Chapter 12
Database Replication: A Tutorial

Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and
Gustavo Alonso

Abstract This chapter provides an in-depth introduction to database replication, in particular how transactions are executed in a replicated environment. We describe a suite of replication protocols and illustrate the design alternatives using a two-step approach. We first categorize replication protocols by only two parameters and present a simple example protocol for each of the resulting categories. Further parameters are then introduced, and we illustrate them by the given replication protocols and some variants.

12.1 Introduction

12.1.1 Why Replication

Database replication has been studied for more than three decades. It is concerned with the management of data copies residing on different nodes and with each copy controlled by an independent database engine. A main challenge of database replication is *replica control*: when data can be updated, replica control is in charge of keeping the copies consistent and providing a globally correct execution. Consistency needs to be enforced through some protocol running across the different nodes so that the independent database engines can make local decisions that still provide some form of global consistency when the system is considered as a whole.

Some particular characteristics differentiate database replication from replication approaches in other domains of distributed computing. While database replication can be used for fault-tolerance and high-availability in a similar spirit as replication is used in distributed computing in general, there are many other purposes of replication. Foremost, often the primary purpose of database replication is to increase the performance and improve the scalability of database engines. Having more database replicas distributed across geographic regions provides fast access to local copies, having a cluster of database replicas provides high throughput. Finally, for some applications replication is a natural choice, e.g., in the context of mobile users that are frequently disconnected from the corporate data server, or for data warehouses, which have to reformat data in order to speed up query processing. These various use cases bring to the fore a tradeoff between consistency and performance, which attracts less attention when replication is targeted at high availability, as studied in previous chapters of this book.

A further difference is that databases, by their name, are *data centric*, usually consisting of a large set of individual data items, and access to this data is enclosed in *transactions*. This means that the database supports operations which are grouped together in transactions, rather than being processed independently of one another. A transaction reflects a logical unit of execution and is defined as a sequence of read and write operations. Transactions come with a set of properties. *Atomicity* requires a transaction to either execute entirely and commit, or abort and not leave any changes in the database. *Isolation* provides a transaction with the impression that no other transaction is currently executing in the system. *Durability* guarantees that once the initiator of the transaction has received the confirmation of commit, the changes of the transaction are, indeed, reflected in the database (they can, of course, later be overwritten by other transactions). Transactions are a particularity of database systems and provide additional challenges compared to object or process replication, as the latter usually only consider individual read and write operations.

12.1.2 Organization of the Chapter

In this chapter we provide a systematic introduction to the principles of replica control algorithms. In the following, the term *replica* mostly refers to a site running the database system software and storing a copy of the entire database. But depending on the context, a replica can also refer to the physical copy of an individual data item. Replica control has to translate the operations that transactions submit on logical data items into operations on the physical data copies. Most algorithms are based on a specific form of *read-one-write-all-(available)* (ROWAA) approach where a transaction is assigned to one site (replica), where it is *local*, and all its read and write operations are executed at this local site. The write operations are also executed at all other replicas in the form of a *remote* transaction. This chapter only considers ROWAA protocols. We refer readers interested in quorum systems for database replication to [24]. Replica control also has to make sure that the copies are consistent. Ideally, all copies of a data item have the same value at all times. In reality, many different levels of consistency exist.

We first introduce two main parameters that were presented by Gray et al. [20] to provide a coarse categorization of replication algorithms. The *transaction location* determines where update transactions are executed, namely either at a *primary replica* or at *any replica*. The *synchronization strategy* determines when update propagation takes place, either before or after commit of a transaction. We use these parameters as a basis to develop a suite of replication algorithms that serve as examples and illustrate the principles behind the tasks of replica control. We keep the description at an abstract level and provide very simple algorithms in order to better illustrate the principles. Many issues are only discussed informally.

