replication system is *based on middleware* [6] when it gathers all replication mechanisms in a replication layer, i.e., a software layer placed between the instance of the database management system and the applications accessing the data. This provides an independence among system components which leads to a high portability (e.g., to migrate the middleware into an environment based on a different DBMS).

Interactive execution vs. service request. A transaction can be submitted for execution either operation by operation, or in a single message. In the former case, called *interactive transaction*, the client submits an operation and waits for its results before sending the next operation. The latter case, called *service request*, is a call to a procedure stored in the database. When the transaction is completed, the transaction outcome

is sent to the client. In the case of interactive transactions, this outcome is a commit or abort confirmation. For service requests, the outcome also includes the results of the request.

Active and passive replication. In a system with *active* replication, the same client request is processed by every node. As opposed to this, each client request in *passive* replication is processed by only one node, which later transfers the updates to the rest of servers. Active replication is usually associated with service request, while passive replication can be used for both interactive execution and service request. Depending on the node that can process a client request, the next two server architectures for passive replication can be distinguished.

Server architecture: primary-backup. The *server architecture* defines where transactions are executed in the first place. Common server architectures are primary-backup and update-everywhere. In a *primary-backup* system, a specific node –called *primary copy* or *master copy*– is associated to each data item. Any update to that item must be first sent to the primary copy, i.e., the primary copy is the delegate server for any update transaction over that item. The rest of the servers, which receive the updates from the primary, are called *backups* and serve only queries over that item. One possible setting is to select a single server as the primary copy for all database items, although this can cause a bottleneck.

Server architecture: update-everywhere. In an *update-everywhere* system, every node is able to serve client requests for updating any data item, so that it is possible that two concurrent updates arrive at different copies of the same data item. In order to avoid inconsistencies, usually some mechanisms are used to propagate the updates and to decide which updates will be successful, aborting transactions when necessary. These mechanisms are the *server interaction* and the *transaction termination* protocols.

Server interaction and writeset application. In the presence of replication, independently of the server architecture, updates from a committed transaction must be propagated to other copies of the affected data items. This is achieved through a mechanism of *server interaction* that sends all write operations to the appropriate sites. This propagation can be made on a per operation basis, distributing writes immediately to other nodes in a *linear interaction* [67] approach; or deferring communication until transaction end, in a *constant interaction* approach. The former case requires more messages (usually one per write operation) than the latter (one message per transaction). The *deferred update* approach [64] is a constant interaction approach, where, once a transaction finishes its operations, its writeset is sent to the appropriate remote nodes, which will then *apply the writeset*, i.e., perform its updates, in their local database copy. Depending on when this propagation and application of updates is made, systems can be *eager* or *lazy* [26]. Eager replication ensures that every node applies the updates inside the transaction boundaries, i.e., before the results are sent back to the client, so that all replicas are updated before such a response is sent. On the other hand, lazy replication algorithms asynchronously propagate updates to other nodes after the transaction commits in its delegate node. A hybrid approach ensures that all replicas have received the updates and the delegate has committed them when the results are sent to the client. Remote nodes will later apply such updates.

Update propagation: ROWA and ROWAA. With either type of server interaction, writes are propagated and applied in remote copies of the affected data items. A basic approach is *Read One Write All* [10], where write operations are required to update all copies so that read operations only need to access one of the replicas. In case of a site failure, it may be impossible to write all replicas and thus the processing must stop. As this is not desirable, the common approach is *Read One Write All Available* (ROWAA) [10]. According to ROWAA, each write operation over data item x is applied at every *available* copy of x, i.e., replicas stored at sites that have not failed. Failed sites are ignored until they *recover* from their failure. Whenever a site recovers from a failure, all its copies must be brought up-to-date before the node can serve read operations.

Update propagation: quorums. Another approach for write propagation is the use of *quorums*. A quorum is the minimum number of replicas that is required for completing an operation. Each transaction operation must then be successfully executed in a quorum of replicas for being considered successful: read operations are required to access a read quorum of replicas before returning the read value to the client, while write operations must update a write quorum of replicas. In a system of size N, sizes for the read quorum, R, and the write quorum, W, are defined in such a way that they guarantee any required property. For example, with $R = 1$ and $W = N$, the previously presented ROWA approach is obtained. A more common quorum configuration ensures that both $W + W$ and $R + W$ are greater than N, so that each write quorum has at least one replica in common with every read quorum and every write quorum, thus ensuring access to the last updated value and also detecting conflicting transactions. Including a majority of the nodes into each quorum allows the system to avoid system partitioning and consequent divergence.

Transaction termination: voting, weak voting and non-voting termination. Whenever a transaction ends, the transaction termination protocol is run to decide the outcome of the transaction (*validation*) and, in case of deciding to commit, take the necessary actions to guarantee transaction durability. Two main approaches can be distinguished: voting and non-voting termination. In a *voting termination*, an extra round of messages is required to coordinate the different sites, as in 2PC. *Weak voting* is a special case of voting termination, where only one node decides the outcome of the transaction and sends its decision to the rest of nodes. In a *non-voting* termination, all sites are able to autonomously and deterministically decide whether to commit or to abort the transaction. In this case, this symmetrical validation process is also called *certification*.[2]

Validation and, thus, certification are usually based on conflicts. The ending transaction T is checked for conflicts with concurrent and already validated (respectively, certified) transactions. If conflicts are found, validation (certification) fails and T is said to be negatively validated (negatively certified) and it is aborted. Otherwise, validation (certification) succeeds and T is said to be validated (certified), successfully validated (successfully certified) or positively validated (positively certified), and it has to be committed in all affected nodes. When validation or certification are not based on conflicts, the validation (certification) succeeds or is positive if the decision taken over T is to commit it. Otherwise, the validation (certification) fails or is negative.

System model. Aspects such as server architecture, server interaction and transaction termination are part of the *system model*, which defines the way in which a system operates.

Replica consistency and inversions. Applying the writesets and ensuring the same order for conflicting operations are necessary actions for maintaining the required level of *replica consistency*. Replica consistency measures the synchronization among the copies of the same data item, i.e., the state of replicas with regard to each other. While the *server-centric view of consistency* defines how consistency is internally enforced at the replicas of the system, the *user-centric view of consistency* is the perception that users, individually or collectively, have about the consistency of the database replication system in use. This perceived consistency is the *user-centric consistency* of the system. Different levels of user-centric replica consistency may be enforced, depending on the needs of the clients. According to Ruiz-Fuertes and Muñoz-Escoí [55], the fact that allows to distinguish between different levels of user-centric consistency is the presence or absence of inversions. An *inversion* occurs when a transaction T_2, which was started by a user after the commitment of a previous transaction T_1, appears to take place *before* T_1 and thus T_2 commits without having been able to see the updates of T_1. Three different levels of user-centric consistency can then be defined. From stricter to more relaxed, they are: absence of inversions, absence of inversions within user sessions, and presence of inversions.

Regarding server-centric consistency, we say that a systems provides *sequentiality* if every replica goes through the same sequence of database states. Some optimizations in writeset application (e.g., letting non conflicting transactions to commit in different orders) may break this sequentiality while still ensuring

[2] We will use the term validation to generically refer to the process of deciding the outcome of a transaction. We will use the term certification to specifically refer to the validation process performed by each node in an independent, deterministic and symmetrical manner. Other authors, however, consider both terms as synonyms.

replica consistency. However, the lack of sequentiality may confuse users and thus it is concealed from their user-centric view of consistency.

Serializable isolation. Transactions are executed under some isolation level, which defines the visibility among operations of different concurrent transactions. The highest isolation level is the *serializable* level, which guarantees a completely isolated execution of transactions, as if they were serially performed, one after the other. Changes made by transaction T are only visible to other transactions after the commitment of T. On the other hand, if T aborts, then its changes are never seen by any other transaction.

Read committed isolation. A more relaxed level is the *read committed* isolation. Under this isolation, data read by a transaction T was written by an already committed transaction, but it is not prevented from being modified again by other concurrent transactions. Thus, these data may have already changed when T commits. For a complete discussion about isolation levels, please refer to Berenson et al. [5].

Correctness criterion. The combination of the replica consistency level and the transaction isolation level guaranteed by a system is called the *correctness criterion* of the database replication system. Bernstein et al. defined the criterion of one-copy serializability (1SR) [10]. According to it, the interleaved execution of users' transactions must be equivalent to a serial execution of those transactions on a standalone database. As highlighted by Ruiz-Fuertes and Muñoz-Escoí [55], 1SR does not impose any level of replica consistency and thus it can be refined with two different subcriteria.

Concurrency control: locks. Local isolation at a DBMS is enforced by the use of a local *concurrency control* mechanism. Concurrency control manages the operations that run in a database at the same time. There are two main options for concurrency control: locks and multiversion systems. In a lock-based system, each data item has a *lock* that regulates the accesses to the item. Operations over that item must previously obtain the corresponding lock. There are *shared* and *exclusive* locks. Shared locks are commonly used for read operations. Several transactions can obtain shared read locks and read the same item at the same time. Write operations require an exclusive lock. No other operation, shared or exclusive, is allowed over an item protected with an exclusive lock.

Concurrency control: multiversion. In multiversion systems, on the other hand, simultaneous accesses to the same data item are resolved by using multiple versions of the item. Versions can be created and deleted but the value they represent is immutable: updates to a data item create a new version of the item. Although there are several versions for each item, only one is the latest: the value written by the last committed transaction that updated the item. More recent versions correspond to transactions which are still on execution and they are not visible to other transactions. Versions generated by aborted transactions are never visible to other transactions. When a transaction starts, a timestamp or a transaction ID is assigned to it. Versions written by a transaction T are marked with the ID of T. With this timestamp information, the multiversion system can determine, for each transaction, which state or snapshot (i.e., which version of each data item) of the database it must read. Thus, a transaction that started at a particular instant t_0 has access, for each data item, to the version of that item which was the latest at time t_0, i.e., which was written by the committed transaction with the highest ID which is smaller than the ID of the reading transaction. As versions are immutable, there is no need to manage locks for read operations. This way, multiversion concurrency control lead to the appearance of a new transaction isolation level called *snapshot isolation*.[3] Some mechanism is usually needed to delete obsolete versions.

Snapshot isolation. In snapshot isolation [5], or SI, transactions get a start timestamp and a snapshot of the database when they start. Transactions are never blocked attempting a read. Write operations are also reflected in the snapshot of the transaction, so that it can access the updated versions afterwards. On the other hand, updates by other transactions active after the transaction start are invisible to the transaction.

[3] Read committed isolation is also possible with a multiversion concurrency control.

When the transaction is ready to commit, a commit timestamp is assigned to it. A transaction T_1 successfully commits only if no other transaction T_2 with a commit timestamp in the interval between the start and the commit timestamps of T_1 wrote data that T_1 also wrote. Otherwise, T_1 will abort. This feature is called the *first-committer-wins* rule.

Two phase locking. Serializability can be achieved by a basic *two-phase locking* (2PL) [10] protocol, where each transaction may be divided into two phases: a *growing* phase during which it obtains locks, and a *shrinking* phase during which it releases locks. Once a lock is released, no new locks can be obtained. 2PL forces all pairs of conflicting operations of two transactions to be executed in the same order and so it achieves serializability. However, using 2PL a *deadlock* may appear. This situation arises when two transactions wait for each other to release a lock. Deadlocks must be detected and one of the transactions aborted in order to remove the deadlock.

Strong strict 2PL. In order to avoid deadlocks and to provide other desirable properties,[4] a variant of 2PL is commonly used: the *strong strict 2PL* (presented by Bernstein et al. [10] as *strict 2PL* but refined later). In strong strict 2PL, all the locks obtained by a transaction are only released after transaction termination.

Atomic commit protocol: 2PC. To ensure consistent termination of distributed transactions, database systems have traditionally resorted to an atomic commit protocol, where each transaction participant starts by voting yes or no and each site reaches the same decision about the outcome of the current transaction: commit or abort. A widely used atomic commit protocol is the two-phase commit protocol (2PC) [24, 40], which involves two rounds of messages for reaching a consensus on the termination of each transaction. 2PC can be centralized or decentralized. In the centralized approach, the coordinator first sends a message to the rest of nodes, with information about the ending transaction. Each server must then reply to the coordinator whether it agrees or not to commit the transaction. If all replies are positive, the coordinator sends a commit message and waits for acknowledgments from all the nodes. If any of the replies was negative, an abort message is sent in the second phase. The decentralized version is similar but with any server starting the process and with responses to every other server. 2PC is a blocking protocol when failures occur.

Atomic commit protocol: 3PC. To support failures, a non-blocking atomic commit protocol (NB-AC) [10, 62] must be used. In these protocols, each participant reaches a decision despite the failure of other participants. A NB-AC protocol fulfills the following properties. (a) Agreement: no two participants decide different outcomes. (b) Termination: every correct participant eventually decides. (c) Validity: if a participant decides commit, then all participants have voted yes. (d) Non triviality: if all participants vote yes, and no participant fails, then every correct participant eventually decides commit. The three-phase commit protocol (3PC) [62] adds an intermediate phase to 2PC to become a non-blocking process. This new (second) phase involves sending a *precommit* message when all nodes have agreed to commit the transaction. After all servers sent their acknowledgments to this *precommit* message, the final commit message is sent. Note that when a transaction needs to abort, such a fact is identified at the end of the first phase. On the other hand, agreement on the commitment is reached at the second phase and the commit is completed in the third phase. Thus, failures in the first phase lead to transaction abortion whilst failures in the second or third ones do not block the protocol nor prevent transaction commitment. The drawback of 3PC is its higher cost due to the extra round of messages. To solve this, Jiménez-Peris et al. proposed another NB-AC protocol that exhibits the same latency as 2PC [31].

Atomic commit protocol: Paxos Commit. The Paxos Commit algorithm [25] runs a Paxos consensus algorithm on the commit/abort decision of each participant to obtain an atomic commit protocol. The result is a complete, decentralized and non-blocking algorithm which is proven to satisfy a clearly stated correctness condition (that of the Paxos algorithm [19, 38, 39, 41]).

[4] Recoverability, cascade abort avoidance and strictness [10], and also commit ordering [52].

Group communication systems and atomic broadcast. The communication among system components is based on message passing. A *Group Communication System* [14], or GCS for short, is commonly used to accomplish communication tasks among servers, by choosing the communication primitive (point to point messages, multicasts, broadcasts) with the appropriate guarantees (e.g., uniform guarantees will be commonly necessary when failures must be tolerated). *Atomic broadcast* (abcast for short) is a group communication abstraction defined by the primitives *broadcast*(m) and *deliver*(m). Abcast satisfies the following properties [27]. (a) Validity: if a correct site broadcasts a message m, then it eventually delivers m. (b) Agreement: if a correct site delivers a message m, then every correct site eventually delivers m. (c) Integrity: for every message m, every site delivers m at most once, and only if m was previously broadcast. (d) Total Order: if two correct sites deliver two messages m and m', then they do so in the same order. Due to this last property, atomic broadcast is also known as total order broadcast.

Optimistic abcast. Two optimistic variants of abcast are the *optimistic atomic broadcast* [49] and the more aggressive *atomic broadcast with optimistic delivery* [36], which allow processes to deliver messages faster, in certain cases. They exploit the spontaneous total order message reception: with high probability, messages broadcast in a local area network are received totally ordered. An atomic broadcast with optimistic delivery is defined by three primitives. First, *TO-broadcast*(m) broadcasts the message m to all nodes in the system. Then, *opt-deliver*(m) delivers a message m optimistically to the application once it is received from the network, in a *tentative order*. Finally, *TO-deliver*(m) delivers m to the application in the *definitive order*, which is a total order. The following properties are satisfied. (a) Termination: if a site TO-broadcasts m, then every site eventually opt-delivers m and TO-delivers m. (b) Global agreement: if a site opt-delivers m (TO-delivers m) then every site eventually opt-delivers m (TO-delivers m). (c) Local agreement: if a site opt-delivers m then it eventually TO-delivers m. (d) Global order: if two sites N_i and N_j TO-deliver two messages m and m', then N_i TO-delivers m before it TO-delivers m' if and only if N_j TO-delivers m before it TO-delivers m'. (e) Local order: a site first opt-delivers m and then TO-delivers m. With such an optimistic delivery, the coordination phase of the atomic broadcast algorithm is overlapped with the processing of messages. This optimistic processing of messages must be only undone when the definitive total order mismatches the tentative one.

3 A Characterization Model for Database Replication Systems

In this section we present a policy-based characterization model that allows us to decompose database replication algorithms into individual interactions between significant system elements, as well as to define some underlying properties, and to associate each interaction with a specific policy that governs it. With this characterization model, a replication system can be described as a combination of policies. This common framework allows an easy understanding and comparison between protocols.

The rest of the section is structured as follows. Subsection 3.1 presents the proposed characterization model, and, next, Subsection 3.2 enumerates the different correctness criteria considered by the surveyed replication systems.

3.1 Interactions, Properties, Strategies and Policies: A Characterization Model

A database replication system can be defined by means of describing the *interactions* among its main components –namely clients, local databases, servers or other system components, and transactions being executed– as well as some basic *behavioral properties*. Each one of these interactions may be performed in different ways. Similarly, each property may take different values. All these options are called *strategies*. A replication system must choose, for each interaction and property, one of the available strategies. The selected strategy is the *policy* that such a system follows for such an interaction or behavior. Each system will provide the necessary mechanisms for implementing the selected strategy. The set of policies a system follows can be divided into four policy families, which gather related policies together: the client policy family, the database policy family, the group policy family, and the transaction policy family.

The client policies regulate the interaction between the client and the rest of the database replication system, i.e., the communication from/to the user. The *request* policy specifies which servers in the system

must receive the client request, and the *response* policy establishes the number of replies that will arrive to the client with the transaction results.

The interaction between the system and the local underlying database management system is defined by internal properties of the DBMS and regulated by the database policies. These policies determine two aspects: the *isolation* level used whenever the transaction operates in the database, and the level of *replication* of the database, i.e., whether it is fully or partially replicated.

The interaction among the servers or other system components is regulated by the group policies. In order to globally coordinate the execution of transactions, the participation of more than one server is required. Communication is established among the instances of the replication protocol running in different nodes of the system group. These policies control any procedure involving available replicas used to coordinate them with regard to each transaction, i.e., any synchronization (real or logical) between the system nodes required for achieving replica consistency. We distinguish four intervals in the transaction lifetime when different group policies may be applied: *start* (at the start of transaction, before the first access), *life* (during the lifetime of the transaction, e.g., per-operation communications), *end* (at the end of transaction, before the commit operation), and *after* (after termination, i.e., after returning the results to the client).

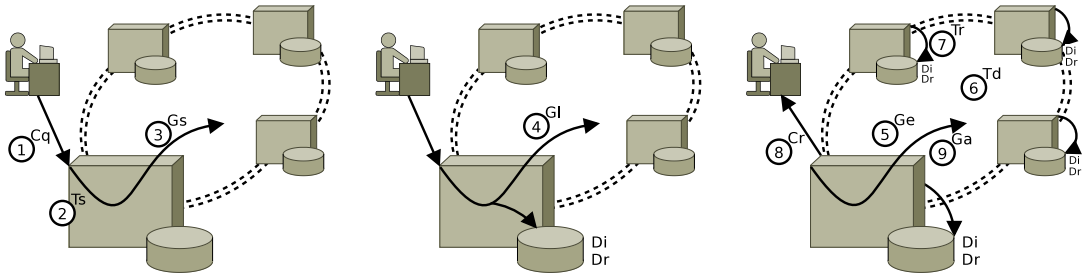
The interaction between the system and the user transactions is regulated by the transaction policies. The *service* policy ensures that the necessary conditions (apart from the obvious resource requirements) hold for the system to serve an incoming request, accepting a transaction for processing. Transaction execution can be split into two phases: the local phase, in which they are only executed in the delegate server (either interactively or not); and the remote phase, in which their execution spans to the rest of replicas, after some kind of coordination between the system nodes. Right before starting the remote execution, the *decision* policy determines the procedure followed to decide which transactions will commit and which ones must be aborted. If the system decides to commit the transaction, the *remote* policy controls the way the transaction is sent to the database to be applied.

Some *glue* procedures will be usually needed to chain the previous policies in a way that guarantees a correct behavior and the isolation level promised by the replication system.

According to our proposal, the lifetime of a transaction is thus controlled by different policies depending on its current execution step (see Figure 1). First, when the client sends its request to the system (identified as interaction 1), the client-request policy determines to which servers this request must be addressed. Once in the appropriate server (or servers), the processing of the transaction is accepted as determined by the transaction-service policy (interaction 2). Once the transaction is accepted for processing, the group-start policy defines if some coordination must be done with the rest of replicas prior to transaction start (interaction 3, e.g., broadcast the start of transaction to the rest of nodes in order to get a global common starting point). After this starting coordination (if it exists), the transaction *enters* the database for the first time, beginning its local execution phase. Database properties affect the transaction execution since this moment on. The database-replication policy will define if there is a copy of the data in the current server or whether the transaction must be distributed among several nodes. At each local DBMS, accesses to the database are controlled by the database-isolation policy. During the local execution phase, a group-life policy may apply, defining a linear coordination among servers (interaction 4). After all operations have been completed, the client asks for the commitment of the transaction.[5] Then, a new communication can be established, following the group-end policy (interaction 5). Prior to transaction termination, a decision process controlled by the transaction-decision policy inspects the transaction and decides if it can be committed or not (interaction 6). If the decision is positive, the transaction enters its remote phase in all nodes. The transaction-remote policy determines when the transaction can access the local database (interaction 7), where it will be applied according to the database-isolation policy. After transaction completion, the client-response policy regulates the sending of transaction results to the user (interaction 8). Finally, a group-after policy may apply (interaction 9), as in lazy systems.

The sequence of interactions presented above may be adapted to different ways of transaction execution, by selecting the proper strategy to execute each interaction or by denoting that certain interaction will not take place.

[5] It is also possible that the client issues an abort operation. In that case, following steps aim to rollback all executed operations instead of committing them.



a. Start of transaction b. Life of transaction c. End of transaction

Figure 1: Policies applied during the lifetime of a transaction. Interactions are numbered following the sequence of their first execution.

Table 1 presents the entire classification of the strategies we have found in existing systems for each policy. Identifiers are given to each strategy. This way, we can easily say that a given replication system follows, e.g., the group start policy Gs0, to represent that no communication is established among the group of servers before transaction start. Note that a greater digit in the identifier means a greater effort or a stricter criterion. Now we offer a detailed description for each policy, following the order on which they regulate the transaction lifetime.

Client-request policy Firstly, the client should address its request to the system, directly communicating with one or more elements. Depending on the characteristics of the system, this request may be forwarded to other elements or redirected to a special component by means of another component acting as a proxy (such as a load balancer or scheduler). This policy thus defines the set of servers that finally receive the original user request for processing. When any server is capable of processing a user request (Cq1), it is commonly the client itself who selects the closest site and sends its request directly. If only one server can process a given user request (Cq2) due, e.g., to some ownership criteria (the primary copy in the system or the server that controls the portion of the data that the user needs to access) or to the decision of a non-trivial scheduler that selects one specific node (not just the least loaded one but one satisfying certain condition), then the user request must be forwarded to this special server. If no kind of request redirection is performed by the system, then the client itself must know how to select the specific node. It is also possible that the user request must arrive to several servers in the system: either a quorum of nodes (Cq3) or the entire group (Cq4). In this last case, some ordering guarantees may be necessary (as in the group policies, as explained below). Thus, a letter 't' appended to the strategy identifier indicates that the multicast must follow total order, which is a usual option for the processing of active transactions. In any case, each system will provide the required mechanisms for implementing this policy.

Transaction-service policy Once the transaction arrives to a node specified by the client-request policy, it enters some sort of queueing system, where it waits for the protocol running in that node to start serving it. This policy reflects the existence of any necessary, non-obvious condition for the node to continue processing incoming requests in general, or this specific request in particular. Waiting for the necessary computational resources (idle threads, available connections to the database, etc.) is considered trivial and included in the default bottom policy (Ts0, immediate service). When the necessary resources are available but any other conditions temporarily prevent the system from processing a transaction, the service is deferred (Ts1). This condition must be locally evaluable, without the participation of other nodes (when a cooperation with the rest of nodes is required to start a transaction, this is reflected in the group-start policy). For example, there may be situations where the node must postpone all incoming requests, e.g., after detecting some inconsistency on the data and until it undergoes reconciliation; or postpone the processing of a query until all pending remote transactions are applied in the node. In a more complex situation, the data could be divided into conflict classes and incoming requests appended to several conflict queues, depending on the data they needed to access. In this scenario, only when the transaction were at the first position in all its queues, it would fulfill the condition to be processed by the system.

Table 1: Available strategies for each policy

Policy family	Policy	Id	Strategy
Client policies (C)	Request (q)	Cq1	any server
		Cq2	special server
		Cq3	quorum of servers
		Cq4[a]	all servers
	Response (r)	Cr1	one answer
		Cr2	multiple answers
Database policies (D)	Replication (r)	Dr1	partial replication
		Dr2	full replication
	Isolation (i)	Di0	undefined
		Di1	read committed
		Di2	snapshot
		Di3	serializable
		Di4	customized
Group policies (G)	Start (s), life (l), end (e), after (a)	Gx0	no communication
		Gx1[b]	one server [0..1]
		Gx2[b]	several servers [0..n]
		Gx3[b]	all servers
Transaction policies (T)	Service (s)	Ts0[c]	immediate service
		Ts1[c]	deferred service
		Ts2	no local service
	Decision (d)	Td0	no decision
		Td1[d]	one server
		Td2[d]	each server
		Td3[d]	quorum-based
		Td4[d]	agreement-based
	Remote (r)	Tr0	no remote execution
		Tr1[e]	concurrent
		Tr2	non-overlapping

[a] An appended *t* indicates a broadcast in total order.
[b] *n*, no order requirements; *f*, FIFO order; *t*, total order.
For Gs and Gl: *a*, asynchronous; *s*, synchronous.
[c] *n*, no interactivity.
[d] *r*, readset required; *w*, writeset required.
[e] *p*, controlled by the protocol; *d*, controlled by the database.

These two cases, the immediate and the deferred service, apply to all transactions that have a local execution in their delegate prior to their remote execution phase: either interactive transactions (where the user sends each transaction operation separately to the database, getting intermediate results as operations are completed), or service requests (calls to stored procedures). In the case of interactivity, it is possible that also intermediate operations of a transaction (not only the first one) are subjected to wait. We extend the concept of deferred service to model also those cases. In order to highlight the non-interactive cases, a letter 'n' will be appended to the strategy identifier.

In distributed (partitioned) and partially replicated databases, the transaction-service policy controls the creation of local subtransactions in each of the participating nodes.

Finally, there are also situations where a transaction is not *locally* –individually– processed by any node, but rather has an active execution in all sites at the same time (generally, this precludes interactivity and is mostly used for service requests). We therefore consider that an active transaction has only *remote* phase because, since its starting point, its execution spans all available servers, i.e., there is no previous phase where it is locally executed by one delegate. However, this could be also considered the other way

around: as the remote execution of transactions is usually based on the application of logs or writesets (previously created by a delegate node which carried out all the transaction operations), we could say that active transactions are *locally* executed by all nodes and thus have no remote phase. However, we select the first approach –only remote phase– and consider that, in these cases, a policy of no local execution (Ts2) applies. Note that the digit of this last identifier is greater than the previous ones, as an active processing of transactions, where all nodes must perform all operations, is generally more costly than having a local execution phase and a later writeset propagation and application.

The two last details, i.e., subtransactions of distributed transactions and active execution, are denoted in Figure 2. In this diagram, which provides a visual representation of the applied policies during the local and the remote phases of the transaction lifetime, the horizontal line is time and it increases rightward.

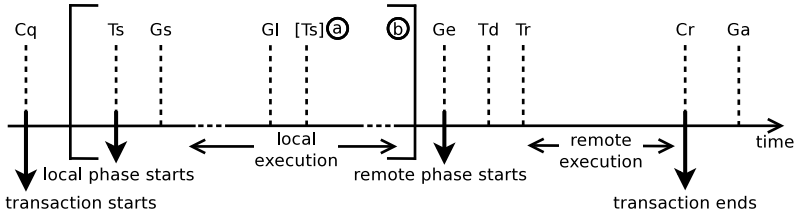


Figure 2: Interactions defining the local and remote portions of a transaction. Non-distributed transactions do not initiate subtransactions in multiple nodes (a). Active transactions do not present local phase (b).

Group-start policy Before starting a transaction, some coordination among nodes may be necessary. This communication will commonly include some global identifier for the transaction, in order to establish a synchronization point before any operation is executed by the transaction. Client-request and group-start policies may seem identical but they present a crucial difference: servers that receive the client request, as expressed in the client-request policy, generally process that request in the same way, i.e., they usually have all the same role regarding to that transaction; while servers contacted at transaction start, as defined in the group-start policy, generally play a different role from that of the first set.

To perform this first coordination, the communication among nodes may or may not need to be synchronous, halting or not the processing of the transaction until some condition holds (e.g., the message is delivered or all replicas reply to the sender).[6] Network communication involves some cost, particularly when some safety or ordering guarantees are required. Thus, an asynchronous communication allows the overlap of the communication cost with the transaction processing, while synchronous communication does not. To distinguish between both situations, an 'a' appended to the policy identifier will denote asynchronous communication, while a 's' will mean the need for synchrony.

All policies of the group family share a common set of strategies, which define the number of servers or other system components the local node must contact with. In the trivial case, no coordination is done (Gx0, where 'x' is 's' for group-start strategies, 'l' for group-life ones, 'e' for group-end options, and 'a' for group-after strategies), e.g., in active replication, the synchronization point is at the beginning of the transaction, so no further synchronization is needed at the end. Non-trivial strategies require the participation of at most another server (Gx1), of a subset of the entire group (Gx2), or the participation of all system nodes (Gx3). In order to implement group coordination, different communication primitives are used. A simple option is the use of node-to-node messages, e.g., some gossip, flooding or cascade mechanisms can be implemented this way. This option was commonly used in initial systems, when GCSs were not yet used for inter-node communication. More complex primitives include reliable multicasts and broadcasts, with or without order requirements. Multicasts to a quorum are used in quorum-based systems. Reliable broadcast is enough when only one node acts as a sender or when other mechanisms already provide all the order requirements of the system. FIFO or total order[7] broadcasts may be necessary. When a specific ordering

[6] Note that although such wait is only necessary in the sender node, it is also part of the required initial coordination, and thus, of the group-start policy.

[7] Virtually all GCSs include FIFO guarantees in their total order primitives, so in practice it is assumed that abcast messages respect FIFO ordering.

guarantee must be provided, an ‘f’, for FIFO, or a ‘t’, for total, is appended to the strategy identifier. An ‘n’ will denote that no ordering guarantees are needed.

Database-replication policy Apart from purely distributed systems with no replication at all, not considered in this work, replicated databases may enforce replication at different degrees. When each node stores a complete copy of the database, the system features full replication (Dr2). Otherwise, a partial replication is maintained (Dr1).

Database-isolation policy Whenever a transaction is being executed in a node (local interactive phase, local non-interactive execution, remote execution and writeset application, and final commit phase), a certain isolation level is enforced in the local database: read committed (Di1), snapshot isolation (Di2), serializable (Di3) or a customized level (Di4), achieved out of the DBMS by directly controlling the locks or making any other management. An additional undefined strategy (Di0) represents that the isolation level was not specified in the system description. Upon the isolation provided in the local database, the replication system is able to enforce certain *global* isolation level for all transactions running in the system.

Group-life policy During the lifetime of a transaction, while it submits operations to the database, some coordination among nodes may be required. When such a coordination exists, it is usually done before or after each single operation (e.g., each SQL statement), sending information about it for, e.g., acquiring locks in remote replicas. Similarly to the group-start case, the execution flow of the transaction may or may not be suspended until this coordination is completed. An appended ‘a’ (‘s’) will denote asynchronous (synchronous) communication.

Group-end policy When the transaction finishes submitting operations, and upon request of commitment, a global coordination is usually needed. During this communication, transaction information, like the readset, writeset and updated values, is commonly spread among system nodes. Sometimes, several rounds are required for appropriate coordination. This is the case of the two-phase or the three-phase commit protocols.

Transaction-decision policy After the transaction has completed its operations and the group-end coordination has been made, a validation process may be run to decide the final outcome of a transaction, i.e., its abortion or commitment, depending on certain conditions. This policy determines which server or servers, if any, are responsible for taking this decision, i.e., for running the decision process. When the rest of the policies is enough –and especially for read-only transactions, which are usually immediately committed in relaxed correctness criteria–, no decision process is executed (Td0). Otherwise, the process may be executed by only one server (Td1), commonly the delegate, which later sends the decision to the rest of nodes; or be performed by each server (Td2) in a symmetric, independent and deterministic way (certification). The process can involve also the collaboration of multiple nodes. This is the case of decisions based on a consensus among a quorum of nodes (Td3), where each server of the quorum informs whether it agrees or not to commit the current transaction; and decisions based on an agreement among all the (available participating) sites (Td4). This latter policy is used when performing a two- or a three-phase commit, where each server says if it agrees to commit the current transaction.

The decision about the final outcome of transactions is usually based on conflicts although it can be also based on some other information (e.g., temporal criteria). When based on conflict checking, an ‘r’ (respectively, a ‘w’) after the policy identifier will indicate the use of readsets (writesets) during the decision process. It is important to note the possible different uses of readsets and writesets, which affect performance at different degrees. Thus, decision may be based on conflicts but delegated to the local DBMS (low cost); or it may be necessary to collect those sets and inspect them at middleware level (medium cost); or even to forward them to other servers (high cost).

A final consideration must be done about the decision process. Although normally it is run upon writeset delivery in order to decide the outcome of such a writeset based on the conflicts with previously delivered transactions, in some systems it is the delivered writeset which, during the decision process run upon its delivery, may cause the abortion of other (local) transactions that, although already broadcast, will

be delivered afterwards. Thus, a decision process is run but not for deciding the outcome of the current writeset, but that of future writesets, in a sort of *early* decision.

Transaction-remote policy After completing the local execution phase and getting a positive decision, transactions start their remote execution. For this, transaction information must be somehow provided to every system node. As this is usually done through a GCS, we refer to this information as a delivered transaction, a delivered writeset or, simply, a writeset. At a given node N_i, when the delivered transaction is local, i.e., N_i is its delegate node, only the commit operation is pending. Otherwise, the writeset is remote and its updates must be applied in the local database of N_i prior to final commitment.[8] The access to the local database is controlled by the transaction-remote policy. Two main strategies are considered: either multiple transactions are sent concurrently to the database (Tr1), thus improving performance, or they are sequentially sent, one at a time, following a non-overlapping policy (Tr2) where each delivered transaction must wait for the completion of the previous one. In the first case, conflicting operations must be controlled in order to maintain replica consistency. This control may be performed by the protocol (e.g., the protocol checks for conflicts between writesets before sending multiple, non-conflicting transactions to the database) or by the concurrency control of the database management system (e.g., transactions set write locks in an appropriate sequence before accessing the database). A letter after the identifier specifies if the control is made by the protocol ('p') or by the database ('d').

A third strategy represents the cases where no remote execution is performed (Tr0), namely for read-only transactions that are executed only in their delegate server.

Another aspect that should be mentioned here is the need to abort local transactions, in their local execution phase, holding locks or otherwise preventing remote transactions from being applied in the local database. Such a process is required for protocol liveness but details about its implementation are rarely given in publications. This *clearing* process differs from the early decision commented above in that the clearing process aborts transactions that were completely local to the running node (i.e., other nodes had no knowledge of their existence), while an early decision may abort transactions that, although local to the running node, were already broadcast to other nodes. Therefore, in an early negative decision, it will be necessary to broadcast this outcome to remote nodes.

Client-response policy After transaction completion, the client must receive the results of its request. Either one or multiple replies can be sent to the client. Commonly, only the delegate server (or some special node or component in the system) replies, so the client receives only one answer (Cr1). In other cases, multiple replies arrive to the client (Cr2). This distinction is important as, in the latter case, the client has to perform some kind of procedure to select the final answer (the first received, a combination of multiple replies, the most voted, etc.).

Group-after policy After sending the response to the client, once the transaction has committed in one or several nodes, a last coordination may be needed, e.g., for updating remote nodes in lazy systems.

3.2 Correctness Criteria for Replicated Databases

Correctness in replicated databases comprises two characteristics: (a) the isolation level, responsible for the isolation among all concurrent transactions being executed in the system; and (b) the replica consistency, or the degree of admissible divergence among the states of all replicas [43]. The first characteristic is provided by means of a local DBMS in each server and by using certain validation rules at replication protocol level. The second aspect is enforced by the replication protocol and involves synchronization among replicas, which can be made easy by means of a group communication tool. Based on the concepts and conclusions of Ruiz-Fuertes and Muñoz-Escoí [55], we consider the correctness criteria of Table 2 for one-copy equivalent systems. The user-centric consistency (i.e., the replica consistency as perceived by users as opposed to the server-centric consistency) level where inversions may arise is considered the standard level, while precluding inversions requires a higher effort. For criteria based on isolation levels

[8] In the case of partially replicated databases, only the updates corresponding to items stored in the node must be applied when processing the delivered writeset.

other than serializability, we choose similar names to those proposed by Ruiz-Fuertes and Muñoz-Escoí [55]: e.g., in the case of snapshot isolation, 1ASI corresponds to systems that preclude inversions, 1SI+ executions ensure the absence of inversions within sessions, and 1SI allows the appearance of inversions. Note that in the context of snapshot isolation, an inversion may occur if a transaction (either a query or an update transaction) is provided with a snapshot which does not correspond to the latest available snapshot in the system, as created by the last committed transaction. If not precluded, either conservatively or optimistically, inversions may appear and a committed transaction T may have read an old value of a data item that was updated by a transaction that committed before the start of T.

Table 2: Correctness criteria for one-copy equivalent replicated databases

Criterion	Isolation	Consistency	Short description
1ASR	serializable	no inversions	The effects of transactions are equivalent to a serial execution in only one node. At each single moment, the committed information in every server is exactly the same from the point of view of clients: a user can execute a transaction in one node and change immediately to another server where they will see the updates made by their previous transaction. Strong serializability [11, 17, 69] is another name used in the literature to refer to this correctness criterion.
1SR+	serializable	no inversions on sessions	Consistency is more relaxed than in the previous case, but inversions are precluded within client sessions, so a user with a single session perceives an inversions-free view of the database. Strong session serializability [17, 69] is also used in the literature to refer to this criterion.
1SR	serializable	inversions	The effects of transactions are equivalent to those of a serial execution in one node. But at a given moment, effects of some transactions may be pending to commit in a server and, thus, a user moving between servers may get inconsistent results.
1ASI	snapshot	no inversions	Transactions are isolated following the snapshot level. Each transaction T gets the latest snapshot of the entire system (conservative approach) or, at least, the latest snapshot as created by previous transactions that updated data items that T reads (this allows an optimistic approach were transactions are restarted if a conflict is detected). This level is also named conventional snapshot isolation (CSI) [20] or strong SI [18].
1SI+	snapshot	no inversions on sessions	Transactions are isolated under the snapshot level. The snapshot provided to a transaction T corresponds to the latest snapshot created by transactions on the same client session (conservative approach) or, at least, by transactions on the same client session that write data items that T reads (which allows an optimistic approach). This level is also named strong session SI [18].
1SI	snapshot	inversions	Transactions are isolated following the snapshot level. But the snapshot provided to a transaction may be arbitrarily old, due to some transactions pending to commit in its delegate node (i.e., inversions occur). This level is also named generalized snapshot isolation (GSI) [20]. Usually, transactions get the latest snapshot of their delegate server, which is also known as prefix-consistent snapshot isolation (PCSI) [20].
1CS	cursor stability	inversions	Cursor stability is enforced in the nodes. This isolation level prevents lost updates for rows read via a cursor. Inversions may arise.
1ARC	read committed	no inversions	The database replication system behaves as only one copy providing read committed isolation. Transactions starting at different nodes at the same time see the same database state.
1RC	read committed	inversions	A read committed isolation level is guaranteed. Inversions may arise.

4 A Comprehensive Survey of Database Replication Systems

In this section we present the research evolution and survey the state of the art of database replication techniques. Our analysis is based on the policy-based characterization model presented in Section 3. Over 50 different systems are fully characterized following this model.

The rest of the section is structured as follows. Subsection 4.1 presents the comprehensive and chronological survey of database replication systems. Subsection 4.2 comments about the scope of the characterization model, as observed during the preparation of this survey. Finally, Subsection 4.3 discusses about the insight the survey offers.

4.1 Replication Systems as Combinations of Strategies: A Survey

Any replication system can be defined as a particular combination of strategies, i.e., a set of specific policies. Obviously, not all combinations will create correct or useful replication systems. Some systems proposed in the literature of database replication are chronologically listed in Table 3, detailing the followed strategies, which are identified as shown in Table 1. The correctness criterion is also specified (see Table 2). For simplicity, when detailing communication processes involving several rounds, only the most demanding is showed in the table (and thus, e.g., a total order requirement signaled in the strategy may be only needed in one of the rounds). In Table 3, when a system (row) follows different strategies for a specific interaction (column) for different types of transactions, these strategies appear at different lines within that *cell* of the table (column-row). Those types of transactions denote usually the difference between read-only and update transactions, or among the several correctness criteria supported by a given system.[9] Whenever multiple lines are present in a row, columns with only one value mean that such a strategy is shared by all transaction types.

A visual *radial* representation for each surveyed system is provided in Figure 3, also in chronological order. Each *radius* of a graph corresponds to a policy, labeled with its initial letters. Concentric *circles* (hendecagons, to be accurate), mark the scale from 0 (the most inner circle), to 4 (the most external one). The digit associated with the strategy followed by the depicted system for each interaction of each different transaction type is then represented in that scale. Whenever several options are possible for a specific policy (e.g., when users can choose between read committed or snapshot isolation), the least costly option is the one that is represented in the graph (read committed in such a case), thus showing the minimum requirements of the system. Those points are finally connected by lines in order to create a figure for each transaction type of the system.[10] Remember also that the more demanding the strategy, the greater the digit of its identifier (e.g., a group-start policy Gs0 denotes the absence of communication at transaction start, while Gs3 requires a synchronization with all servers). This way, the *bigger* the resulting figure, the more costly the execution of that transaction type. These representations allow us to visually compare different systems as well as to get an idea of their cost. For example, regarding communication costs, all policies involving communication (Cq, Gs, Gl, Ge and Ga) are grouped together in the eastern/northeastern zone of the graph (from 12 until 4:30 in a clock). A figure widening out in that zone depicts a system which relies on communication and thus its performance will depend on the GCS and the network. On the other hand, regarding database requirements, the radius of Di (database-isolation policy) allows a quick comparison between the strictness in the local isolation level required for the correct functioning of different systems.

Next we offer thorough descriptions for all the surveyed protocols and systems. Letters in brackets reference superindexes in the corresponding row of Table 3.

Alsberg and Day [2] proposed a protocol following the single primary, multiple backup model, where backups are linearly ordered. The client can address its request to any replica in the system, which will

[9] When possible, the strategies at the same line represent the same type of transaction, showing, e.g., the policies for read-only transactions in the first line and those for updates in the second one. However, for more complex cases (e.g., systems distinguishing not only between queries and updates but also among different correctness criteria), Table 3 still depicts all followed strategies but such *one-line-one-type* clarification is not made. Please refer to the textual description and the visual representation of Figure 3 of those complex systems for a detailed distinction among their transaction types.

[10] Those types are labeled 'q' for queries, and 'u' for update transactions. When different correctness criteria are provided, a distinction is made in parentheses.