As a second step, we discuss a wide range of other parameters. In particular, we have a closer look at the level of correctness provided by different replica control solutions in regard to atomicity, isolation and durability. We also have a closer look at the choice of concurrency control mechanism (e.g., optimistic vs. pessimistic), the degree of replication (full vs. partial replication), and other design choices that can have a significant influence on the performance and practicality of the replication solution.

At the end of this chapter we present some of the research and commercial replication solutions, and how they fit into our categorization. We also discuss the relationship between replication and related areas of data management, such as materialized views, caching, and parallel database systems. They all maintain data copies, and data maintenance can, to some degree, be categorized with the parameters presented in this chapter. However, they have some fundamental differences that we would like to point out.

In a replicated database, the replica control needs to interact closely with the general transaction management. Transaction management includes concurrency control (which delivers isolation) and logging to allow an aborting transaction to roll back any changes previously made (this is part of ensuring atomicity). In most of this chapter, we assume that each site uses strict two-phase locking (2PL) for concurrency control; that is, an exclusive lock is acquired on an item before that item is written, a shared lock is acquired before reading the item, and each lock is held until the commit or abort of the transaction holding the lock. Other concurrency control techniques are possible, as we discuss briefly in Section 12.4.3. More details can be found in [7].

Another essential feature of all replica control protocols is the inter-site communication. Since we discuss ROWAA approaches, write operations must be sent to all available sites. We assume that a primitive, called multicast, is used to propagate information to all replicas. In most of the chapter, we require FIFO multicast, which means that all recipients get the messages from the same sender in the order they were sent. We will introduce more powerful multicast primitives later in the chapter when we discuss protocols that take advantage of them.

12.2 Basic Taxonomy for Replica Control Approaches

Gray et. al [20] categorize replica control solutions by only two parameters. The parameter *transaction location* indicates *where* transactions can be executed. In general, a transaction containing only read operations can be executed at any replica. Such a read-only transaction is then called a local transaction at this replica. For update transactions, i.e., transactions that have at least one write operation, there exist two possibilities. In a *primary copy* approach, all update transactions are executed at a given replica (the primary). The primary propagates the write operations these transactions perform to the other replicas (secondaries) (see Chapter 2 for primary-backup object replication). This model is also called passive replication (as in Chapters 11 and 13). In contrast, in an *update anywhere* approach, update transactions can be executed at any site, just as the read-only transactions which then

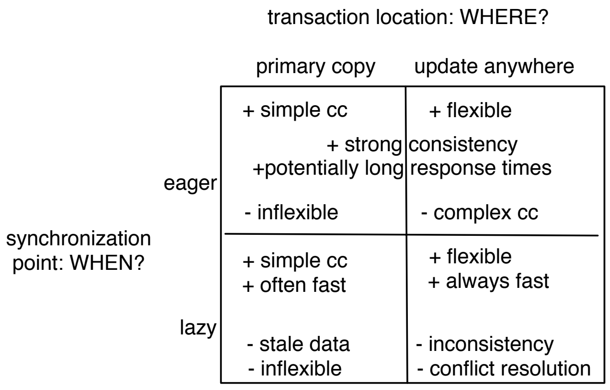

Fig. 12.1 Categories.

takes care of update propagation. This model is also called multi-primary (as in Chapter 11). Using a primary copy approach, conflicts between concurrent update transactions can all be detected at the primary while an update anywhere approach requires a more complex distributed concurrency control mechanism. However, a primary copy approach is less flexible as all update transactions have to be executed at a specific replica.

The *synchronization strategy* determines when replicas coordinate in order to achieve consistency. In *eager* replication, coordination for the updates of a transaction takes place before the transaction commits. With *lazy* replication, updates are asynchronously propagated after the transaction commits. Eager replication often results in longer client response times since communication takes place before a commit confirmation can be returned, but it can provide strong consistency more easily than lazy replication (see Chapter 1 for a definition of strong consistency).

Using these two parameters, there are four categories as shown in Figure 12.1, and each replica control algorithm belongs to one of these categories. The definitions so far contain some ambiguity and we will refine them later. Each category has its own implications in regard to performance, flexibility and the degree of consistency that can be achieved. We illustrate these differences by providing an example algorithm for each of the categories. At the same time, these algorithms provide an intuition for the main building blocks needed for replica control, and also reflect some other design choices that we will discuss in detail later.

In the following algorithms, a transaction T_i submits a sequence of read and write operations. A read operation $r_i(x)$ accesses data item x, a write operation $w_i(x, v_i)$ sets data item x to value v_i. At the end of execution, a transaction either submits a commit request to successfully terminate a transaction or an abort request to undo all the updates it has performed so far. We ignore the possibility that operations might fail due to application semantics (e.g., updating a non-existing record). The algorithms, as we describe them, do not consider failures, e.g., failure of replicas or the network. Our discussions, however, mention, how the protocols could be extended in this respect.

12.2.1 Eager Primary Copy

Eager primary copy protocols are probably the simplest protocol type to understand. We present a protocol that is a straightforward extension of non-replicated transaction execution.

Example Protocol

Figure 12.2 shows an example protocol using strict 2-phase-locking (2PL) for concurrency control at each replica. When a client submits a transaction T_i it sends all the operations of this transaction to one replica R. T_i is then a local transaction at R and R is responsible for returning the corresponding responses for the requests associated with T_i. Update transactions may only be submitted to the primary. Read operations are executed completely locally (lines 1-5). They acquire a shared lock before accessing the local copy of the data item. For a write operation (lines 6-12), the primary replica first acquires an exclusive lock locally and performs the update. Then it multicasts the request to the other replicas in FIFO order. Aborts can occur due to deadlock (lines 13-16). If an update transaction aborts the primary informs the secondary replicas if they had already received some write requests for this transaction. Similar actions are needed if the client decides to abort the transaction (line 17). When the client submits the commit request after completion of the transaction (lines 18-24), an update transaction needs to run a 2-phase-commit protocol (2PC) to guarantee the atomicity of the transaction (we discuss this later in detail). The primary becomes the coordinator of the 2PC. The 2PC guarantees that all decide on the same commit/abort outcome of the transaction and that all secondaries have successfully executed the transaction before it commits at the primary. Read-only transactions can simply be committed locally. After successful commit (lines 22-24), the transaction releases its lock and the local replica returns the confirmation to the client. When a secondary replica receives a write request for a transaction from the primary (lines 25-26), it acquires an exclusive lock (in the order in which messages are received), and executes the operation. Although an update transaction might be involved in a deadlock at the secondary, it is not aborted at the secondary. If it is a deadlock among update transactions only, the primary will detect such a deadlock and act appropriately. If the deadlock involves local read-only transactions, those read-only transactions need to be aborted. When a secondary receives an abort request for an update transaction it has to abort it locally.