Table 3: Database replication systems expressed as combinations of strategies

	Client request	**Transaction** service	**Group** start	**Database** replication	**Database** isolation	**Group** life	**Group** end	**Transaction** decision	**Transaction** remote	**Client** response	**Group** after	**Correctness criterion**
Alsberg-Day [2]	Cq2[a]	Ts0	Gs0	Dr2	Di0[b]	Gl0	Ge1-n[c]	Td0[d]	Tr2[e]	Cr1[f]	Ga1-n[g]	1A-[h]
2PL & 2PC [24]	Cq1	Ts0	Gs0	Dr1	Di3[a]	Gl1-n-s[b] Gl2-n-s[c]	Ge2-n[d]	Td4-rw[e]	Tr0 Tr1-d[f]	Cr1	Ga0	1ASR[g]
BTO & 2PC [7]	Cq1	Ts0	Gs0	Dr1	Di4[a]	Gl1-n-s Gl2-n-s	Ge2-n	Td4-rw	Tr0 Tr1-d	Cr1	Ga0	1ASR
Bernstein-Goodman [9]	Cq1	Ts0	Gs0	Dr1	Di3	Gl1-n-s[a] Gl2-n-s[b]	Ge2-n	Td4-rw	Tr0 Tr1-d	Cr1	Ga0	1ASR
OPT & 2PC [61]	Cq1	Ts0	Gs0	Dr1	Di4[a]	Gl1-n-s	Ge2-n[b]	Td4-rw[c]	Tr0 Tr1-d[d]	Cr1	Ga0	1ASR
O2PL & 2PC [12]	Cq1	Ts0	Gs0	Dr1	Di3	Gl1-n-s	Ge2-n[a]	Td4-rw	Tr0 Tr1-d	Cr1	Ga0	1ASR
Bcast all [1]	Cq1	Ts0	Gs0	Dr2	Di3	Gl3-t-s	Ge3-t	Td0	Tr1-d[a]	Cr1	Ga0	1ASR
Bcast writes [1]	Cq1	Ts0	Gs0	Dr2	Di3	Gl0[a] Gl3-t-s	Ge3-t	Td1-rw	Tr0[b] Tr1-d	Cr1	Ga0	1ASR
Delayed bcast wrts [1]	Cq1	Ts0	Gs0	Dr2	Di3	Gl0	Ge3-t	Td1-rw	Tr0 Tr1-d	Cr1	Ga0	1ASR
Single bcast txns [1]	Cq1	Ts0	Gs0	Dr2	Di3	Gl0	Ge0 Ge3-t	Td1-rw[a] Td2-rw	Tr0 Tr1-d	Cr1	Ga0	1SR
Lazy Txn Reordering [50]	Cq1	Ts0	Gs0	Dr2	Di2	Gl0	Ge3-t	Td2-rw	Tr2	Cr1	Ga0	1SR
OTP-99 [36]	Cq1[a] Cq4-t[b]	Ts0-n[c] Ts2[d]	Gs0	Dr2	Di0[e]	Gl0	Ge0	Td0	Tr0[f] Tr1-p[g]	Cr1	Ga0	1SR
Fast Refresh Df-Im [45]	Cq1[a] Cq2[b]	Ts0	Gs0	Dr1	Di3[c]	Gl0	Ge0	Td0[d]	Tr0[e] Tr1-d[f]	Cr1	Ga0[g] Ga2-f[h]	1SR
Fast Refresh Im-Im [45]	Cq1 Cq2	Ts0	Gs0	Dr1	Di3	Gl0[i] Gl2-f-a[j]	Ge0[k] Ge2-f[l]	Td0	Tr0 Tr1-d	Cr1	Ga0	1SR
DBSM [47]	Cq1	Ts0	Gs0	Dr2	Di3	Gl0	Ge0[a] Ge3-t[b]	Td0[c] Td2-rw[d]	Tr0 Tr1-d[e]	Cr1	Ga0	1SR
SER [35]	Cq1	Ts0	Gs0	Dr2	Di4[a]	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td1-rw[e]	Tr0 Tr1-d[f]	Cr1	Ga0	1SR
CS [35]	Cq1	Ts0	Gs0	Dr2	Di4[a]	Gl0	Ge0 Ge3-t	Td0 Td1-rw	Tr0 Tr1-d	Cr1	Ga0	1CS
SI [35]	Cq1	Ts0	Gs0	Dr2	Di2	Gl0	Ge0[a] Ge3-t[b]	Td0[c] Td2-w[d]	Tr0 Tr1-d[e]	Cr1	Ga0	1SI
Hybrid [35]	Cq1	Ts0	Gs0	Dr2	Di2[a] Di4[b]	Gl0	Ge0[c] Ge3-t[d]	Td0[e] Td1-rw[f]	Tr0 Tr1-d[g]	Cr1	Ga0	1SR
NODO [46]	Cq1[a] Cq4-t[b]	Ts1[c]	Gs0	Dr2	Di2	Gl0	Ge0[d] Ge3-n[e]	Td0[f]	Tr0[g] Tr1-p[h]	Cr1	Ga0	1SR
REORDERING [46]	Cq1 Cq4-t	Ts1	Gs0	Dr2	Di2	Gl0	Ge0 Ge3-f[i]	Td0	Tr0 Tr1-p	Cr1	Ga0	1SR
Pronto [48]	Cq2[a]	Ts0	Gs0	Dr2	Di3	Gl0	Ge3-t[b]	Td2[c]	Tr2	Cr2[d]	Ga0	1ASR
DBSM-RAC [63]	Cq2[a]	Ts0	Gs0	Dr1	Di3	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td4-rw[e]	Tr0[f] Tr1-d[g]	Cr1	Ga0	1SR
Epidemic restricted [30]	Cq1	Ts0	Gs0	Dr1	Di3	Gl0	Ge0[a] Ge2-n[b]	Td0[c] Td2-rw[d]	Tr0[e] Tr2[f]	Cr1	Ga0	1SR
Epidemic unrestrict. [30]	Cq1	Ts0	Gs0	Dr1	Di3	Gl2-n-s[g]	Ge0 Ge2-n	Td0 Td2-rw	Tr0 Tr2	Cr1	Ga0	1SR
OTP [37]	Cq4-t[a]	Ts2[b]	Gs0	Dr2	Di0[c]	Gl0	Ge0	Td0	Tr1-p[d]	Cr1	Ga0	1ASR

Continued on next page

Table3 – continued from previous page

	Client request	**Transaction** service	**Group** start	**Database** replication	**Database** isolation	**Group** life	**Group** end	**Transaction** decision	**Transaction** remote	**Client** response	**Group** after	**Correctness criterion**
OTP-Q [37]	Cq1[e] Cq4-t	Ts1[f] Ts2	Gs0	Dr2	Di0	Gl0	Ge0	Td0	Tr0[g] Tr1-p	Cr1	Ga0	1SR
OTP-DQ [37]	Cq1 Cq4-t	Ts1 Ts2	Gs0	Dr2	Di0	Gl0	Ge0	Td1-rw[h] Td0	Tr0 Tr1-p	Cr1	Ga0	1SR
OTP-SQ [37]	Cq1 Cq4-t	Ts0[i] Ts2	Gs0	Dr2	Di0	Gl0	Ge0	Td0	Tr0 Tr1-p	Cr1	Ga0	1SR
RJDBC [23]	Cq1[a]	Ts0	Gs0	Dr2	Di0[b]	Gl3-t-s[c]	Ge3-t[d]	Td0[e]	Tr2[f]	Cr1	Ga0	1-[g]
RSI-PC [51]	Cq1 Cq2[a]	Ts1[b] Ts0[c]	Gs0	Dr2	Di2[d] Di1[e]	Gl0	Ge0	Td0[f]	Tr0 Tr2[g]	Cr1[h]	Ga0[i] Ga2-f[j]	1ASI 1SI 1ARC 1RC
SRCA [42]	Cq1[a]	Ts0	Gs0	Dr2	Di2	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td1-w[e]	Tr0 Tr2	Cr1	Ga0	1SI
SRCA-Rep [42]	Cq1	Ts1[a]	Gs0	Dr2	Di2	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td2-w[e]	Tr0 Tr1-p	Cr1	Ga0	1SI
DBSM* [68]	Cq1[a] Cq2[b]	Ts0	Gs0	Dr2	Di3	Gl0	Ge0 Ge3-t	Td0 Td2-w	Tr0 Tr1-d	Cr1	Ga0	1SR
PCSI Distr. Cert. [20]	Cq1	Ts0	Gs0	Dr2	Di2	Gl0	Ge0[a] Ge3-t[b]	Td0[c] Td2-w[d]	Tr0 Tr2	Cr1	Ga0	1SI
Tashkent-MW [21]	Cq1	Ts0	Gs0	Dr2	Di2	Gl0	Ge0[a] Ge1-n[b]	Td0[c] Td1-w[d]	Tr0[e] Tr2[f]	Cr1	Ga0[g]	1SI
Tashkent-API [21]	Cq1	Ts0	Gs0	Dr2	Di2	Gl0	Ge0 Ge1-n	Td0 Td1-w	Tr0 Tr1-d[h]	Cr1	Ga0	1SI
DBSM-RO-opt [44]	Cq1	Ts0	Gs0	Dr2	Di3	Gl0	Ge3-t[a]	Td1-rw Td2-rw[b]	Tr0 Tr1-d	Cr1	Ga0	1ASR[c]
DBSM-RO-cons [44]	Cq1	Ts0	Gs3-t-s[a] Gs0[b]	Dr2	Di3	Gl0	Ge0[c] Ge3-t[d]	Td0[e] Td2-rw[f]	Tr0 Tr1-d	Cr1	Ga0	1ASR
Alg-Weak-SI [18]	Cq1 Cq2[a]	Ts0	Gs0	Dr2	Di2	Gl0	Ge0[b]	Td0[c]	Tr0 Tr1-p[d]	Cr1	Ga0[e] Ga3-f[f]	1SI[g]
Alg-Str.-SI / Alg-Str.Ses.-SI [18]	Cq1 Cq2	Ts1[a] Ts0[b]	Gs0	Dr2	Di2	Gl0	Ge0	Td0	Tr0 Tr1-p	Cr1	Ga0 Ga3-f	1ASI 1SI+[c]
One-at-a-time / Many-at-a-time [58]	Cq2[a]	Ts0	Gs0	Dr1	Di3[b]	Gl0	Ge0[c] Ge3-t[d]	Td0[e] Td3-rw[f]	Tr0[g] Tr2[h]	Cr1	Ga0	1SR
k-bound GSI [4]	Cq1	Ts0	Gs3-t-a[a]	Dr2	Di2	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td2-w[e] Td1-rw[f]	Tr0 Tr2	Cr1	Ga0	1SR 1ASI 1SI
Tashkent+ [22]	Cq2[a]	Ts0	Gs0	Dr2[b]	Di2	Gl0	Ge0 Ge1-n	Td0 Td1-w	Tr0 Tr2	Cr1	Ga0	1SI[c]
Mid-Rep [33]	Cq1	Ts0[a] Ts1[b]	Gs0 Gs3-t-s[c]	Dr2	Di2	Gl0	Ge0 Ge3-t	Td0 Td1-rw	Tr0 Tr2[d]	Cr1	Ga0	1SR 1ASI 1SI
SIRC [56]	Cq1	Ts0	Gs0	Dr2	Di1 Di2[a]	Gl0	Ge0[b] Ge3-t[c]	Td0[d] Td2-w[e]	Tr0 Tr2	Cr1	Ga0	1SI 1RC
Serrano et al. [60]	Cq2[a]	Ts1[b]	Gs0	Dr1	Di2	Gl1-n-s[c]	Ge2-n[d] Ge3-t[e]	Td0[f] Td2-w[g]	Tr0[h] Tr2[i]	Cr1	Ga0	1SI
MPF/MCF [70]	Cq1[a] Cq2[b]	Ts0	Gs0	Dr2	Di3	Gl0	Ge0 Ge3-t	Td0 Td2-rw	Tr0 Tr1-d	Cr1	Ga0	1SR
WCRQ [53]	Cq1	Ts0	Gs0	Dr2	Di4[a]	Gl0	Ge2-n[b] Ge3-t[c]	Td3-rw[d]	Tr0 Tr2[e]	Cr1[f]	Ga0	1ASR
AKARA [16]	Cq1	Ts0-n Ts2[a]	Gs3-t-s[b]	Dr2	Di2	Gl0	Ge3-n[c] Ge0[d]	Td0	Tr2	Cr1	Ga0	1SI

Continued on next page

Table3 – continued from previous page

	Client request	**Transaction** service	**Group** start	**Database** replication	**Database** isolation	**Group** life	**Group** end	**Transaction** decision	**Transaction** remote	**Client** response	**Group** after	**Correctness criterion**
BaseCON for 1SR [69]	Cq1[a] Cq4-t[b]	Ts0[c] Ts2[d]	Gs0	Dr2	Di3[e]	Gl0	Ge0	Td0[f]	Tr0 Tr1-p[g]	Cr1[h]	Ga0	1SR
BaseCON for SC [69]	Cq2[a] Cq4-t[b]	Ts0[c] Ts2[d]	Gs0	Dr2	Di3[e]	Gl0	Ge0	Td0[f]	Tr0 Tr1-p[g]	Cr1[h]	Ga0	1SR+
BaseCON for strong 1SR [69]	Cq4-t[a]	Ts0 Ts2	Gs0	Dr2	Di3	Gl0	Ge0	Td0	Tr0 Tr1-p	Cr1	Ga0	1ASR
*g*B-SIRC [57]	Cq1	Ts0	Gs3-t-a[a]	Dr2	Di1 Di2[b]	Gl0	Ge0[c] Ge3-t[d]	Td0[e] Td2-w[f] Td1-rw[g]	Tr0 Tr2	Cr1	Ga0	1ASI 1SI 1RC

then forward it to the primary [*a*]. The proposed system aims at offering a resilient sharing of distributed resources but it is not specially tailored to any service. In particular, it is not tailored for database replication. Thus, some database-related points are not detailed in the paper, such as the database-isolation [*b*] or the transaction-remote policy [*e*] (for which a conservative, non-overlapping option is assumed in our survey). Authors focus on requests that change the state of the replicas, which can be regarded as update transactions. Processing is slightly different depending on whether the client directly addresses to the primary or not. In any case, a two-host resiliency is always ensured. The first backup receives the updates at the end of the transaction [*c*], before a unique reply is sent to the client [*f*]. After this, the rest of backups are updated in a cascade mode [*g*]: whenever a backup commits its new state, it forwards the updates to the following backup. As all transactions are executed in the primary (the rest of nodes only act as backups), the concurrency control of the primary server is enough and no decision process is necessary [*d*]. The resulting correctness criterion includes one-copy equivalence and the guarantee of no inversions (from the user point of view, there is only one centralized server that runs all requests). The exact correctness criterion directly depends on the local isolation of the primary replica [*h*]. If, for example, the isolation in the primary node is serializable, the correctness criterion will be 1ASR.

2PL & 2PC (Distributed Two-Phase Locking with Two-Phase Commit) was originally described by Gray [24]. 2PL is an algorithm for distributed concurrency control, intended for distributed databases where some degree of replication is also probable. This involves the use of distributed transactions. The underlying database provides a serializable level of isolation, with long read and write locks [*a*]. Once in the appropriate cohort [*b*], read operations set a read lock on the local copy of the item and read the data, but updates must be approved at all replicas before the transaction proceeds [*c*]. Thus, write locks are required on all copies in a pessimistic way. All locks are held until the transaction has committed or aborted. Deadlocks can appear and are solved by aborting the youngest transaction. A snoop process periodically requests waits-for information from all sites, detecting and resolving deadlocks. This process rotates among the servers in a round-robin fashion. A two-phase commit [*d*] is executed for each transaction that requests commitment. The initiating server (coordinator) sends a *prepare* message to all nodes. Each server then replies with a positive or negative message. If all messages are positive, then the coordinator sends a commit message. Every server confirms with another message the success of the commitment. If any of the replies in the first phase is negative, the coordinator sends an abort message and all servers report back to the coordinator once the abortion is complete. Decision is, thus, agreement-based [*e*]: aborts are possible due to deadlocks, node or disk failures, problems in the log, etc., so the first phase of 2PC achieves an agreement among the servers about the decision for the current transaction. In nodes holding copies of replicated items, subtransactions are initiated as in cohorts, but only for update transactions. Locks ensure that conflicting operations are not concurrently made [*f*]. The correctness criterion is 1ASR [*g*].

A similar protocol is Wound-Wait (WW), which was proposed by Rosenkrantz et al. [54]. The only difference with regard to distributed 2PL is the way in which deadlocks are handled. In WW, deadlocks are prevented by the use of timestamps. When a transaction requests a lock which is held by a younger

transaction (with a more recent initial start-up time), the youngest transaction is immediately aborted unless it is already in the second phase of its 2PC.

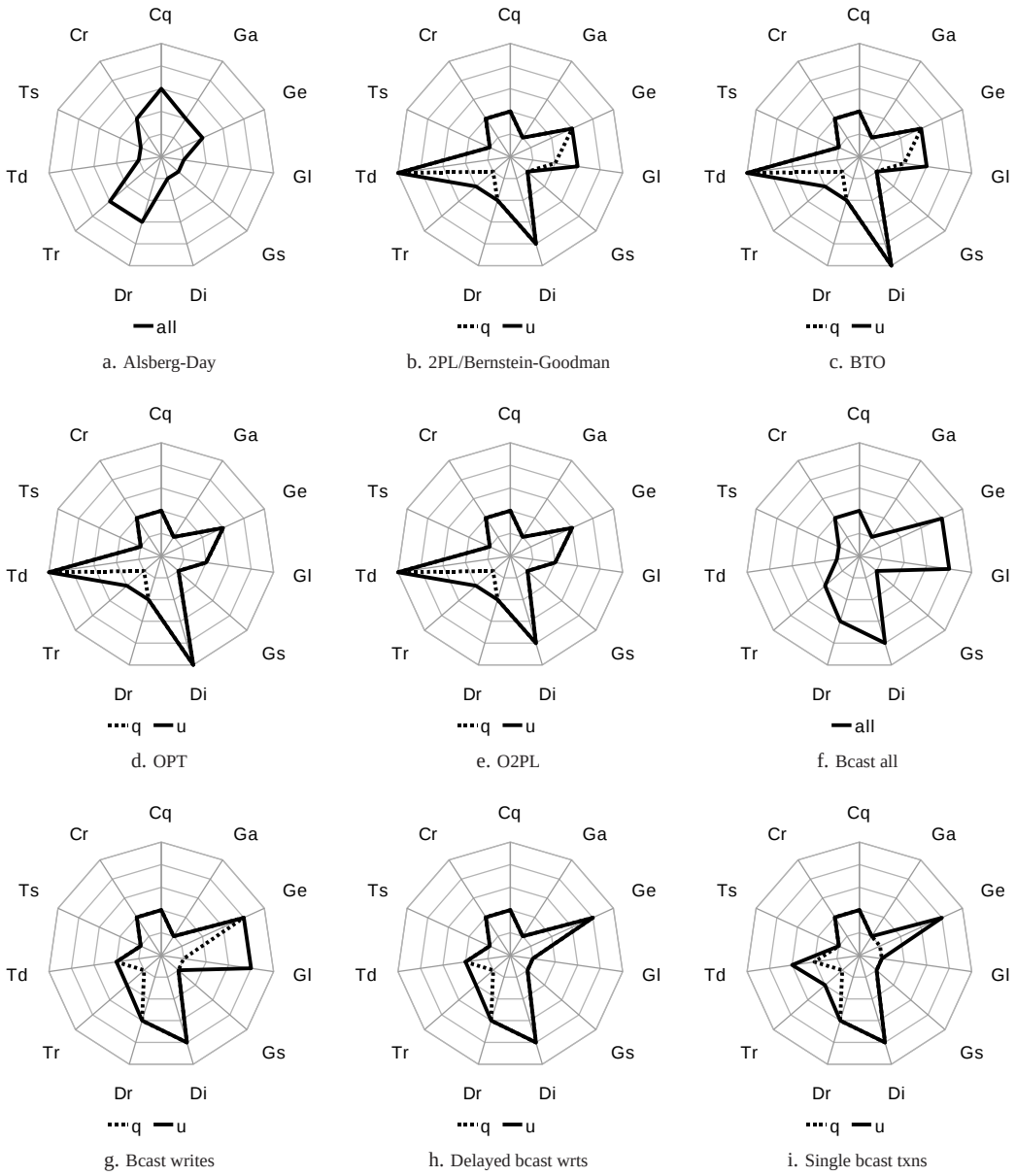
BTO & 2PC (Basic Timestamp Ordering with Two-Phase Locking) was originally proposed by Bernstein and Goodman [7]. BTO is identical to distributed 2PL except for the fact that local isolation is based on start-up timestamps [a]. Each data item has a read timestamp corresponding to the most recent reader, and a write timestamp corresponding to the most recent writer. When a transaction requests a read operation, it is permitted if the timestamp of the requester exceeds the write timestamp of the object. A write request is permitted if the timestamp of the requester exceeds the read timestamp of the item. In this case, if the write timestamp of the item is greater than the timestamp of the requester, the update is simply ignored. Write locks must be granted in all remote copies before proceeding with update operations, which are kept in private workspaces until commit time so that other writers are not blocked. On the other hand, approved read operations must wait until the precedent writes are applied in order during the commit operation of previous transactions. The used mechanisms enforce 1ASR.

Bernstein and Goodman later proposed a concurrency control algorithm [9] for achieving 1ASR in replicated distributed databases. This algorithm, which also employs 2PL and 2PC (and so the strategies of both algorithms match), is specially enforced to tolerate failures and recover nodes. Each data item x has one or more copies (x_a, x_b, ...). Each copy is stored at a site. Site failures are clean: when a site fails, it simply stops running (Byzantine failures are not considered); when it recovers, it knows that it failed and starts a recovery phase. Other sites in the system can detect when a site is down. Neither network partitions nor network failures are considered.

Each data item x has an associated directory $d(x)$, which can be replicated. Each copy of a directory contains two kinds of information: a list of the available copies of x, and a list of the available copies of $d(x)$. Special status transactions change the contents of the directories to reflect site failures. User transactions perform read and write operations over data items. Both types of transactions require an available copy of $d(x)$ for each access over x. Access to data items and directories are both protected by locking. Read and write locks over data items conflict in the usual way. New locks are created for accessing directories: din-locks, in-locks, ex-locks and user-locks. The three first are all conflicting among them. The last one, user-lock, is set by user transactions and it conflicts only with in-locks, being compatible with the rest of directory locks. Two-phase locking (2PL) is used for concurrency control.

There are three types of status transactions: directory-include, include and exclude. A directory-include transaction, *DIRECTORY-INCLUDE*(d_t), makes directory copy d_t available. It initializes d_t to the "current value" of $d(x)$ and adds d_t to the directory list of every available copy of $d(x)$ (din-locks are required in the original copy of the directory, in the new one and in all the updated copies). An include transaction, *INCLUDE*(x_a), makes data item copy x_a available. It first initializes x_a to the current value of x and then it adds x_a to the data-item list of every available copy of $d(x)$ (in-locks are requested in the local available copy and in all the updated copies of $d(x)$; also, a read-lock is set on the original data item copy of x and a write-lock protects the access to the new copy x_a). Finally, an exclude transaction, *EXCLUDE*(x_a), makes data item copy x_a unavailable (ex-locks are required in the original and in all the updated copies of $d(x)$). When executing any of these status transactions, if it is detected that some directory copy d_u has become unavailable, the transaction also removes d_u from the directory list of every available copy of $d(x)$. The distributed database system invokes exclude transactions when a site fails, and include and directory-include transactions when a site recovers. There is no directory-exclude transaction; d_t becomes unavailable the instant its site fails.

User transactions access data items in read and write operations. When a read operation is requested, the corresponding directory is accessed to find an available copy of x to read, which is then protected with a read-lock [a]. For a write operation, a user-lock is set on d_t, the local available copy of $d(x)$ (step 1). For each available data-item copy x_a, a write-lock is set [b] (step 2). Finally, all still available copies which were locked in the previous step are written (step 3). As sites may fail at any point in time, exclude transactions can be applied concurrently with user transactions. Copies that could be locked in step 2 but become unavailable before step 3 are ignored. Finally, when the user transaction reaches its locked point (when it owns all of its locks), the following procedure is executed: (1) for each read item x_a, if x_a is not in



a. Alsberg-Day b. 2PL/Bernstein-Goodman c. BTO

d. OPT e. O2PL f. Bcast all

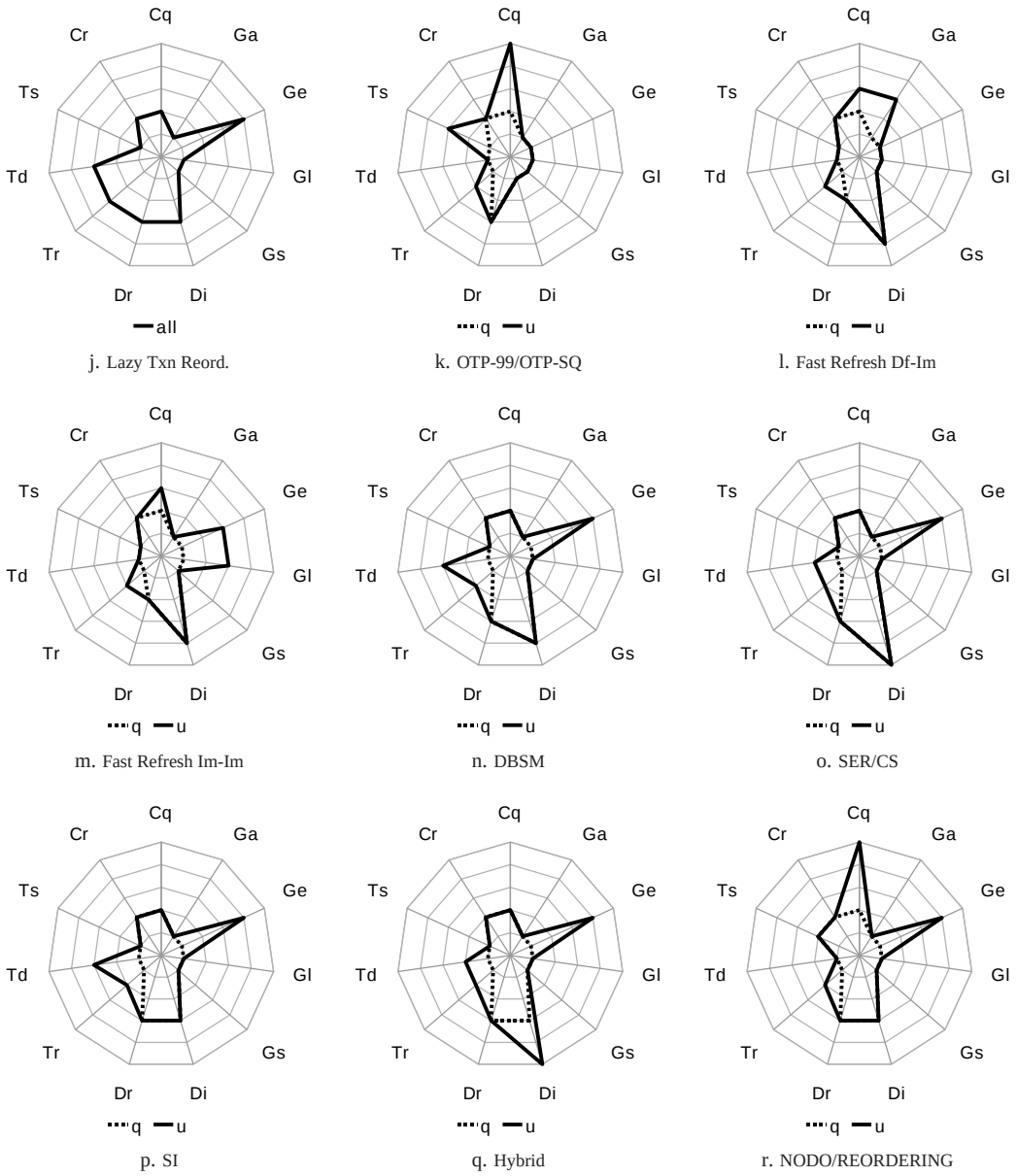
g. Bcast writes h. Delayed bcast wrts i. Single bcast txns

Figure 3: Visual radial representations of the surveyed systems

the data-items list of local directory copy d_t, or if $EXCLUDE(x_a)$ has an ex-lock on d_t, then the transaction is aborted. (2) In parallel, all user-locks and read-locks are released, and the step 3 of the write procedure is finished and all write-locks are also released.

A two-phase commit (2PC) procedure is used to commit transactions. The first phase of 2PC can run before the transaction reaches its locked point. However, phase 2 must wait until the end of the step 1 of the locked-point procedure. Phase 2 of 2PC and step 2 of the locked-point procedure can use the same messages. Due to the use of 2PL and 2PC, 1ASR is guaranteed.

OPT & 2PC (Distributed Certification with Two-Phase Commit) corresponds to the first of the two distributed concurrency control protocols proposed by Sinha et al. [61]. As in BTO, all data items have a read and a write timestamp corresponding to the most recent reader and writer, respectively [*a*]. However,



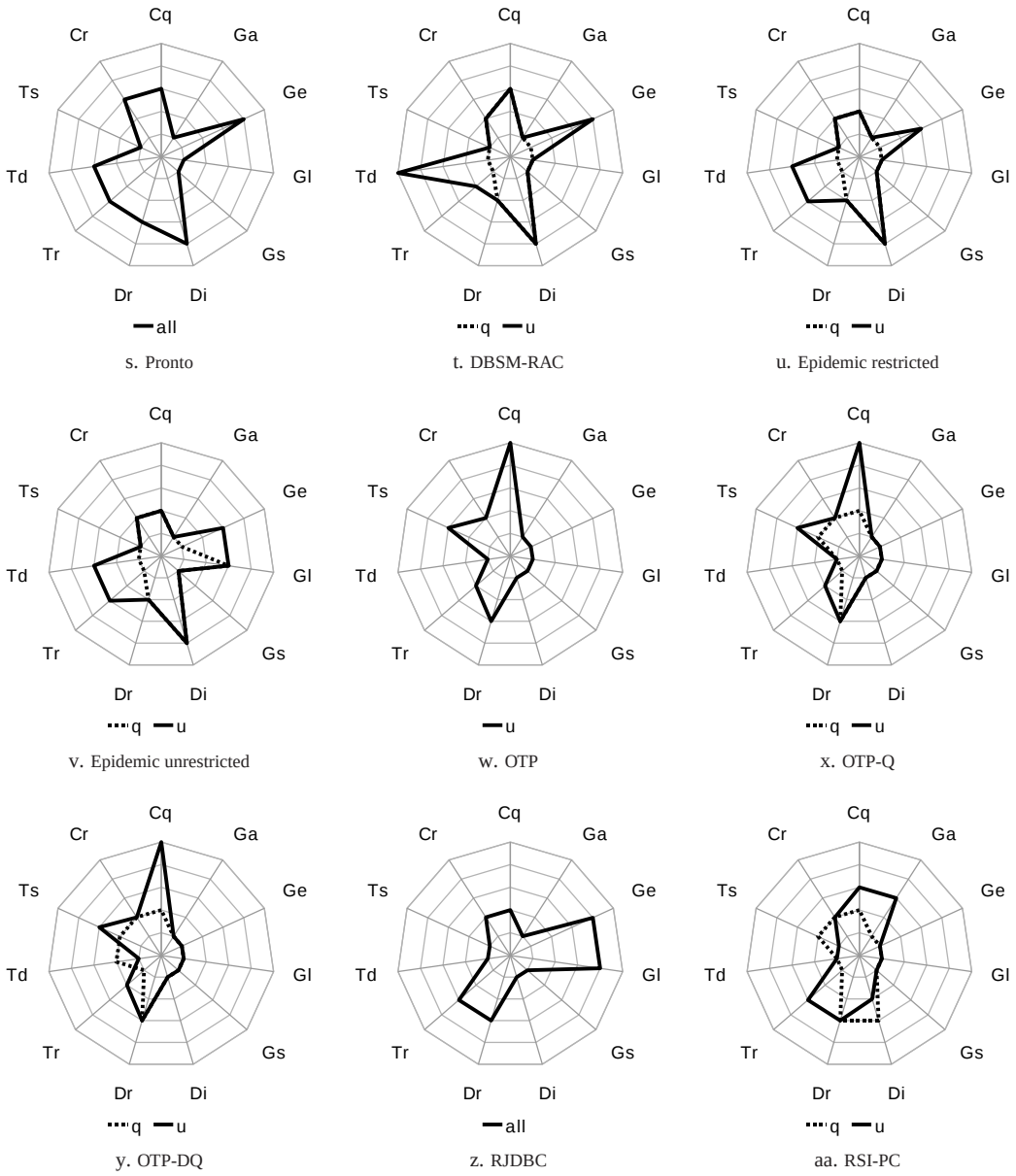
j. Lazy Txn Reord. k. OTP-99/OTP-SQ l. Fast Refresh Df-Im

m. Fast Refresh Im-Im n. DBSM o. SER/CS

p. SI q. Hybrid r. NODO/REORDERING

Figure 3: Visual radial representations of the surveyed systems (cont.)

in OPT, transactions are allowed to proceed freely, storing any updates in private workspaces. For each read operation, the transaction must remember the write timestamp of the read item. Before starting the two-phase commit [*b*], a unique timestamp is assigned to the transaction. A certification is then performed for each transaction in each cohort. If there is some replication, remote updaters (which store copies of the written objects) receive the writeset in the *prepare* message of 2PC and take also part in the certification process. A read request is certified if the version that was read is still the current version and no write with a newer timestamp has been already certified. A write request is certified if no later reads have been locally certified or committed. The term *later* refers to the timestamp assigned at the start of the 2PC. A transaction is certified globally if local certification succeeds for all its cohorts and all its remote updaters [*c*]. This certification process is run inside the local DBMS, which allows a concurrent execution of writesets in remote updaters while ensuring a right concurrency control [*d*]. The optimism of this algorithm, which



s. Pronto t. DBSM-RAC u. Epidemic restricted

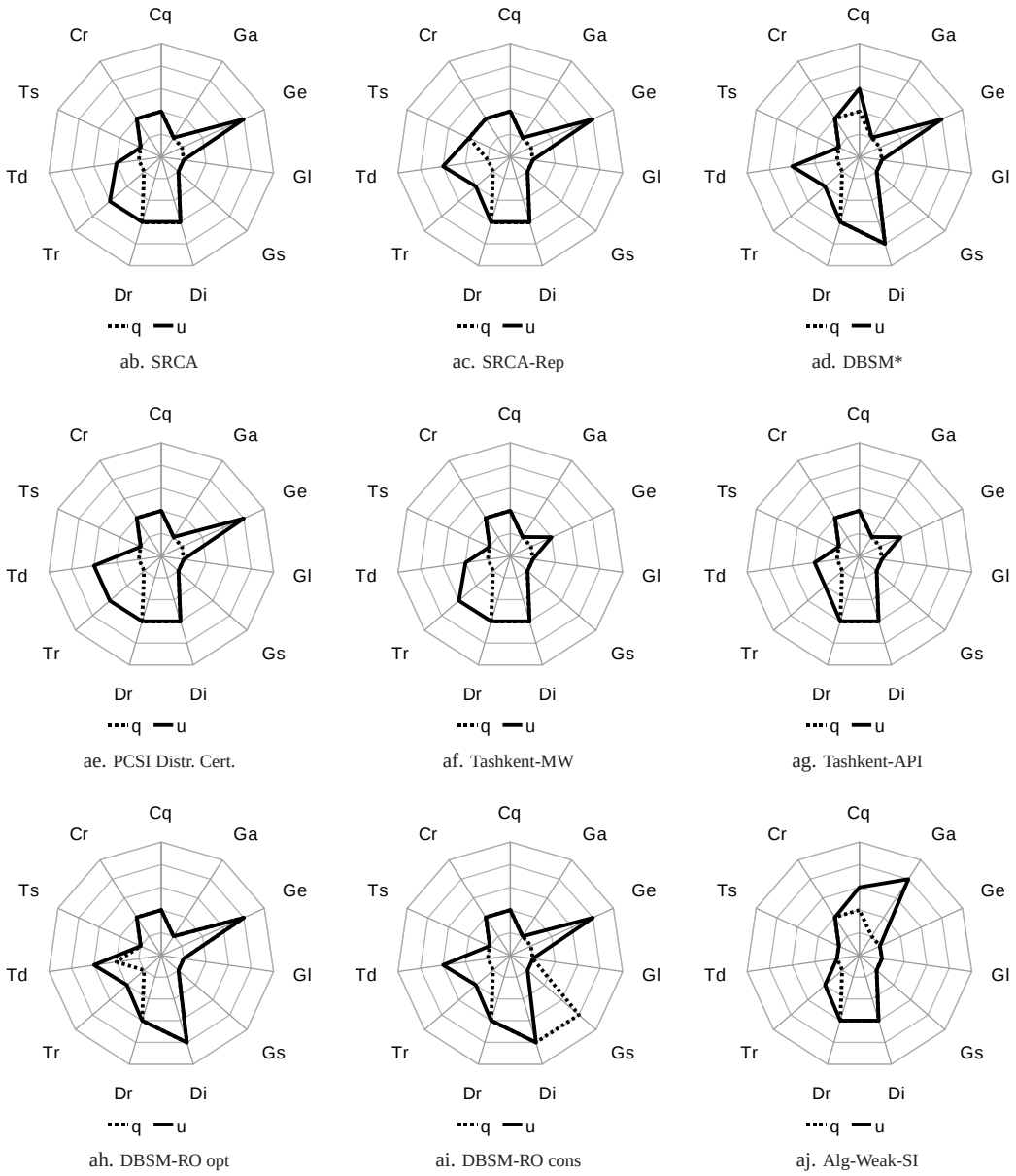
v. Epidemic unrestricted w. OTP x. OTP-Q

y. OTP-DQ z. RJDBC aa. RSI-PC

Figure 3: Visual radial representations of the surveyed systems (cont.)

lets read operations proceed without getting any locks, allows the appearance of potential inversions during the execution of transactions. These potential inversions are later detected and aborted during certification, based on the total order of the timestamps assigned to the transactions, thus providing 1ASR.

O2PL & 2PC (Distributed Optimistic Two-Phase Locking with Two-Phase Commit) was proposed by Carey and Livny [12] as an optimistic version of distributed 2PL. Both algorithms are identical in the absence of replication. However, O2PL handles replicated data as OPT does: when a cohort updates a replicated data item, a write lock is requested on the local copy of the item, but the request of write locks in remote copies is delayed until commit time. During 2PC [*a*], remote nodes must acquire the write locks required by the transaction (this info is in the *prepare* message of the 2PC) before answering to the coordinator. As read operations are required to first get a read lock and certified transactions get all write



ab. SRCA ac. SRCA-Rep ad. DBSM*

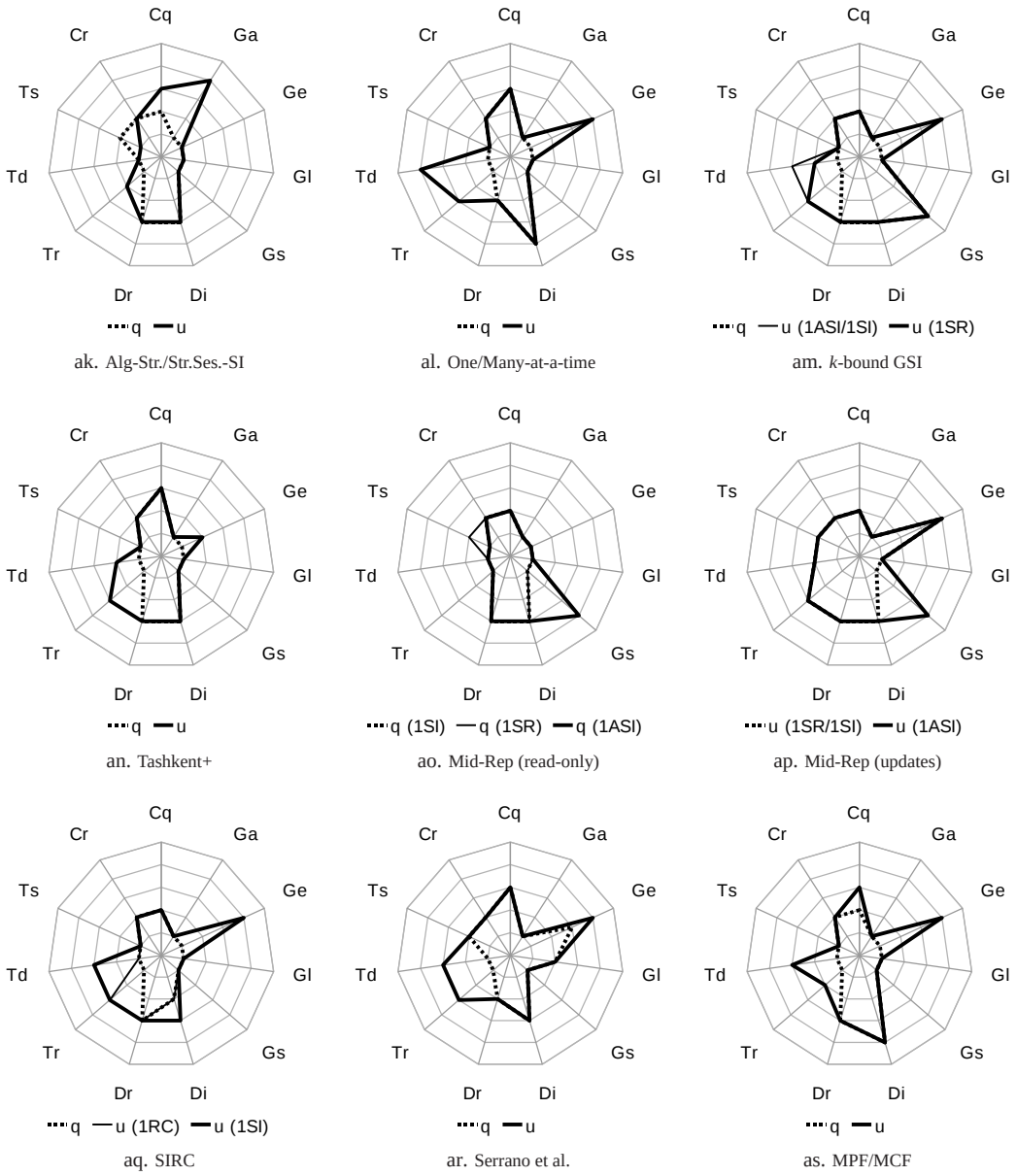
ae. PCSI Distr. Cert. af. Tashkent-MW ag. Tashkent-API

ah. DBSM-RO opt ai. DBSM-RO cons aj. Alg-Weak-SI

Figure 3: Visual radial representations of the surveyed systems (cont.)

locks at all replicas before committing, the correctness criterion of 1ASR is ensured.