Example Execution

Figure 12.3 shows a simple example execution under this protocol. In this and all following examples, time passes from top to bottom. Furthermore, T_1 acquiring a shared lock on data item x is denoted as $S_1(x)$, and T_1 acquiring an exclusive lock on x is denoted as $X_1(x)$. $T_1 = r_1(x), w_1(x, v_1)$ is an update transaction local at primary $R1$. $T_2 = r_2(y), r_2(x)$ is a read-only transaction local at secondary $R2$. Read operations are executed locally. The write operation of T_1 is multicast to all replicas and executed everywhere. When T_1 wants to commit it requires a 2-phase-commit

```
Upon: $r_i(x)$ for local transaction $T_i$
 1: acquire shared lock on $x$
 2: if deadlock then
 3:     call $abortHandling(T_i)$
 4: else
 5:     return $x$

Upon: $w_i(x, v_i)$ for local transaction $T_i$    {only at
    primary replica}
 6: acquire exclusive lock on $x$
 7: if deadlock then
 8:     call $abortHandling(T_i)$
 9: else
10:     $x := v_i$
11:     multicast $w_i(x, v_i)$ to secondaries in FIFO
        order
12:     return ok

Upon: $abortHandling(T_i)$
13: if $T_i$ update transaction and at least one
    $w_i(x, v_i)$ was multicast then
14:     multicast $abort(T_i)$ to all replicas
15: abort $T_i$, release locks of $T_i$
16: return aborted
Upon: abort request for local transaction $T_i$
17: call $abortHandling(T_i)$
Upon: commit request for local transaction $T_i$
18: if $T_i$ update transaction then
19:     run 2PC among all replicas to commit $T_i$
20: else
21:     commit $T_i$

Upon: successful commit of transaction $T_i$
22: release locks of $T_i$
23: if $T_i$ local transaction then
24:     return committed

Upon: receiving $w_j(x, v_j)$ of remote
    transaction $T_j$ from primary replica
25: acquire exclusive lock on $x$
26: $x := v_j$

Upon: receiving $abort(T_j)$ for remote
    transaction $T_j$ from primary replica
27: abort $T_j$, release locks of $T_j$
```

Fig. 12.2 Eager Primary Copy Algorithm.

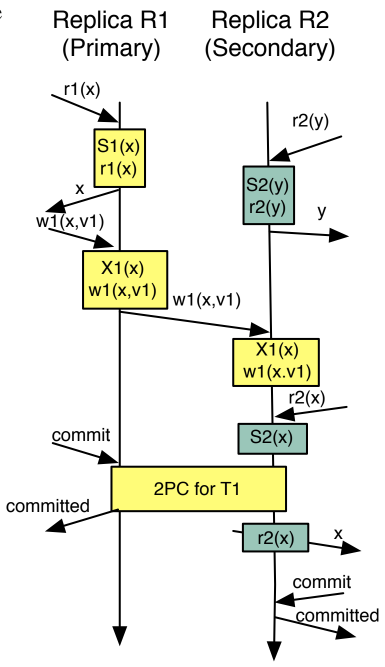

Fig. 12.3 Eager Primary Copy Example.

protocol. When T_2 requests the shared lock on x it has to wait because T_1 holds the lock. Only after T_1 commits can T_2 get the lock and complete. If there were a deadlock between T_1 and T_2, the secondary would abort read-only transaction T_2.

Discussion

Advantages In most primary copy approaches concurrency control is nearly the same as in a non-replicated system. For our protocol, the only difference is that secondaries have request locks for update operations in the order they receive them from the primary. And in case of deadlock they have to abort local read-only transactions. Since secondaries execute conflicting updates in the same order as they are executed at the primary and strict 2PL is used, execution is globally serializable. This means that the concurrent execution of transactions over the physical copies is equivalent to a serial execution of the transactions over a single logical copy of the database.

As execution is eager and a 2PC is run, all copies are virtually consistent (have the same value at commit time of transactions). This means that read operations at the secondaries never read stale data. This also provides strong guarantees in case of failures. If the primary fails, active transactions are aborted. For all committed transactions it is guaranteed that the secondaries have the updates. Therefore, if one of the secondaries takes over and becomes the new primary, no updates are lost. However, if the primary has failed during a 2PC some transactions might be blocked. This could be resolved by the system administrator. Clients connected to the primary can reconnect to the new primary. Some systems provide automatic reconnection.

Disadvantages Requiring all update transactions to execute at the primary leads to a loss of replication transparency and flexibility. Clients need to know that only the primary replica can execute update transactions. Some primary copy systems automatically redirect update transactions to the primary. However, in this case, the system needs to know at the start of a transaction whether it is an update transaction or not even if the first operation submitted is a read operation, as is the case in the example of Figure 12.3.

The price to pay for an eager protocol that uses 2PC are long execution times for update transactions as they only commit at the primary once they have completely executed at all secondaries. We discuss in Section 12.2.5 some eager protocols that do not have this behavior.