Agrawal et al. [1] propose the use of atomic broadcast to simplify the design of replication protocols, thus eliminating the need for acknowledgments, global synchronization or two-phase commitment protocols. Four protocols are proposed. **Broadcast all** is a naive solution that follows the state machine approach [59] broadcasting each operation, read or write, in total order to all replicas and waiting until its delivery to execute it. A final commit operation is also broadcast and applied in the nodes [a]. Thus, every site delivers all operations in the same order: independent transactions may commit at different orders while conflicting operations are ensured to be executed at their delivery order. As a result, a 1ASR correctness criterion is enforced. **Broadcast writes**, optimizes previous algorithm by sparing the broadcast of read operations [a]. A delegate node sends a commit or abort message for its transaction T, as it is the only node where



ak. Alg-Str./Str.Ses.-SI al. One/Many-at-a-time am. *k*-bound GSI

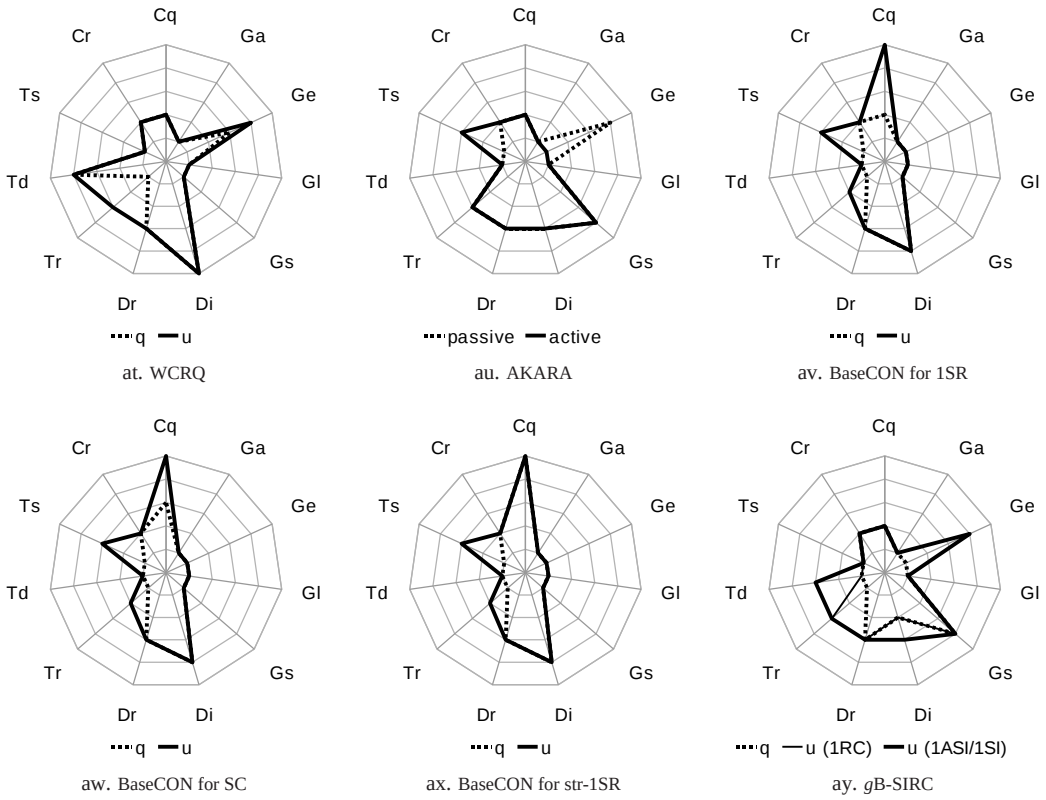
an. Tashkent+ ao. Mid-Rep (read-only) ap. Mid-Rep (updates)

aq. SIRC ar. Serrano et al. as. MPF/MCF

Figure 3: Visual radial representations of the surveyed systems (cont.)

the read operations of T were performed. Read-only transactions are committed only in their delegate [*b*], while update transactions are committed at every replica. 1ASR is guaranteed. **Delayed broadcast writes**, packs all write operations in a single broadcast at the end of transaction T. When this message is delivered, nodes request write locks and execute writes. When T commits in its delegate node, a commit operation is broadcast to all sites. Assuming that read-only transactions are also broadcast at the end and a validation is performed for them, 1ASR is guaranteed. Finally, **single broadcast transactions**, reduces all communication down to a single broadcast at the end of the transaction, only for update transactions (the decision to commit a read-only transaction is done locally [*a*] and, as a result, inversions may occur). Readset and writeset information is contained in this message, for each node to be able to independently certify transactions and grant all write locks. The resulting correctness criterion is 1SR.

These four protocols are referenced to as protocols A1, A2, A3 and A4 and their performance is evalu-



at. WCRQ au. AKARA av. BaseCON for 1SR

aw. BaseCON for SC ax. BaseCON for str-1SR ay. *g*B-SIRC

Figure 3: Visual radial representations of the surveyed systems (cont.)

ated by Holliday et al. [29], who claim that the usage of atomic broadcast in database replication protocols simplifies message passing and conflict resolution, thus making replication efficient, even when providing full replication and update-everywhere capability.

The **Lazy Transaction Reordering** protocol was proposed by Pedone et al. [50] as a replication protocol able to reduce the abort rate of existing lazy approaches[11] by reordering transactions during certification when possible. Traditional Kung-Robinson's certification, where a delivered transaction T is aborted if its readset intersects with the union of the writesets of concurrent and previously delivered committed transactions, is thus changed for a new one where serial order does not necessarily have to match up with that of the atomic broadcast used to send the certification message. The reordering protocol tries to find a position in the serial order where T can be inserted without violating serializability. If no position can be found, T is aborted. The reordering nature of the protocol may increase inversions. As local write operations are tentative and are only confirmed in a non-overlapping manner after certification, a serial execution is achieved despite not using local serializable isolation. As a result, the correctness criterion is 1SR.

OTP algorithm, Optimistic Transaction Processing, was proposed by Kemme et al. [36] and later refined [37]. In order to distinguish between both versions, we denote the initial one with the year of publication **OTP-99**. In OTP-99, all accesses to the database are assumed to be done through the use of stored proce-

[11] In such a paper [50], an approach is identified as lazy if it locally executes transactions and sends certification information to the rest of the system at commit time, as opposed to eager approaches, which are identified in the paper as those that synchronize each data access by communicating with other nodes during transaction execution. Note that these definitions do not match those of Section 2, where eager approaches broadcast the writeset and apply it at all nodes before replying to the client, as opposed to lazy approaches that send and apply writesets at remote nodes after committing in the delegate.

dures. Each stored procedure accesses only one of a set of disjoint conflict classes into which the database is divided. Each node maintains a queue per conflict class, in order to serialize conflicting transactions at middleware level, and has a mechanism that maintains different versions of the data of each conflict class [*e*]. Read-only transactions can be executed at any replica [*a*], using the corresponding snapshot [*c*] and committing locally without further processing [*f*]. To provide consistent snapshots for queries, the different maintained versions of the data are labeled with the index (the position inside the definitive total order) of the transaction that created the version. If T_i was the last processed TO-delivered transaction at the time a query Q starts, then the index for Q is $i.5$ (a *decimal* index). When Q wants to access some data, it is provided with the data corresponding to the maximum version which is lower than the index of the query.

Update transactions are broadcast in total order to all the sites [*b*], where they will be executed in an active way [*d*]. An optimistic atomic broadcast is used, so first transactions are opt-delivered in a tentative order to be later TO-delivered in the definitive total order. When a transaction is opt-delivered, it is inserted in the corresponding queue. All transactions at the heads of the queues can be executed concurrently [*g*], as they do not conflict. When a transaction T is finally TO-delivered, any conflicting transaction T' tentatively ordered before T and not yet TO-delivered must be reordered (as the definitive total order is used as the serialization order for conflicting transactions). If T' already started execution, it must be aborted and later re-executed. T is then rescheduled before all non-TO-delivered transactions in its corresponding queue. On the other hand, when the definitive order matches the tentative one, T can be committed as soon as it is fully executed. After commitment, T is removed from its queue and the following transaction can be submitted to execution.

Regarding the correctness criterion, if we consider each conflict class, i.e., each division of the database, as a logically different database, transactions running in any of those divisions are guaranteed a 1SR correctness criterion, as the snapshots used by queries allow inversions.

Pacitti et al. [45] propose **Fast Refresh Deferred-Immediate** and **Fast Refresh Immediate-Immediate**, two refreshment algorithms for a lazy master replication system. In those systems, each data item has a primary copy stored in a master node, and updates to that data item are only allowed in that node [*b*], while read operations are allowed in any replica [*a*] (inversions may arise). A transaction can commit after updating one copy. The rest of replicas are updated in separate refresh transactions. Partial replication is used in this work, but all transactions are assumed to access only local data (there are no distributed transactions). Secondary copies are stored in other servers. Write operations are propagated to the secondary copies, which are updated in separate refresh transactions. Two types of propagation are considered: deferred (all the update operations of a transaction T are multicast within a single message after the commitment of T) [*h*], and immediate (each write operation is immediately multicast inside an asynchronous message, without waiting for the commitment of T) [*j, l*]. Read operations are not propagated [*e, g, i, k*]. Refresh transactions can be triggered in the secondaries in three different ways: deferred, immediate and wait. The combination of a propagation parameter and a triggering mode defines a specific update propagation strategy. In the deferred-immediate algorithm, a refresh transaction is started as soon as the corresponding message is received by the node; while in the immediate-immediate one, a refresh transaction is started as soon as the first message corresponding to the first operation is received, thus achieving higher replica freshness. Finally, the ordering parameter defines the commit order of refresh transactions. Depending on the system topology, this ordering must be refined in order to maintain replica consistency (secondary copies are updated and no inconsistent state is observable in the meantime). A FIFO reliable multicast with a known upper bound is used. This upper bound, timestamps and drift-bounded clocks in nodes allow the protocol to achieve a total ordering among messages, when necessary, letting refresh transactions to run concurrently [*f*] but forcing them to wait before their final commit operation, in order to perfectly correspond with the commitment order in the master nodes. A serializable isolation level in local databases [*c*] allows the system to ensure 1SR. No decision phase is required, as local concurrency control in master nodes is enough [*d*].

DBSM (Database State Machine Replication) [47] applies the state machine model to the termination protocol of a database replication system. Read-only transactions are directly committed when the user requests to, i.e., no communication is established with other nodes and no decision process is performed

[*a, c*]. Update transactions atomically broadcast [*b*] their information (readset, writeset, updates) to the rest of nodes, where a certification [*d*] is performed, checking for write-read conflicts. Successfully certified transactions request all their write locks and, once they are granted, the updates are performed [*e*]. To make sure that each database site will reach the same state, write-conflicting transactions must request their locks (and be applied to the database) in the same order they were delivered. On the other hand, transactions that do not conflict are commutable: they do not need to be applied to the database in the same order they were delivered. Although different sites may follow different sequences of states (depending on such commutable transactions), write locks –held from certification to final commitment–, prevent users from perceiving such inconsistency. On the other hand, as read-only transactions are not certified (they are locally and immediately committed upon request), inversions are possible. The ensured correctness criterion is 1SR.

During remote writeset application, conflicting local transactions are aborted. Establishing a trade-off between consistency and performance, Correia et al. [15] relax the consistency criteria of the DBSM with Epsilon Serializability. Read-only transactions can define a limit in the inconsistency they import, i.e., the aggregated amount of staleness of read data. Update transactions define a limit in the inconsistency they export to concurrent transactions. This way, during lock acquisition before writeset application, a local mechanism verifies if the inconsistencies introduced by the committing writeset do not force a local query to exceed its limits, or the remote update to exceed its limits. If both limits are not exceeded, the local query can continue and the remote writeset can be applied. Otherwise, the local query is aborted.

SER [35] is a protocol for serializable databases, where long local read locks are requested for read operations, while write operations are delayed until the end of the reading phase [*a*]. When a read-only transaction finishes its operations and requests commitment, it is immediately committed, without any communication with the rest of the system [*b, d*]. On the other hand, update transactions are broadcast in total order [*c*]. When delivered, a validation mechanism is performed [*e*] but not for deciding about this transaction but about local ones not yet delivered. This validation is based on locks. All write locks for the delivered writeset are atomically requested. If there is no lock on the object, the lock is granted. If there is a write lock or all the read locks are from transactions already delivered, then the request is enqueued. If there is a read lock from a transaction T_j not yet delivered, T_j is aborted and the write lock is granted. If T_j was already broadcast, an abort message is sent. After this lock phase, if the delivered transaction was local, a commit message is sent. Thus, a second message, only reliable, is sent for every broadcast transaction with its final outcome, following a weak voting approach (the commit vote for a transaction T_c is sent after the delivery of the writeset of T_c; the abort vote for a transaction T_a is sent before the delivery of the writeset of T_a). Whenever a write lock is granted, the corresponding operation is performed. The queues of requested locks ensure that conflicting operations are serially performed [*f*]. Updates of non-conflicting transactions can be committed at different orders, but the write locks prevent users from perceiving the lack of sequentiality. Nevertheless, inversions may arise. The correctness criterion is 1SR. A PostgreSQL implementation of the SER protocol, Postgres-R, was published and further discussed by Kemme and Alonso [34].

CS [35] is a version of SER for databases where read locks are released after the read operation if the transaction will not update the object later on [*a*]. The algorithm is identical to that of SER, except for the decision phase, where transactions holding short read locks are not aborted when the delivered transaction requests a write lock in the same object. Instead, the delivered transaction waits for the short locks to be released. This way, read operations are less affected by writes, but inversions may increase and the resulting execution may be non-serializable. The update application mechanism, as in SER, allows independent transactions to commit in a different order to that of their delivery, although this is concealed to users by holding write locks from decision time. Therefore, the ensured correctness criterion is 1CS.

SI [35] is deployed upon a local database providing snapshot isolation. On request, read-only transactions are immediately committed without any communication with the rest of the system [*a, c*], while update transactions must be atomically broadcast [*b*]. The sequence number of their writesets is used as end-of-transaction (EOT) timestamp. The begin-of-transaction (BOT) timestamp of T_j is set to the highest

sequence number EOT_i such that T_i and all transactions with lower sequence numbers have terminated in the replica at the starting time of T_j. Certification [d] is made by checking the EOT timestamp of the last transaction currently holding or waiting to acquire a write lock on an object X with the BOT timestamp of the delivered transaction wanting to write X. If both transactions are concurrent (EOT $>$ BOT), the delivered transaction is aborted. If there is no write lock on object X, the comparison is made with the EOT of the transaction that wrote its current version. Non-aborted transactions request their write locks in delivery order. As soon as a lock is granted, the corresponding operation is performed [e]. Again, write locks prevent users from perceiving the lack of sequentiality when non-conflicting transactions are applied at different orders in different replicas. The correctness criterion is 1SI, as inversions are not precluded.

Hybrid [35] is a combination of previous protocols SER and SI. Read-only transactions are executed in snapshot isolation mode [a]: they get a version of the database at their delegate corresponding to their start time (inversions may arise). Update transactions are executed like in SER [b]. Read-only transactions commit immediately without any communication with the rest of the nodes [c, e], whereas update transactions are broadcast in total order [d]. A validation phase based on locks is performed whenever a transaction is delivered [f]. Only the delegate of the transaction is able to decide the final outcome, which is reliably broadcast at the end of the lock phase. As in SER, only local transactions not yet delivered can be aborted. Thus, the decision is not for the delivered transaction, but for transactions that would be subsequent in the total order. Whenever the delivered transaction gets a write lock, the corresponding operation is performed [g], which can produce the reordering of independent transactions. The guaranteed correctness criterion is 1SR.

NODO (NOn-Disjoint conflict classes and Optimistic multicast) was proposed by Patiño-Martínez et al. [46] as a middleware-based replication protocol that aims to enhance scalability of existing systems reducing the communication overhead. Data is partitioned into disjoint basic conflict classes, which are then grouped into distinct compound conflict classes. Each compound conflict class has a master or primary site, which allows the protocol to rely on the local concurrency control for deciding the outcome of transactions [f]. A transaction accesses any compound conflict class, which is known in advance. Read-only transactions can be executed in any node [a], as a complete copy of the database is stored at each node. Update transactions, however, are broadcast in total order to all sites [b]. An optimistic delivery allows the overlap of the time needed to determine the total order with the time needed to execute the transaction. For concurrency purposes, each site has a queue associated to each basic conflict class. When a transaction T is optimistically delivered (opt-delivered), all sites queue it in all the basic conflict classes it accesses. At the master site of T, whenever T is the first transaction in any of its queues, the corresponding operation is executed [c]. When T is delivered in total order (TO-delivered), if the tentative order was correct, T can commit as soon as it finishes execution. Then, its writeset is reliably broadcast in a commit message to all sites [e], where updates are applied after T is TO-delivered and as soon as it reaches the head of each corresponding queue [h]. When all updates are applied, T commits. Committed transactions are removed from the queues. If messages get out of order, any conflicting transaction T' opt-delivered before T and not yet TO-delivered is incorrectly ordered before T in all the queues they have in common. T must be reordered before the transactions that are opt-delivered but not yet TO-delivered. If T' already started its execution, it must be aborted at the master site. Read-only transactions are queued at their delegate node after transactions that have been TO-delivered and before transactions that have not yet been TO-delivered [c]. Once a read-only transaction ends, it is locally committed with no further communication [d, g]. A performance evaluation of an implementation of this protocol was conducted by Jiménez-Peris et al. [32].

A drawback of NODO is that a mismatch between the tentative and the definitive orders may lead to an abortion. Taking advantage of the master copy nature of this protocol, a new version was also proposed in that paper [46]: the **REORDERING** algorithm, where a local site can unilaterally decide to change the serialization order of two local transactions following the tentative order instead of the definitive one in order to avoid such aborts. Remote nodes must be informed about the new execution order (this information is added to the commit message). Restrictions apply, as reordering is only possible if the conflict class of the reordered transaction, the first one in the local tentative order, is a subset of the conflict class of the

so-called serializer transaction, the one that comes first in the definitive total order. The commit message of REORDERING contains the identifier of the serializer transaction and follows a FIFO [*i*] order (several transactions can be reordered with respect to the same serializer transaction, and their commit order in all sites must be the same). When a transaction T_i is TO-delivered at its master site, any non-TO-delivered local transaction T_j whose conflict class is a subset of that of T_i is now committable (it will be committed when it finishes execution, as if it were TO-delivered in the NODO algorithm). Local non-TO-delivered conflicting transactions that cannot be reordered and have started execution must be aborted. At remote sites, reordered transactions are only committed when its serializer transaction is TO-delivered at that site. Both NODO and REORDERING allow inversions and ensure the correctness criterion of 1SR.

Pronto [48] follows the primary-backup approach, so transactions must be addressed to the primary [*a*]. Clients do not need to know which node is the primary at any moment, as the first part of their algorithm is devoted to find the current primary by consecutively asking all the replicas. After transaction execution in the primary, Pronto sends to the backups the ordered sequence of all its SQL sentences [*b*]. This allows heterogeneity in the underlying DBMS as long as they follow the same SQL interface. Possible non determinism is said to be solved by introducing ordering information that allows the backups to make the same non-deterministic choices as the primary. As all replicas completely execute each transaction, Pronto assimilates to an active approach. But unlike active replication, backups process transactions after the primary, allowing the primary to make non-deterministic choices and export them to the backups. The certification process [*c*] does not consider the conflicts between transactions. Instead, a simple integer comparison is performed to check if the transaction was executed in the same epoch where it is trying to commit. A change in the epoch, which results in another server being the primary, occurs when any backup suspects the primary to have failed and broadcasts (also in the total order used for broadcasting transactions) a new epoch message. As these suspicions may be false, the primary may be still running and so it aborts all transactions in execution upon the delivery of the new epoch message. However, due to the time it takes for the message to be delivered, it is possible that multiple primaries process transactions at the same time. To prevent possible inconsistencies, delivered transactions are committed in backups only if they were executed in the current epoch (by the current primary). After termination, all replicas (primary and backups) send the transaction results to the client [*d*]. As inversions are precluded by serving all transactions in the primary, the correctness criterion is 1ASR.

DBSM-RAC, Database State Machine with Resilient Atomic Commit and Fast Atomic Broadcast, was proposed by Sousa et al. [63] as an adaptation of DBSM for partial replication. In partial replication, nodes maintain only the transaction information that refers to data items replicated in that node. Due to this, certification is no longer ensured to reach the same decision at all nodes. Instead, a non-blocking (to tolerate failures) atomic commit protocol must be run, in order to reach a consensus on transaction termination [*e*]. But atomic commit protocols can abort transactions as soon as a participant is suspected to have failed. This goes against the motivation for replication, as the more replicas an item has, the higher the probability of a suspicion, and the lower the probability of a transaction accessing that item to be finally committed. Resilient atomic commit solves this problem by allowing participants to commit a transaction even if some of the other servers are suspected to have failed, for which it requires a failure detector oracle. The second abstraction presented is Fast Atomic Broadcast, a total order broadcast which can tentatively deliver (FST-deliver) multiple times a message before deciding on the final total order (FNL-deliver). This optimistic behavior allows the overlap of the time needed to decide the total order with the time needed to run the resilient atomic commit, thus overcoming the penalty of the latter. A transaction T must start in a node that replicates all the items accessed by T [*a*]. Read-only transactions are locally committed [*b, d, f*]. Update transactions spread their information using the fast atomic broadcast [*c*]. As soon as T is first FST-delivered, all participating sites (those replicating any item accessed by T) certify T and send the certification result as their vote for the resilient atomic commit protocol. When T is FNL-delivered, if the tentative order was correct, the result of the resilient atomic commit is used to decide the final outcome of T. If T can commit, its write locks are requested and its operations executed as soon as they are granted [*g*]. Whenever the orders mismatch, the certification and resilient atomic commit started for T are discarded and the process is repeated for the final order. As in DBSM, write locks prevent users from perceiving the

lack of sequentiality caused by independent transactions committing in different orders in different nodes. As inversions are not precluded, the correctness criterion is 1SR.

Holliday et al. [30] propose a pair of partial database replication protocols supporting multi-operation transactional semantics and aimed to environments where servers are connected by an unreliable network, subject to congestion and dynamic topology changes, where messages can arrive in any order, take an unbounded amount of time to arrive, or be completely lost (however, messages will not arrive corrupted). Each site maintains an event log of transaction operations, where the potential causality among events is preserved by vector clocks. Records of this log are exchanged with the rest of servers in an epidemic way with periodic point-to-point messages. This exchange ensures that eventually all sites incorporate all the operations of the system. A node N_i also maintains a table $\mathscr{T}_i$ that contains the most recent knowledge of N_i of the vector clocks at all sites. This time-table, also included in the epidemic messages, ensures the time-table property: if $\mathscr{T}_i[k, j] = v$ then N_i knows that N_k has received the records of all events at N_j up to time v (which is the value of the local clock of N_j).

Transactions are executed locally. In the restricted access approach, **Epidemic restricted**, a transaction T can access only those data items that are permanently stored in the delegate node of the transaction. When T finishes, if it is read-only it is immediately committed without further processing [a, c, e]. Otherwise, its readset, writeset (with the updated values) and timestamp are stored in a pre-commit record in the delegate node to be epidemically spread [b]. The timestamp used is the i^{th} row of the time-table of N_i, $\mathscr{T}_i[i, *]$, with the i^{th} component incremented by one (the clock value at each node is incremented every time a new record is inserted into the log). This timestamp allows the protocol to determine concurrency between transactions in order to certify them. When N_i knows, by the clock information from epidemic messages, that this record has reached all sites, N_i must have received any concurrent transactions initiated in other nodes and thus has all the required information to certify T [d]. As there is no order guarantee, when a conflict is found between two concurrent transactions, both transactions must be aborted. Not aborted transactions are applied and committed at each node [f].

In the remote access approach, **Epidemic unrestricted**, remote objects can be read and written by maintaining a local temporary database in memory. When a local transaction wants to read a remote data item, the temporary database and pre-commit records from other sites are inspected trying to get a valid version of the item. If no valid version is available, a request record is added to the event log and epidemically transmitted [g]. A site replicating that item would be able to turn that record into a response one, storing it at its log and transmitting it later. On the other hand, when a local transaction wants to write a remote data item, the current value of the item is not required and the transaction can perform the write operation and continue.

Both the restricted and the unrestricted versions of this algorithm allow inversions of read-only transactions. The updates to the local database are applied following the causal order of the log. As a result, 1SR is guaranteed.

OTP was proposed by Kemme et al. [37] (along with OTP-Q, OTP-DQ and OTP-SQ) to achieve high performance by overlapping communication and transaction processing in database replication systems providing full replication and one-copy serializability. OTP is a more refined version of OTP-99 [36], where transactions were restricted to access only one conflict class. OTP only considers update transactions, issued by clients that invoke stored procedures. Whenever a client sends a request to a node, this node forwards it to all sites in an atomic broadcast with optimistic delivery [a]. This primitive allows the overlap of the time needed to determine the total order with the processing of the message. To this end, a message is optimistically delivered (opt-delivered) in an initial tentative order. When the order is agreed, the message is delivered in total order (TO-delivered). Tentative and total orders may differ. The processing of transactions is then done in an active way: all sites execute all operations, i.e., there is no delegate node [b]. When the request is opt-delivered, all required locks are requested in an atomic step. This consists in queueing a read or write lock entry in the queue corresponding to the accessed data item. These queues are maintained by the protocol, so concurrency control is done at middleware level [c], deferring the execution of operations until the corresponding lock is granted [d]. This way, transactions are executed optimistically, but the

commit operation is not performed until the total order is decided. If a transaction T is already executed when it is TO-delivered, or is already TO-delivered when it finishes execution, this means that the tentative order was correct. T commits and releases all its locks. On the other hand, if a transaction T is TO-delivered before it finishes execution, all its lock entries are inspected. Any transaction T' not yet TO-delivered with a conflicting granted lock is aborted: all its operations are undone, all its locks are released and its execution will be restarted later, as the tentative order was not correct and T must be executed before. Finally, all the locks of T are scheduled before the first lock corresponding to a transaction not yet TO-delivered. Independent transactions may commit at different orders. Moreover, as OTP does not consider read-only transactions and update transactions are strictly serialized, inversions are avoided and 1ASR is guaranteed.

OTP-Q, **OTP-DQ** and **OTP-SQ** complement OTP with the management of read-only transactions. In all these protocols requests must be declared in advance as queries or update transactions. Queries are only locally executed with no communication overhead [*e, g*]. A basic approach is taken in OTP-Q, a query Q is treated as if it were an update transaction being TO-delivered: any transaction not yet TO-delivered with conflicting granted locks is aborted and the locks of Q are inserted before the first lock entry corresponding to a not yet TO-delivered transaction. Operations are deferred until the corresponding lock is granted [*f*].

Although simple, OTP-Q requires that queries know in advance all the data items they want to access, which might not be feasible due to their usual ad hoc character. Moreover, queries may access many items and run for a long time. Locking all data at the beginning will thus lead to considerable delay for update transactions. In order to overcome these disadvantages, authors propose OTP-DQ, which treats queries dynamically, allowing queries to request their locks whenever they want to access a new item. To avoid violations of the one-copy serializability, data items are labeled with version numbers corresponding to the position inside the total order of the last transaction that updated them. Each update transaction is also identified with such a version number. Queries maintain two timestamps corresponding to the version numbers of a pair of transactions between which the query can be safely serialized. Each time an update transaction requests a lock on an item read by a query, or whenever the query reads an item, timestamps are adjusted in order to ensure that the query does not reverse the serial order established by the total order. In case that it is detected that the order has been reversed, the query is aborted [*h*].

Both OTP-Q and OTP-DQ place read-only transactions properly inside the serial order but, as queries are not enforced to respect real-time precedence (their processing is local and the validation rules merely aim for serializability), the correctness criterion is 1SR.

Finally, OTP-SQ uses multiversioning for providing each read-only transaction with appropriate versions of all data items it wants to access, i.e., with a snapshot. This way, queries do not acquire locks, do not interfere with updates and can be started immediately [*i*]. The correctness criterion is the same of OTP-Q and OTP-DQ.

RJDBC [23] is a simple and easy to install middleware that requires no modification in the client applications nor in the database internals. A client request arrives to a system node [*a*], which, for each operation of the transaction, and depending on the underlying database concurrency control in use [*b*], decides to broadcast the operation in total order to all replicas [*c*] or not (e.g., read operations in a multi-version concurrency control providing snapshot isolation are not required to be broadcast). If not broadcast, the operation is executed locally. Otherwise, it is sequentially executed upon delivery (the same applies for the final commit operation [*d, f*]). As all nodes execute all significant operations in the same order, no decision phase is necessary [*e*]. The guaranteed correctness criterion depends on the underlying concurrency control and on the decision to broadcast operations [*g*]. If serializability is used for local isolation but read operations are not broadcast, then 1SR is provided. On the other hand, broadcasting also read operations allows the system to achieve 1ASR. Similarly, if snapshot isolation is provided, then 1SI can be achieved without broadcasting read operations, while 1ASI requires such a broadcast.

RSI-PC (Replicated Snapshot Isolation with Primary Copy) was proposed by Plattner and Alonso [51] as a scheduling algorithm for their middleware-based replication platform, Ganymed, where there is a master node and n slave nodes. RSI-PC takes advantage of the non-blocking nature of read operations in snapshot isolation (read operations are never blocked by write operations nor cause write operations to

wait for readers) by treating read-only and update transactions in different ways, thus providing scalability without reducing consistency. All client requests are addressed to the scheduler, which forwards update transactions to the master node and performs load balancing with read-only transactions among the slaves [*a*]. Updates are started in the master without any delay [*c*] and handled under snapshot (actually, under the serializable mode of the underlying Oracle or PostgreSQL databases, which is a variant of snapshot isolation where conflict detection is performed progressively by using row write locks, ensuring that the transaction sees the same snapshot during its whole lifetime) or read committed isolation [*e*]. No decision phase is necessary [*f*], as the local concurrency control of the master replica is enough. After an update transaction commits in the master, its writeset is sent to the scheduler, which has, for every slave, a FIFO update queue and a thread that applies the contents of that queue to its assigned replica [*j*]. Although this constitutes a lazy behavior (update propagation is done out of the transaction boundaries), this algorithm is equivalent to an eager service as strong consistency can be always guaranteed.

Read-only transactions are processed in the slaves using snapshot isolation [*d*], thus no conflicts appear between writeset application and query processing in the slaves, as readers are never blocked by writers in snapshot isolation. However, to ensure strong consistency for read-only transactions, they are delayed until all pending writesets are applied in the selected slave [*b*], thus providing read-only transactions with the latest global database snapshot. For read-only transactions that cannot tolerate any delay there are two choices: to be executed in the master replica (thus reducing the available capacity for updates), or to specify a staleness threshold. No group communication is established by read-only transactions [*i*].

As transaction-remote and client-response strategies are not detailed in the paper, we assume the most plausible choice [*g, h*]. The ensured correctness criterion depends on the isolation mode of update transactions and the staleness toleration of read-only transactions: if queries do not tolerate staleness, they are provided with inversions-free consistency.

SRCA [42] is a centralized protocol, where all transaction operations must be addressed to the centralized middleware, which redirects the operations to any replica [*a*]. Read-only transactions are locally committed without any global communication [*b, d*]. For update transactions, the group end coordination [*c*] is made *after* the decision [*e*] is taken by the centralized middleware. The sequential application of writesets, combined with snapshot isolation at database level and no mechanisms for inversion preclusion[12] results in the correctness criterion of 1SI.

SRCA-Rep was proposed by Lin et al. [42] as a middleware protocol that guarantees one-copy snapshot isolation in replicated databases. Each replica in the system is locally managed by a DBMS providing snapshot isolation. The database is fully replicated, so transactions can be executed in a delegate replica until the commit operation is requested. Then, read-only transactions are locally committed without any communication [*b, d*], whereas writesets from update transactions are broadcast to the rest of replicas in uniform total order [*c*]. Each replica performs a certification [*e*] for each writeset, following the delivery order. Successfully certified writesets are then enqueued in the *tocommit* queue, to be later applied and committed in the local copy of the database, and in the *ws* list, which contains all the transactions applied in the system.

To reduce the overhead of the certification, it is performed in two steps. Each successfully certified transaction receives a monotonically increasing identifier called *tid*. When a transaction T requests commitment in its delegate node R_d, a local validation is performed: its writeset is compared against those of the transactions in the *tocommit* queue of R_d. If any conflict is found (non-empty intersection of writesets), T is aborted. Otherwise, the *tid* of the last certified transaction in R_d is set as the *cert* value of T. When T is delivered at remote replica R_r, its writeset is compared against those of the *ws* list whose *tid* is greater than the *cert* of T. Any conflict leads to the abortion of T. Otherwise, T receives its *tid* and is enqueued in both the *tocommit* queue and the *ws* list of each of the replicas.

To improve performance, a concurrent writeset application is allowed. When some conditions are

[12] Remember that, in snapshot isolation, inversions are conservatively precluded if the snapshot provided to transactions always corresponds to the latest available snapshot in the entire system. Optimistically, transactions may get an older snapshot but be restarted (getting a new snapshot) when the inversion is detected.

satisfied,[13] several non-conflicting transactions from the *tocommit* queue are sent to the database to be applied and committed. This can alter the commit order, causing *holes* and breaking the sequentiality. Thus, new local transactions must be prevented from starting [*a*] as long as there are holes in the commit order. The correctness criterion is 1SI, as inversions are not precluded.

DBSM* was proposed by Zuikevičiūtė and Pedone [68] as a readsets-free version of DBSM. Local isolation is still managed with 2PL, but the certification test enforces the first-committer-wins rule of snapshot isolation. In order to maintain the original 1SR, a conflict materialization technique is used. The database is logically divided into disjoint sets, and each one is assigned to a different node, which is responsible for processing update transactions that access that set [*b*]. Read-only transactions, on the other hand, are scheduled independently of data items accessed [*a*]. An additional control table containing one dummy row for each logical set allows the materialization of write-read conflicts, in order to be detected in the certification. This way, a transaction that reads data from a remote logical set is incremented with an update to the corresponding row in the control table. As inversions are not precluded, the resulting correctness criterion is 1SR.

PCSI Distributed Certification [20] provides prefix consistent snapshot isolation (PCSI), a form of generalized snapshot isolation (GSI), which is equivalent to 1SI. In this distributed certification protocol, read-only transactions directly commit [*c*] without communicating with the rest of nodes [*a*] and update transactions broadcast their writeset in total order [*b*] and are later certified [*d*] and applied at each replica.

Tashkent-MW and **Tashkent-API** were proposed by Elnikety et al. [21] with the goal of uniting both transaction ordering and durability, whose separation in common database replication systems is claimed by these authors as being a major bottleneck due to the high cost associated to sequential disk writes required to ensure in the database the same commit order decided in the middleware. The replication system proposed is compound of a set of database replicas and a replicated certifier, responsible for validating transactions [*d*] and providing replicas with remote writesets. A snapshot isolated database is used in each replica, where read-only transactions are locally committed (no validation nor communication needed [*a*, *c*, *e*]). When an update transaction T finishes in its delegate, it is sent to the certifier [*b*], which replies with the validation result, the writesets generated in remote replicas and the commit order to be enforced in all nodes. The delegate then applies remote writesets and commits or aborts T, depending on the validation result and respecting the global order imposed by the certifier.

In Tashkent-MW, durability is moved to the middleware and, thus, commit operations are fast in-memory operations, which are done serially to ensure the same global order at each replica. Writesets are also serially sent to the database [*f*], but synchronous writes to disk are disabled. On the contrary, in Tashkent-API, commit ordering is moved to the underlying database management system, which is modified to accept a commit order, so multiple non-conflicting writesets can be sent concurrently to the database [*h*] while ensuring the correct commit order. This way, the database can group the writes to disk for efficient disk IO. In both protocols, the transmission of writesets to remote nodes is completely decoupled from transaction execution, as it is done as part of the reply of the certifier to the requests of other nodes [*g*].

Both in Tashkent-MW and Tashkent-API, the commit order followed by replicas is the same. However, the state of the underlying database replicas is updated by grouping multiple commit operations into one single disk write. As this grouping is not forced to be the same in all replicas, servers will not follow the same exact sequence of states: some of them may omit some intermediate states that were present at other servers.[14] Nevertheless, this does not impair consistency and 1SI (as inversions are not precluded) is guaranteed.

[13] The conditions that must hold to send a writeset T to the database of replica R are: (a) no conflicting writeset is ordered before T in the *tocommit* queue; and (b) either T is local or there are no local transactions waiting to start in R or T does not start a new hole.

[14] Imagine a Tashkent system with an initial state, or version, v_0. There are three nodes in the system and each one starts the execution of a local update transaction: R_1 executes T_1, R_2 executes T_2 and R_3 executes T_3. If all the transactions are independent, they will all positively pass their validation at the certifier. Suppose T_1 finishes the first. The certifier responds with the positive decision

DBSM-RO-opt [44] aims to extend the DBSM replication in order to provide inversions-free consistency [*c*] among the nodes. To do so, an optimistic approach is followed: read-only transactions are locally executed in their delegate replica but are also atomically broadcast when the user requests to commit [*a*], so both read-only and update transactions are checked, looking for write-read conflicts: read-only transactions are inspected only by their delegate replica, while update transactions are certified by each replica in the system [*b*]. This avoids inversions and, as a result, the correctness criterion is 1ASR.

DBSM-RO-cons [44] also aims to extend the DBSM replication in order to provide inversions-free consistency but, in this case, a conservative (pessimistic) approach is followed: read-only transactions are atomically broadcast when they begin [*a*] and are executed only when all update transactions ordered before are committed in the executing replica. This means that not only the group-start communication is synchronous but the query must also wait for all pending writesets to be applied (this extra waiting time could be considered as a late occurrence of a deferred transaction-service, Ts1). Update transactions do not need to be broadcast at start time [*b*]. When finished in its delegate replica, a read-only transaction does not need any further communication [*c*] nor any certification [*e*], but update transactions must broadcast their information in total order [*d*] and undergo the usual conflict checking process [*f*]. The resulting correctness criterion is the same than in DBSM-RO-opt.

Alg-Weak-SI [18], as well as Alg-Strong-Session-SI and Alg-Strong-SI, is used in a system with a primary replica and several secondary nodes, where clients send transactions to any replica. Read-only transactions can be executed in the secondaries (without any further communication with the rest of nodes, [*e*]), but update transactions are forwarded to the primary [*a*]. This protocol follows a lazy propagation of updates, so no communication is established during the lifetime of transactions [*b*]. Instead, local concurrency control in the primary replica is the only responsible for deciding the outcome of update transactions [*c*], whose start, updates and final operation (commit or abort) are registered in a log which is later used to lazily propagate [*f*] these operations in order (a FIFO order is required, which provides a total order broadcast as there is only one sender) to the secondary replicas. The sending process inspects each log entry: a start operation is immediately propagated; update operations are inserted in the update list of the transaction they belong to; a commit entry for transaction T causes the broadcast of this operation along with the update list of T; an abort entry of transaction T is also propagated, discarding in this case the corresponding update list. In the secondaries, delivered messages are buffered in the *update* queue and processed in order. When the start message of T_i is processed –after waiting for the *pending* queue to be completely empty–, a refresh transaction T_i' is started. When the commit message of T_i (with the updates associated) is processed, a new thread is created to apply the updates of T_i using transaction T_i', and the commit operation of T_i is appended to the *pending* queue. This allows the protocol to concurrently apply writesets while ensuring the same commit order of transactions [*d*]. As read-only transactions are executed in secondary replicas without inversion preclusion, inversions may occur (queries may get an old snapshot). Thus, the correctness criterion is 1SI [*g*].

Alg-Strong-SI and **Alg-Strong-Session-SI** [18] guarantee strong snapshot isolation (1ASI) and strong session snapshot isolation (1SI+), respectively [*c*]. While 1ASI avoids all inversions, 1SI+ prevents inversions within the same user session. In order to provide 1SI+, a version number is assigned to each session, corresponding to the version installed by the last update transaction in that session. When a read-only transaction of the same session wants to start, all writesets with version numbers inferior to the session

but it does not have any pending writeset for R_1, so this node commits T_1, reaching (from version v_0) version v_1 (corresponding to the updates of T_1). Now R_2 finishes the execution of T_2 and sends it to the certifier, which responds with the positive decision and with the writeset of T_1. As transactions are independent, R_2 sends them together to the database, which writes their commits in a single disk write, thus moving from version v_0 directly to version v_2 (corresponding to the updates of T_1 and T_2). Finally, R_3 finishes the local execution of T_3 and sends it to the certifier, which responds with the positive decision and with the writesets of T_1 and T_2. R_3 applies the three transactions in a single disk write, thus passing from version v_0 to version v_3 (corresponding to the updates of the three transactions). This way, the three replicas end with the same final state and no other possible transaction has been able to see any inconsistent state, but the sequences of database states differ from replica to replica.

version number must be applied in the secondary replica prior to the start of the read-only transaction [*a*]. If, instead of having one session per client, there is a single session for the system, then 1ASI is provided. Update transactions can start immediately as they are all executed in the same primary replica [*b*]. Apart from this, these protocols are identical to Alg-Weak-SI.

One-at-a-time and **Many-at-a-time** [58] are two termination protocols that extend DBSM to provide a quasi-genuine partial replication, where a node permanently stores not more than the transaction identifier for those transactions that do not access any item replicated in that node. To avoid consequent unnecessary abortions, a non-trivial validation is performed, based on quorums. A transaction T can only be executed on a site that replicates all items accessed by T [*a*]. Read and write operations are executed locally according to the strict two-phase locking rule [*b*]. When a read-only transaction requests commitment, it is locally committed [*c, e, g*]. In the case of an update transaction, the transaction (identifier, delegate site, readset, writeset with updates, and the logical timestamp of the transaction submission) is broadcast [*d*] in a *weak ordering reliable broadcast*, an optimistic primitive that takes advantage of network hardware characteristics to deliver messages in total order with high probability. A consensus procedure is used to decide the total order of delivered transactions. The non-trivial validation consists in a voting phase where each site sends the result of its validation test to the rest of nodes. Each site can then safely decide the outcome of a transaction T when it has received votes from a voting quorum of T [*f*], i.e., a set of sites such that for each data item read by T, there is at least one site that replicates this item. Instead of first using consensus to determine the next transaction T and then executing the voting phase for T, a different approach is taken, overlapping both processes. In the one-at-a-time algorithm, each site votes for its next undecided transaction T and proposes it for consensus. By the time the consensus decides for transaction T, luckily every site will already have received the votes for T. If consensus decides a transaction different from that voted by a site, a vote message is sent for the decided transaction. When a transaction is successfully validated, it is applied in the site [*h*] and the global version counter used to timestamp transactions is increased. This algorithm validates one transaction at a time, which can be a bottleneck if many transactions are submitted. The many-at-a-time algorithm, which does not rely on spontaneous total order, tries to solve this by proposing sequences of transactions and changing the validation test accordingly. As inversions are not precluded, the correctness criterion is 1SR.

***k*-bound GSI** [4] is able to bound the degree of snapshot outdatedness from a relaxed GSI (1SI) to a strong SI (1ASI), while optimistically executing transactions; and it also provides a serializable level for those transactions requiring higher isolation (1SR). As local DBMSs are only required to provide snapshot isolation, serializable transactions are parsed in order to transform SELECT statements into SELECT FOR UPDATE ones. This simplifies the detection of write-read conflicts, which are then governed by the first-committer-wins rule.

Two snapshots taken at the same *real* time in different replicas may be different, as only states at the same *logical* time are guaranteed to be consistent. To allow an optimistic execution, before the first operation of each transaction T, an asynchronous $T.ID$ message is broadcast in total order [*a*], so that the logical starting time of T can be established. Then, the optimistic execution of T overlaps with the time required to complete such initial communication. Moreover, T specifies a value k as the maximum *distance* between the snapshot it took (corresponding to the real time of its start operation) and the snapshot created by the last transaction that committed in any system node before T started (corresponding to the logical time of the start operation of T). This distance is measured as the number of colliding writesets that are applied in the delegate node of T from the real starting time of T until its logical starting time (the delivery of $T.ID$). A colliding writeset is a writeset that has a non-empty intersection with the intended readset of T, which has to be declared in advance. When the number of colliding writesets is greater than k, T is aborted and will be restarted when $T.ID$ is processed. Thus, with $k = 0$, the transaction is executed under 1ASI; with $k > 0$, the achieved correctness criterion is 1SI (authors, to highlight the possibility of defining different staleness levels, refer to the different *bound values* for the GSI criterion, as opposed to the standard GSI, which occurs with an infinite value of k). Overloading the meaning of k, a value of -1 indicates that T requires serializability.

When a read-only transaction finishes its operations, it is locally committed (after receiving its own

T.ID message and as long as it has not been aborted in the meantime, for transactions with $0 \leq k < \infty$) without any communication with the rest of nodes [*b, d*], whereas the writeset of an update transaction is broadcast in total order to the rest of replicas [*c*]. A certification process [*e*] is performed in every replica for each delivered writeset. In order to avoid sending readsets, the decision for serializable transactions is taken in their delegate node [*f*] and then broadcast to the rest of nodes.

Tashkent+ [22] was proposed as an evolution of Tashkent-MW where a memory-aware load balancing is performed in order to further minimize the disk IO requirements. Changes to the previous system include the addition of a scheduler, with different scheduling algorithms, and an optimization, called update filtering, for reducing the update propagation load. Transaction types are predefined and the scheduler is able to estimate the amount of memory, called working set, that each type will need. With this information, authors use a bin packing heuristic to group transaction types so that their combined working sets fit together into the available memory, thus avoiding memory contention and subsequent disk IO. Servers are assigned to transaction groups and this allocation can be dynamic for changing workloads. Different proposed scheduling algorithms differ in the way the working sets are estimated. This way, each transaction is dispatched to a server assigned to its transaction group [*a*].

Update filtering consists in identifying unused tables in a replica (those not accessed by the transaction group to which the replica is assigned) and filtering out the updates to those tables, thus reducing the overhead of update propagation. This optimization is only possible under stable workloads, i.e., when the assignment of replicas to transactions groups is permanent. This way, Tashkent+ is essentially a fully replicated design but may, under some conditions, present the advantages of partial replication [*b*].

Apart from the changes explained above, the rest of the system works as in Tashkent-MW. Authors claim that the correctness criterion is 1SI+ (inversions precluded within sessions), as a given connection can execute only one specific transaction type and will be, thus, always assigned to the same group of replicas. But no details are provided about how all replicas in the same group are atomically updated or how they provide transactions with updated snapshots regardless of the replica where the transaction starts. Thus, the correctness criterion is here considered to be 1SI [*c*].

Mid-Rep is a pessimistic weak voting protocol proposed by Juárez et al. [33] that provides three different correctness criteria on top of a DBMS supporting SI: 1SR, 1ASI and 1SI. Transactions define the criterion they require. For 1SR transactions, all SELECT statements are turned into SELECT FOR UPDATE ones. For 1ASI transactions, a start message is sent in total order [*c*] and the transaction must wait for its delivery to proceed. When a read-only transaction finishes its operations, it is immediately committed at its delegate replica and no further processing is required. On the other hand, update transactions broadcast their writeset in total order to all available replicas, which will apply them sequentially [*d*] and terminate (commit or abort) each transaction according to the voting message sent by the master site (the delegate) of the transaction. During writeset application, no other potentially conflicting local operation is allowed to start: all write operations and also read operations performed by 1SR transactions are thus disabled [*b*] (read operations from 1ASI or 1SI transactions are not deferred [*a*]).

SIRC [56] concurrently supports snapshot and read committed isolation, as long as both levels are provided by the local DBMS [*a*]. Read-only transactions are locally committed without any communication with the rest of replicas [*b, d*], while update transactions are broadcast in total order [*c*]. For SI transactions, a certification based on write-write conflicts is performed [*e*]. RC transactions do not need any decision phase [*d*]. Writeset application follows the delivery order.

Serrano et al. [60] propose a replication protocol aimed to increase scalability of traditional solutions, commonly based on full replication and on a 1SR correctness criterion. These two characteristics are claimed to introduce an important overhead and to limit concurrency. Consequently, their proposal is to use partial replication and a more relaxed correctness criterion, 1SI, where inversions are not precluded and underlying databases are snapshot-isolated. The client connects to a site [*a*] that at least stores the data accessed in the first operation of the transaction T. This node acts as the coordinator, assigning a starting

timestamp to T and redirecting operations [c] to other nodes when necessary. In those other nodes, T must use the same snapshot, the one corresponding to its starting timestamp. To this end, each node starts dummy transactions each time that a transaction commits. When the redirected operation is the first operation of T in the forwarded node, the corresponding dummy transaction is associated to T (later operations will use the same transaction). Prior to execute each redirected operation, the changes previously produced by T are applied at the forwarded site (naturally, only those affecting data stored at that node) [b] (remember that the transaction-service policy applies at each participating node in the case of distributed transactions). After execution, the forwarded site propagates the result of the operation and all the new changes to the coordinator, which applies these changes before executing the next operation. When the client requests commitment of a read-only transaction, the coordinator multicasts a commit message to all participating sites [d] (there is no need for validation [f] nor execution in remote nodes [h]). In the case of an update transaction, the coordinator broadcasts its writeset in total order [e]. All sites perform then a certification [g]. If certification succeeds, all nodes apply the writeset (those nodes that have already performed some operations of T apply only the missing updates) in a non-overlapping way [i] and T can commit.

Zuikevičiūtė and Pedone [70] proposed a scheduling algorithm for the DBSM replication protocol. Aborts can be reduced if conflicting transactions are executed in the same node, thus letting the local concurrency control appropriately serialize them. On the other hand, parallelism improves performance, reducing response times. Considering this trade-off, a hybrid load balancing technique is proposed, which allows database administrators to give more or less significance to minimizing conflicts or maximizing parallelism. Maximizing Parallelism First, **MPF**, prioritizes parallelism and so it initially assigns transactions to nodes trying to keep the load even. If more than one option exists, then it tries to minimize conflicts. Minimizing Conflicts First, **MCF**, avoids assigning conflicting transactions to different nodes. If there are no conflicts, it tries to balance the load among the replicas. A compromise between the two opposite schemes can be achieved by a factor *f*. This way, update transactions are analyzed and a specific replica is chosen to be the delegate [b]. On the other hand, both techniques assign read-only transactions to the least loaded replica [a]. Apart from this novel scheduling, the followed strategies are the same as in DBSM and, thus, the correctness criterion is also the same.

WCRQ [53] is a bridge between consensus-based and quorum-based replication. Underlying databases provide serializability, using long read locks for reading operations and deferring write operations until the end of the reading phase [a]. When an update transaction T finishes in its delegate replica, a uniform total order broadcast [c] is sent to the rest of replicas with the transaction writeset. When it is delivered, each replica tries to get write locks for each item in the writeset of T. If there was one or more read locks on an object, every transaction holding them which is not yet serialized is aborted (by sending an abort message in uniform total order if it was already broadcast), and the write lock is granted to T. If there was a write lock in the object, or if some read locks are from transactions serialized before T, T waits until those locks are released. When a replica gets all the locks of a transaction, it sends a point-to-point acknowledgment message to the delegate. When the delegate gets all the write locks of the transaction and receives acknowledgment from a write-quorum of replicas, it sends a commit message in a uniform reliable broadcast. When this message is delivered, every replica commits the transaction. As these messages are not ordered, independent transactions may commit at different orders in different nodes. When a transaction commits, all other transactions waiting to get write locks in the updated objects are aborted (their delegate sends an abort message in a uniform reliable broadcast). When a read-only transaction finishes in its delegate replica, a message with the readset is sent to a read-quorum of replicas [b]. When this message is delivered, replicas try to get read locks for the items on the readset. When the locks are acquired, if the version is the same as the one read in the delegate, the replica sends back a positive acknowledgment message. Otherwise, a negative acknowledgment message is sent. In any case, read locks are released as soon as this validation is done. When the delegate receives positive acknowledgments from a read-quorum of replicas, it commits the transaction. Otherwise, if any negative acknowledgment is received, the transaction is aborted. For both read-only and update transactions, a quorum of replicas is required to get locks on the items and check that the current versions are equal to the accessed versions in the delegate [d]. As transaction-remote and client-response strategies are not detailed in the paper, we assume the most

plausible choice [*e*, *f*]. Inversions are avoided by ensuring that queries do not read old values. As a result, the 1ASR correctness criterion is guaranteed.

AKARA [16] allows transactions to be executed either in an active or in a passive manner (in both cases, interactivity is precluded). Upon transaction submission, the type (either active or passive) and the conflict classes of the transaction are computed, and an initial total order broadcast is sent with transaction information. After delivery, and once the transaction is the first in the processing queue, the transaction is started [*b*] (this introduces an additional wait to the synchronous message transmission, for both active and passive transactions). For passive transactions, a local execution phase is performed and afterwards the writeset is reliably sent to the rest of replicas [*c*] to be applied following the total order established by the broadcast sent at transaction start. For active transactions, no local phase exists [*a*]: transactions are initiated, executed and committed in all nodes at the same logical time (that of their slot inside the total order). No extra communication is needed for these transactions [*d*]. As isolation corresponds to the snapshot level and no mechanisms avoid inversions, the ensured correctness criteria is 1SI.

Zuikevičiūtė and Pedone [69] characterized different correctness criteria for replicated databases and presented three variants of BaseCON, one for each of the discussed correctness criteria. With **BaseCON for 1SR** transactions are serialized but the causal order may not be preserved. In **BaseCON for SC** (session consistency), transactions are serialized and the real-time order of those belonging to the same user session is also preserved and, thus, clients can always read their own previous updates (this corresponds to the 1SR+ correctness criterion [55]). These two variants are identical except for the way the scheduler selects the executing replica for read-only transactions [*a*]: in BaseCON for 1SR, all replicas are considered and the transaction is forwarded to the least loaded one; in BaseCON for SC, the scheduler considers only those replicas where previous update transactions of the same session have been already applied. Once in the executing replica, read-only transactions start as soon as they are received [*c*] and are committed locally. On the other hand, update transactions are broadcast in total order to every replica in the system [*b*] and executed in active manner, so no local phase exists [*d*]. Strict two-phase locking is used to achieve serializability [*e*]. No decision phase is required [*f*] as all transactions can commit, but update transactions must wait [*g*] for all previously delivered conflicting update transactions to commit in this replica before starting. The commit order of all update transactions is required to be the same as their delivery order. Transaction results are sent from each executing replica to the scheduler, which sends to the client only the first of the replies [*h*].

The third version of the protocol by Zuikevičiūtė and Pedone [69] is **BaseCON for strong 1SR**, which always preserves the real-time (or causal) order of transactions in their serialization. To this end, some changes are applied to previous systems: read-only transactions are directed to the scheduler but also broadcast in total order to all replicas, like update transactions [*a*].[15] The scheduler then determines, for the read-only transaction, the set of replicas where preceding update transactions of any client have already been committed. From this set, the scheduler selects the least loaded server, where the query immediately starts its optimistic execution. When this transaction is delivered in the chosen replica by the total order broadcast, a test is performed to check if the scheduler has changed since this transaction was scheduled. In this case, the transaction is aborted and restarted. This check allows the system to tolerate failures and cannot be considered as a decision, as the transaction always commits. Inversions are precluded by scheduling read-only transactions to updated replicas, thus achieving 1ASR correctness criterion.

***g*B-SIRC** [57] is deployed upon a database offering both read committed and snapshot isolation levels [*b*]. This protocol provides several correctness criteria: one based on the read committed isolation (1RC) and another based on snapshot isolation, with a configurable level of staleness, defined by factor g, from 1ASI (or strong SI) with $g = 0$, to 1SI (or standard GSI) with an infinite value of g. Intermediate values of factor g allow transactions to define the exact amount of outdatedness they can tolerate (authors refer to this

[15] However, unlike update transactions and despite being the client request addressed to all replicas in the system, only the node chosen by the scheduler will execute the transaction, thus serving the client request.

non-standard criterion as *g*-Bound). Similarly to *k*-bound GSI, all SI-based transactions (those providing a value for *g*) broadcast an asynchronous *T.ID* message in total order when they start [*a*], which allows their optimistic execution while establishing a global starting point that would be enforced when transactions abort due to a number of conflicts greater than *g*. Read-only transactions can be locally committed without any global communication [*c, e*]: RC queries commit as soon as they finish their operations, while SI ones must wait to the processing of their *T.ID* message and, if they have not been aborted during the meantime, they can be locally committed. Regarding update transactions, once they finish their operations, their writesets are broadcast in total order [*d*]. For 1RC update transactions, no decision phase is implemented [*e*]. All other update transactions are certified in search for write-write conflicts [*f*]. Whenever a writeset is committed in a replica, local SI-based transactions are validated in search for write-read conflicts with it [*g*], which are tolerated in a number up to *g*. Writesets are applied in a non-overlapping manner. Inversions are precluded only for 1ASI transactions.

4.2 Scope of the Proposed Model

The policy-based characterization model proposed in Section 3 and used for this survey is intended to be general enough to cover all possibilities in replication systems, thus providing a tool able to represent their basic skeleton. The set of strategies followed by a replication system constitutes its operational basis and allows an adequate comparison between systems.

Obviously, many finest-grained details, like optimizations or concurrency control rules, are not covered by this characterization, as intending otherwise would result in an extremely complex model. Thus, there is a trade-off between simplicity and completeness.

Moreover, and despite our efforts, this model is not valid for all replication systems. This is the case of *distributed versioning* [3], a replication protocol tailored to back-end databases of dynamic content web sites, characterized by presenting a low rate of update operations. This protocol aims to achieve scalability while maintaining serializability. The cluster architecture for distributed versioning consists of an application server, a scheduler, a sequencer and a set of database replicas. In order to achieve serializability, a separate version number is assigned to each table. Each transaction issued by the application server is sent to the scheduler, specifying all the tables that are going to be accessed in the whole transaction and whether these accesses are for reading or for writing. The scheduler forwards this information to the sequencer, which atomically assigns table versions to be accessed by the operations in that transaction. This assignment establishes the serial order to be enforced and allows transactions to concurrently execute operations that do not conflict. All transactions can commit (Td0), those conflicting will follow the serial order dictated by the sequencer. New versions become available when a previous transaction commits or as a result of last-use declarations (an optimization for reducing conflict duration). After the assignment for transaction T is completed, the application server can start to submit the operations of T. The conflict-aware scheduler is able to forward a read operation to the least loaded updated replica. Write operations are broadcast to all replicas and actively executed. This scheduling could be interpreted as a Cq2 for reads and Cq4 for writes. However, in this case it is not the whole transaction which is scheduled but each single operation inside the transaction. It could also be represented as Gl3 and Ge3 for write and commit operations, but the communication initiative is not taken by a server executing the affected transaction. Indeed, no communication is ever established among replicas. Instead, each operation sent from the application server is forwarded by the scheduler to the corresponding replica(s), which execute them independently. Thus, communication is done only between the application server and the scheduler, and between the scheduler and the replicas. Once in a replica, an operation must wait for all its version numbers to be available, which could be represented as Ts1 or Tr1-p (although, again, it is not the transaction start which is deferred but the start of each single operation inside the transaction). Concurrency control is thus made at middleware level (the database isolation level is not detailed in the paper). Once a replica executes the operation, it returns its results to the scheduler. The first reply received by the scheduler is sent back to the application server (Cr1). As read operations must wait also for their version numbers to be available before starting, the correctness criterion is 1ASR. In summary, the main problem for describing distributed versioning with our model is that this system divides transactions into their individual operations and, while the management of each of these operations can be represented with our strategies, the handling of the whole transaction

cannot be depicted by our model.

A similar middleware-based system is presented by Cecchet et al. [13] for clustering back-end databases of large web or e-commerce sites. C-JDBC also features a scheduler that sends update operations to all involved servers while performing load balancing for read operations. But in this case, client requests contact a server and each server contains a scheduler component. C-JDBC supports both full and partial replication, while ensuring inversions-free consistency between replicas: at any single moment, only one updating operation (write, commit or abort operation) is in progress in the virtual database, and responses are returned to the client only once all servers have processed the request. The correctness criterion will then depend on the isolation level offered by the underlying databases. As for the previous system, the processing of each single operation can be depicted with our model, but that of the whole transaction would constitute a loop of such a representation of single operations.

4.3 Discussion

Chronologically ordered characterizations of Table 3 summarize the evolution of database replication systems since their appearance. Earlier systems –distributed databases with some degree of replication– were devoted to provide the highest correctness criterion, 1ASR, using to this end the serializable isolation level in local databases and rigid synchronization mechanisms, inherited from standalone database management systems, such as distributed locking for concurrency control (which involves a linear communication with other servers), or an atomic commit protocol like 2PC (which requires several rounds), in order to reach a consensus among participants about transaction termination and thus ensure consistency. Examples of these earlier systems are 2PL & 2PC [24], BTO & 2PC [7], Bernstein-Goodman [9], OPT & 2PC [61] and O2PL & 2PC [12]. However, these mechanisms restricted concurrency, thus severely reducing performance and scalability.

Research efforts focused then on improving these factors trying to reduce communication and replication overhead with new concurrency control algorithms and more efficient termination management, localizing the execution of operations in delegate or master sites, simplifying termination with the use of group communication systems, using optimistic communication primitives, considering different topologies, relaxing isolation and consistency or introducing partial replication schemas. One example of such relaxed isolation, which is still valid for a wide range of applications, is the snapshot isolation level. An interesting feature of snapshot isolation is that read operations are never blocked by write operations, nor cause write operations to wait for readers. SI became popular and many database replication systems started to provide this isolation. Some systems exploiting this level and offering correctness criteria based on snapshot isolation are SI [35], RSI-PC [51], SRCA and SRCA-Rep [42], PCSI Distr. Cert. [20], Tashkent-MW and Tashkent-API [21], Alg-Weak-SI, Alg-Str.-SI and Alg-Str.Ses.-SI [18], *k*-bound GSI [4], Tashkent+ [22], Mid-Rep [33], SIRC [56], Serrano et al. [60], AKARA [16] and *g*B-SIRC [57].

Other proposals aimed at adaptability, designing systems able to provide different consistency guarantees that would fit better the requirements of modern applications, which usually include different types of transactions that require different levels of isolation. This led to a new generation of protocols that support different correctness criteria at the same time (such as RSI-PC [51], *k*-bound GSI [4], Mid-Rep [33], SIRC [56] and *g*B-SIRC [57]), which improved performance by executing each transaction at the minimum required level of isolation.

Considering each policy separately, it is clear that in some cases there is a majority strategy with very few exceptions, while in other policies there is no pronounced trend towards any specific strategy. Some choices may have strong implications in consistency or performance, and this may make systems favor ones against others. Let us analyze each policy in detail.

The most used client-request (Cq) strategy is Cq1: any server can process a request. This policy allows an easy management of requests and load balancing, although it requires a correct global concurrency control in order to avoid inconsistencies. On the other hand, systems that use primaries or master sites may rely on the local concurrency control of such nodes but require client requests to be addressed or forwarded to such servers (Cq2). This is the case of Alsberg-Day [2], Fast Refresh Df-Im and Fast Refresh Im-Im [45], Pronto [48], RSI-PC [51], DBSM* [68], Alg-Weak-SI, Alg-Str.-SI and Alg-Str.Ses.-SI [18], and Tashkent+ [22]. Cq2 is also used by systems that provide partial replication and need to address client requests to a server containing the data required by the operation (DBSM-RAC [63], One-at-a-time and

Many-at-a-time [58], Serrano et al. [60]). Other systems also use Cq2 because they require transactions to be addressed to updated replicas, in order to provide stronger consistency, such as BaseCON for SC [69]. Finally, scheduling algorithms MPF and MCF [70] may also select a specific server in order to minimize abortions by executing conflicting transactions at the same node, thus relying on local concurrency control to appropriately serialize transactions.

Forwarding the client request to all system nodes (Cq4) in a total order broadcast is a possible approach for establishing an early synchronization point. Systems such as NODO and REORDERING [46], and the families of OTP [36, 37] and BaseCON [69] implement such a client-request policy. Systems that require total order guarantees for synchronizing transaction execution must wait for such a communication primitive to agree on the delivery order. NODO and REORDERING move their synchronization point to the start of the transaction and use an optimistic delivery which allows the system to overlap the time needed by the GCS for the agreement with the time needed to execute the transaction. By the time the delivery order is decided, the transaction has already progressed with its operations in its delegate server. On the other hand, both OTP and BaseCON families follow a Cq4 policy in order to execute update transactions in an active manner, while queries are executed at only one server.

Surveyed systems mainly follow, in their transaction-service policies (Ts), the strategy of immediate service (Ts0), under which transactions are started as soon as the server has enough free resources. In some cases, it is necessary to block the processing of transactions until some condition holds (Ts1). This is the case of several systems: NODO, REORDERING [46], OTP-Q and OTP-DQ [37], where concurrency control is done at middleware level and thus transactions must wait for the end of previous conflicting operations in order to be started; RSI-PC [51], Alg-Str.-SI and Alg-Str.Ses.-SI [18], where the service of transactions is deferred to guarantee stronger consistency; SRCA-Rep [42], where local transactions must be prevented from perceiving the lack of sequentiality; Mid-Rep [33], where potentially conflicting local operations are disabled during writeset application; and the algorithm by Serrano et al. [60], where cohorts of distributed transactions must apply the updates of previous operations of the transaction (served by other nodes) before executing the requested operation in their local database.

Systems that actively execute transactions (OTP-99 [36], OTP, OTP-Q, OTP-DQ and OTP-SQ [37], AKARA [16], BaseCON for 1SR, BaseCON for SC and BaseCON for strong 1SR [69]) are said to implement the strategy of no local service (Ts2) for those active transactions.

Regarding group-start (Gs) strategies, only few systems require to make a communication at transaction start, thus establishing a global starting point for transactions. That is the case of DBSM-RO-cons [44], which totally orders queries to provide 1ASR; *k*-bound GSI [4], Mid-Rep [33] and *g*B-SIRC [57], which guarantee 1ASI by totally ordering transaction starts; and AKARA [16], which moves the required synchronization point to transaction start and allows active and passive transaction processing.

With regard to the degree of replication (Dr), earlier systems (2PL & 2PC [24], BTO & 2PC [7], Bernstein-Goodman [9], OPT & 2PC [61], O2PL & 2PC [12]) were mostly distributed databases where replication was not widely used (Dr1). After the generalization of full replication (Dr2), only few systems (Fast Refresh Df-Im and Fast Refresh Im-Im [45], DBSM-RAC [63], Epidemic restricted and Epidemic unrestricted [30], One-at-a-time and Many-at-a-time [58], Serrano et al. [60]) feature partial replication (Dr1), mainly to minimize the cost of update propagation and application, although other mechanisms or constraints must be applied for the correct management of transaction execution.

To alleviate their complexity and allow replication protocols to focus on their native purpose of ensuring replica consistency, systems usually delegate local concurrency control to the DBMS with the appropriate isolation level (Di) for which the protocol has been conceived. Depending on the correctness criterion, systems require local DBMSs to provide different isolation levels. Thus, earlier systems and those requiring a high level of isolation (2PL & 2PC [24], Bernstein-Goodman [9], O2PL & 2PC [12], Bcast all, Bcast writes, Delayed bcast writes and Single bcast transactions [1], Fast Refresh Df-Im and Fast Refresh Im-Im [45], DBSM [47], Pronto [48], DBSM-RAC [63], Epidemic restricted and Epidemic unrestricted [30], DBSM* [68], DBSM-RO-opt and DBSM-RO-cons [44], One-at-a-time and Many-at-a-time [58], MPF and MCF [70], BaseCON for 1SR, BaseCON for SC and BaseCON for strong 1SR [69]) rely on the serializable isolation level (Di3) of their underlying databases, which adequately serializes transactions executed at each server. Other systems relax their correctness criteria or are able to increase the locally provided guarantees, and thus also relax the isolation level of their local databases. Snapshot (Di2) isolation (Lazy Txn Reordering [50], SI and Hybrid [35], NODO and REORDERING [46], RSI-PC [51], SRCA and

SRCA-Rep [42], PCSI Distr. Cert. [20], Tashkent-MW and Tashkent-API [21], Alg-Weak-SI, Alg-Str.-SI and Alg-Str.Ses.-SI [18], *k*-bound GSI [4], Tashkent+ [22], Mid-Rep [33], SIRC [56], Serrano et al. [60], AKARA [16], *g*B-SIRC [57]) or, more rarely, read committed (Di1) isolation (RSI-PC [51], SIRC [56], *g*B-SIRC [57]) are requested by such systems.

Among those systems not specifying a concrete level of isolation (Di0), two of them (Alsberg-Day [2] and RJDBC [23]) are based on the local concurrency control of their DBMSs and may function with different isolation levels at their local databases. The rest of the systems with a Di0 strategy perform concurrency control at the protocol layer and therefore they do not require any specific underlying isolation. This is the case of OTP-99 [36], which uses a queue per conflict class and allows transactions to proceed when they are at the head position of their queue. OTP, OTP-Q, OTP-DQ and OTP-SQ protocols [37] follow a similar approach but, in this case, there is a queue per data item and so transactions are not restricted to access only one conflict class.

Finally, there are few systems that require some customization (Di4) of their underlying databases. Thus, BTO & 2PC [7] and OPT & 2PC [61] require the maintenance of read and write timestamps for each data item; and SER, CS, Hybrid [35] and WCRQ [53] delay the acquisition of write locks until the remote phase of transactions.

Regarding group-life (Gl) communications, as linear interaction is costly, only few systems make such synchronization. While most of the systems follow a Gl0 strategy (no communication during local transaction execution), systems such as 2PL & 2PC [24], BTO & 2PC [7], Bernstein-Goodman [9], OPT & 2PC [61], O2PL & 2PC [12], Epidemic unrestricted [30] or the protocol by Serrano et al. [60] are obliged to use linear interaction due to their partial replication and the consequent distributed nature of their transactions, which may potentially require to access data items at other nodes. Bcast all, Bcast writes [1] and RJDBC [23] execute their significant operations in an active mode, sending a message for each of such operations to all servers (Gl3). Finally, Fast Refresh Im-Im [45] uses a Gl2 strategy to immediately propagate updates to secondary copies in order to increase their freshness.

Regarding the group-end (Ge) policy, as most of the systems do not apply read-only transactions at remote nodes, they neither broadcast them to the group upon commit request (Ge0). However, in some cases, read-only transactions require a synchronization point. Two of the surveyed systems are able to identify read-only transactions and manage them differently from update transactions while they still require certain synchronization at group-end for queries. This is the case of WCRQ [53], which sends queries to a read-quorum of replicas (Ge2) in order to provide those queries with strong consistency. The algorithm by Serrano et al. [60] also applies a Ge2 strategy for queries, in order to commit the distributed transaction at all participating sites.

In order to ensure replica consistency, a synchronization is always needed for update transactions, either at the beginning, at the end or after the execution of the transaction in its delegate node. OTP-99 [36], OTP, OTP-Q, OTP-DQ and OTP-SQ [37], the active processing of AKARA [16], BaseCON for 1SR, BaseCON for SC and BaseCON for strong 1SR [69] make the synchronization point of update transactions at the beginning (either with the client-request or the group-start policies), thus rendering unnecessary to synchronize with a group-end strategy (Ge0). Systems such as Fast Refresh Df-Im [45], RSI-PC [51], Alg-Weak-SI, Alg-Str.-SI and Alg-Str.Ses.-SI [18] choose a lazy synchronization after transaction commitment, and thus they also follow a Ge0 strategy.

Apart from the systems synchronizing update transactions at the beginning or after the commitment in the delegate, the rest of the systems make such synchronization at the end of the transaction in the delegate server, i.e., before the final commit operation, with a non-null group-end strategy. Alsberg-Day [2], which makes an update propagation in cascade mode, and the Tashkent family (Tashkent-MW and Tashkent-API [21], Tashkent+ [22]), which sends the writeset to a central certifier, follow the Ge1 strategy that requires the communication with only one server or component in the system. Among the remaining surveyed systems, some of them, based on partial replication, need to send transaction information only to a subset of system nodes (Ge2). This is the case of 2PL & 2PC [24], BTO & 2PC [7], Bernstein-Goodman [9], OPT & 2PC [61], O2PL & 2PC [12], Fast Refresh Im-Im [45], Epidemic restricted and Epidemic unrestricted [30]. The rest of the systems follow a Ge3 strategy, where the transaction information is broadcast to all nodes of the system.

In order to agree on the outcome of a broadcast transaction,[16] systems run the decision process. Weak voting, where a single node decides (Td1) and later communicates it decision to the rest of servers, as well as certification, where all nodes deterministically reach the same decision (Td2) are the preferred strategies. Only three of the surveyed systems base their decisions on the agreement of a quorum (Td3): One-at-a-time and Many-at-a-time [58], and WCRQ [53]. In One-at-a-time and Many-at-a-time, partial replication is used and nodes store information only about transactions that access items replicated at that node. To perform the decision process, nodes vote and then safely decide the outcome of a transaction T once they have received the votes of a voting quorum of T. In WCRQ [53], a read-quorum decides the outcome of a read-only transaction in order to ensure strong consistency, while write-quorums decide the commitment of update transactions. Finally, there are systems that base their decisions on the agreement of all servers (Td4): 2PL & 2PC [24], BTO & 2PC [7], Bernstein-Goodman [9], OPT & 2PC [61] and O2PL & 2PC [12], which all use the 2PC protocol; and DBSM-RAC [63], which employs a non-blocking atomic commit protocol.

The transaction-remote (Tr) policy defines the way transactions are applied at remote nodes. Most of the systems identify read-only transactions and do not apply them at remote servers (Tr0). The only exceptions are: Alsberg-Day [2], which is not specially tailored for database replication and thus it does not identify queries; Bcast all [1], where all operations are broadcast and executed in all the servers; Lazy Txn Reordering [50], where all transactions are broadcast and possibly reordered to minimize abortions; Pronto [48], which assimilates to an active approach by sending to the backups the SQL sentences instead of the writeset; RJDBC [23], where all significant operations (including the commit operation) are broadcast to all replicas; and AKARA [16], which broadcasts all transactions at their starting point. In all these systems, no different treatment is given to read-only transactions. On the other hand, the rest of the surveyed systems do not execute queries at remote nodes, but only update transactions. In order to increase performance, systems usually apply remote transactions in a concurrent manner (Tr1), by controlling, either at the protocol level or inside the database, that conflicting transactions are applied in the same order at all replicas. However, to avoid the possible increase in complexity of such control, many systems apply writesets in a sequential, non-overlapping manner (Tr2).

With regard to the client-response (Cr) policy, only one of the surveyed systems, Pronto [48], returns multiple responses to the client (Cr2), whereas the rest of the systems always opt for returning a single answer (Cr1). The Cr2 client-response policy may require further processing in the client to select or compute a final result if multiple *different* answers are sent.

Group-after (Ga) policies can seriously affect consistency, in that update propagation outside the scope of transactions may lead to inconsistent states in different replicas. Thus, when using lazy propagation special care must be taken to ensure that consistency is maintained or that some reconciliation mechanisms are able to restore the system to a consistent state. Only few of the surveyed systems follow a non-null group-after strategy: Alsberg-Day [2], Fast Refresh Df-Im [45], RSI-PC [51], Alg-Weak-SI, Alg-Str.-SI and Alg-Str.Ses.-SI [18]. All these systems consider a primary copy configuration, where updates are made at only one node (the primary copy of the system or the master site of the updated data) and are later lazily propagated to the slaves or secondary nodes. As only one node processes updates, no inconsistencies are introduced.

5 Conclusion

In this paper we present a characterization model that provides a common framework to describe and compare different database replication systems. This model is the result of a careful analysis of different community proposals made since the beginning of this research field. In this study, we identify the relevant steps that are common to all replication protocols, and the different approaches that protocols follow in such steps. A policy is associated with each step, and the different approaches or options are called strategies. Policies are grouped into families, according to the relation among the interactions they regulate. With this model, we can detail the strategy that each protocol follows for each of its main steps.

[16] Those systems that locally commit queries without any communication with the rest of the nodes usually employ the bottom strategy (Td0) for such read-only transactions: no decision process is run for them.

This model is then used in order to characterize more than 50 database replication systems, in an extensive survey that reviews the chronological evolution of this research field. While many different strategies have been followed in order to accomplish the required system interactions, some of them seem to have been preferred over others, due to several reasons, from performance issues, to easiness of protocol design and implementation.

Acknowledgments

This work has been partially supported by EU FEDER and the Spanish MICINN under grants TIN2009-14460-C03 and TIN2010-17193; and by the Spanish MEC under grant BES-2007-17362.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting Atomic Broadcast in Replicated Databases (Extended Abstract). In *3rd International Euro-Par Conference on Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science (LNCS)*, pages 496–503. Springer, 1997.

[2] P. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *2nd International Conference on Software Engineering (ICSE)*, pages 562–570. IEEE-CS Press, 1976.

[3] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-End Databases of Dynamic Content Web Sites. In *ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science (LNCS)*, pages 282–304. Springer, 2003.

[4] J. E. Armendáriz-Íñigo, J. R. Juárez-Rodríguez, J. R. González de Mendívil, H. Decker, and F. D. Muñoz-Escoí. *k*-bound GSI: A Flexible Database Replication Protocol. In *ACM Symposium on Applied Computing (SAC)*, pages 556–560. ACM, 2007.

[5] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A Critique of ANSI SQL Isolation Levels. In *ACM SIGMOD International Conference on Management of Data*, pages 1–10. ACM, 1995.

[6] P. A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM (CACM)*, 39(2):86–98, 1996.

[7] P. A. Bernstein and N. Goodman. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *6th International Conference on Very Large Data Bases (VLDB)*, pages 285–300. IEEE-CS Press, 1980.

[8] P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[9] P. A. Bernstein and N. Goodman. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. *ACM Transactions on Database Systems (TODS)*, 9(4):596–615, 1984.

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[11] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *The VLDB Journal*, 1(2):181–239, 1992.

[12] M. J. Carey and M. Livny. Conflict Detection Tradeoffs for Replicated Data. *ACM Transactions on Database Systems (TODS)*, 16(4):703–746, 1991.

[13] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *FREENIX Track, USENIX Annual Technical Conference*, pages 9–18. USENIX Association, 2004.

[14] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

[15] A. Correia Jr., A. L. Sousa, L. Soares, F. Moura, and R. C. Oliveira. Revisiting Epsilon Serializabilty to improve the Database State Machine (Extended Abstract). In *SRDS Workshop on Dependable Distributed Data Management (WDDDM)*, 2004.

[16] A. Correia Jr., J. Pereira, and R. C. Oliveira. AKARA: A Flexible Clustering Protocol for Demanding Transactional Workloads. In *OTM Confederated International Conferences (Part I)*, volume 5331 of *Lecture Notes in Computer Science (LNCS)*, pages 691–708. Springer, 2008.

[17] K. Daudjee and K. Salem. Lazy Database Replication with Ordering Guarantees. In *20th International Conference on Data Engineering (ICDE)*, pages 424–435. IEEE Computer Society, 2004.

[18] K. Daudjee and K. Salem. Lazy Database Replication with Snapshot Isolation. In *32nd International Conference on Very Large Data Bases (VLDB)*, pages 715–726. ACM, 2006.

[19] R. De Prisco, B. W. Lampson, and N. A. Lynch. Revisiting the Paxos Algorithm. In *11th International Workshop on Distributed Algorithms (WDAG)*, volume 1320 of *Lecture Notes in Computer Science (LNCS)*, pages 111–125. Springer, 1997.

[20] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database Replication Using Generalized Snapshot Isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 73–84. IEEE-CS Press, 2005.

[21] S. Elnikety, S. G. Dropsho, and F. Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. In *EuroSys Conference*, pages 117–130. ACM, 2006.

[22] S. Elnikety, S. G. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases. In *EuroSys Conference*, pages 399–412. ACM, 2007.

[23] J. Esparza Peidro, F. D. Muñoz-Escoí, L. Irún-Briz, and J. M. Bernabéu-Aubán. RJDBC: A Simple Database Replication Engine. In *6th International Conference on Enterprise Information Systems (ICEIS)*, pages 587–590, 2004.

[24] J. Gray. Notes on Database Operating Systems. In *Advanced Course: Operating Systems*, volume 60 of *Lecture Notes in Computer Science (LNCS)*, pages 393–481. Springer, 1978.

[25] J. Gray and L. Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[26] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *ACM SIGMOD International Conference on Management of Data*, pages 173–182. ACM, 1996.

[27] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, 1994.

[28] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[29] J. Holliday, D. Agrawal, and A. El Abbadi. The Performance of Database Replication with Group Multicast. In *29th IEEE International Symposium on Fault-Tolerant Computing Systems (FTCS)*, pages 158–165. IEEE-CS Press, 1999.

[30] J. Holliday, D. Agrawal, and A. El Abbadi. Partial Database Replication using Epidemic Communication. In *22nd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 485–493. IEEE-CS Press, 2002.

[31] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and S. Arévalo. A Low-Latency Non-blocking Commit Service. In *15th International Conference on Distributed Computing (DISC)*, volume 2180 of *Lecture Notes in Computer Science (LNCS)*, pages 93–107. Springer, 2001.

[32] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *22nd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 477–484. IEEE-CS Press, 2002.

[33] J. R. Juárez-Rodríguez, J. E. Armendáriz-Íñigo, J. R. González de Mendívil, F. D. Muñoz-Escoí, and J. R. Garitagoitia. A Weak Voting Database Replication Protocol Providing Different Isolation Levels. In *7th International Conference on New Technologies of Distributed Systems (NOTERE)*, pages 261–268, 2007.

[34] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, a New Way to Implement Database Replication. In *26th International Conference on Very Large Data Bases (VLDB)*, pages 134–143. Morgan Kaufmann, 2000.

[35] B. Kemme and G. Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Transactions on Database Systems (TODS)*, 25(3):333–379, 2000.

[36] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 424–431. IEEE-CS Press, 1999.

[37] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using Optimistic Atomic Broadcast in Transaction Processing Systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(4):1018–1032, 2003.

[38] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2): 133–169, 1998.

[39] L. Lamport. Paxos Made Simple. *ACM SIGACT News* (Distributed Computing Column), 32(4): 18–25, 2001.

[40] B. W. Lampson. Atomic transactions. In *Advanced Course: Distributed Systems*, volume 105 of *Lecture Notes in Computer Science (LNCS)*, pages 246–265. Springer, 1981.

[41] B. W. Lampson. How to Build a Highly Available System Using Consensus. In *10th International Workshop on Distributed Algorithms (WDAG)*, volume 1151 of *Lecture Notes in Computer Science (LNCS)*, pages 1–17. Springer, 1996.

[42] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation. In *ACM SIGMOD International Conference on Management of Data*, pages 419–430. ACM, 2005.

[43] F. D. Muñoz-Escoí, J. M. Bernabé-Gisbert, R. de Juan-Marín, J. E. Armendáriz-Iñigo, and J. R. González de Mendívil. Revising 1-Copy Equivalence in Replicated Databases with Snapshot Isolation. In *OTM Confederated International Conferences (Part I)*, volume 5870 of *Lecture Notes in Computer Science (LNCS)*, pages 467–483. Springer, 2009.

[44] R. C. Oliveira, J. Pereira, A. Correia Jr., and E. Archibald. Revisiting 1-Copy Equivalence in Clustered Databases. In *ACM Symposium on Applied Computing (SAC)*, pages 728–732. ACM, 2006.

[45] E. Pacitti, P. Minet, and E. Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. In *25th International Conference on Very Large Data Bases (VLDB)*, pages 126–137. Morgan Kaufmann, 1999.

[46] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *14th International Conference on Distributed Computing (DISC)*, volume 1914 of *Lecture Notes in Computer Science (LNCS)*, pages 315–329. Springer, 2000.

[47] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, 1999.

[48] F. Pedone and S. Frølund. Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases. In *19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 176–185. IEEE-CS Press, 2000.

[49] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In *12th International Symposium on Distributed Computing (DISC)*, volume 1499 of *Lecture Notes in Computer Science (LNCS)*, pages 318–332. Springer, 1998.

[50] F. Pedone, R. Guerraoui, and A. Schiper. Transaction Reordering in Replicated Databases. In *16th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 175–182. IEEE-CS Press, 1997.

[51] C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *ACM/IFIP/USENIX International Middleware Conference*, volume 3231 of *Lecture Notes in Computer Science (LNCS)*, pages 155–174. Springer, 2004.

[52] Y. Raz. The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment. In *18th International Conference on Very Large Data Bases (VLDB)*, pages 292–312. Morgan Kaufmann, 1992.

[53] L. Rodrigues, N. Carvalho, and E. Miedes. Supporting Linearizable Semantics in Replicated Databases. In *7th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 263–266. IEEE-CS Press, 2008.

[54] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems (TODS)*, 3(2):178–198, 1978.

[55] M. I. Ruiz-Fuertes and F. D. Muñoz-Escoí. Refinement of the One-Copy Serializable Correctness Criterion. Technical Report ITI-SIDI-2011/004, Instituto Tecnológico de Informática, Valencia, Spain, 2011.

[56] R. Salinas, J. M. Bernabé-Gisbert, F. D. Muñoz-Escoí, J. E. Armendáriz-Íñigo, and J. R. González de Mendívil. SIRC: A Multiple Isolation Level Protocol for Middleware-based Data Replication. In *22nd International Symposium on Computer and Information Sciences (ISCIS)*, pages 1–6. IEEE-CS Press, 2007.

[57] R. Salinas, F. D. Muñoz-Escoí, J. E. Armendáriz-Íñigo, and J. R. González de Mendívil. A Performance Evaluation of g-Bound with a Consistency Protocol Supporting Multiple Isolation Levels. In *OTM Confederated International Workshops and Posters*, volume 5333 of *Lecture Notes in Computer Science (LNCS)*, pages 914–923. Springer, 2008.

[58] N. Schiper, R. Schmidt, and F. Pedone. Optimistic Algorithms for Partial Database Replication. In *10th International Conference on Principles of Distributed Systems (OPODIS)*, volume 4305 of *Lecture Notes in Computer Science (LNCS)*, pages 81–93. Springer, 2006.

[59] F. B. Schneider. Synchronization in Distributed Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):125–148, 1982.

[60] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme. Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation. In *13th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 290–297. IEEE-CS Press, 2007.

[61] M. K. Sinha, P. D. Nanadikar, and S. L. Mehndiratta. Timestamp Based Certification Schemes for Transactions in Distributed Database Systems. In *ACM SIGMOD International Conference on Management of Data*, pages 402–411. ACM, 1985.

[62] D. Skeen. Nonblocking Commit Protocols. In *ACM SIGMOD International Conference on Management of Data*, pages 133–142. ACM, 1981.

[63] A. L. Sousa, R. C. Oliveira, F. Moura, and F. Pedone. Partial Replication in the Database State Machine. In *IEEE International Symposium on Network Computing and Applications (NCA)*, pages 298–309. IEEE-CS Press, 2001.

[64] I. L. Traiger, J. Gray, C. A. Galtieri, and B. G. Lindsay. Transactions and Consistency in Distributed Database Systems. *ACM Transactions on Database Systems (TODS)*, 7(3):323–342, 1982.

[65] M. Wiesmann and A. Schiper. Comparison of Database Replication Techniques Based on Total Order Broadcast. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(4):551–566, 2005.

[66] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding Replication in Databases and Distributed Systems. In *20th International Conference on Distributed Computing Systems (ICDCS)*, pages 464–474. IEEE-CS Press, 2000.

[67] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database Replication Techniques: A Three Parameter Classification. In *19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 206–215. IEEE-CS Press, 2000.

[68] V. Zuikevičiūtė and F. Pedone. Revisiting the Database State Machine Approach. In *VLDB Workshop on Design, Implementation and Deployment of Database Replication*, 2005.

[69] V. Zuikevičiūtė and F. Pedone. Correctness Criteria for Database Replication: Theoretical and Practical Aspects. In *OTM Confederated International Conferences (Part I)*, volume 5331 of *Lecture Notes in Computer Science (LNCS)*, pages 639–656. Springer, 2008.

[70] V. Zuikevičiūtė and F. Pedone. Conflict-Aware Load-Balancing Techniques for Database Replication. In *ACM Symposium on Applied Computing (SAC)*, pages 2169–2173. ACM, 2008.