12.2.2 Eager Update Anywhere

Eager update anywhere replica control algorithms were the first replica control algorithms to be proposed in the literature. The early algorithms extended traditional concurrency control mechanisms to provide globally serializable execution with a large emphasis on correctly handling failures and recoveries.

Example Protocol

Figure 12.4 shows the changes to the eager primary copy algorithm of Figure 12.2 to allow for update anywhere. Both read-only and update transactions can now be local at any replica which coordinates their execution. Read operations are executed as before at the local replica. A write operation has to execute at all replicas (lines 1-11). The local replica multicasts the request to the other replicas and then acquires an exclusive lock locally and performs the update. Then, it waits for acknowledgements from all other replicas before returning the ok to the client. The acknowledgements are needed as conflicting requests might now occur in different order at the different replicas and it is not guaranteed that the remote replicas can execute the request in the same order. In fact, distributed deadlocks can occur, as we discuss below. Aborts for local transactions are handled as in the primary copy protocol. Commits are handled as before with the only difference that the 2PC can now be initiated by any replica that wants to commit a local update transaction. When a replica receives a write request from a transaction that is local at another replica (lines 12-17), it acquires an exclusive lock, executes the operations and sends an acknowledgement back to the local replica. When a deadlock is detected, it might involve remote transactions. The system can choose to abort a remote transaction; if that is the case, the replica where the transaction is local is informed accordingly. Similarly, any replica has to abort a remote transaction when it is informed by the transaction's local replica (line 18).

Example Execution

Figure 12.5 shows an example execution under this protocol indicating the special case of a distributed deadlock. This time, $T_1 = r_1(x), w_1(x, v1)$ is local at $R1$. $T_2 = r_2(y), w_2(x, v2)$ also updates x and is local at $R2$. As the lock requests in this execution were processed in different orders at the two replicas, there is a deadlock. This cannot be detected with information from a single site, but the system must have a distributed deadlock mechanism or timeout. In this execution, T_2 is chosen to abort.

Discussion

Advantages Being an update anywhere approach it is more flexible than the primary copy approach and provides transparency as it allows update transactions to be submitted to any replica. As it is again eager, using a 2PC, all data copies are virtually consistent. Failures are tolerated without loss of correctness. Given that the protocol extends strict 2PL, it provides global serializability.

Disadvantages Although the concurrency control mechanism appears very similar to the one of the eager primary copy approach, the complexity is higher as distributed deadlocks may occur. As distributed algorithms to detect distributed deadlocks are expensive, many systems use timeouts, but these are hard to set sensibly. If

```
Upon: $w_i(x, v_i)$ for local transaction $T_i$
 1: multicast $w_i(x, v_i)$ to all other replicas
 2: acquire exclusive lock on $x$
 3: if deadlock then
 4:     call $abortHandling(T_i)$
 5: else
 6:     $x = v_i$
 7:     wait for all to return answer
 8:     if all return ok then
 9:         return ok
10:     else
11:         call $abortHandling(T_i)$

Upon: receiving $w_j(x, v_j)$ of
      remote transaction $T_j$ from replica $R'$
12: acquire exclusive lock on $x$
13: if deadlock then
14:     send abort($T_j$) to $R'$
15: else
16:     $x := v_j$
17:     send ok back to $R'$

Upon: commit request for local transaction $T_i$
18: if $T_i$ update transaction then
19:     run 2PC among all replicas to commit $T_i$
20: else
21:     commit $T_i$

Upon: successful commit of transaction $T_i$
22: release locks of $T_i$
23: if $T_i$ local transaction then
24:     return committed

Upon: receiving $abort(T_j)$ for
      remote transaction $T_j$ from replica $R'$
25: abort $T_j$, release locks of $T_j$
```

Fig. 12.4 Eager Update Anywhere Algorithm.

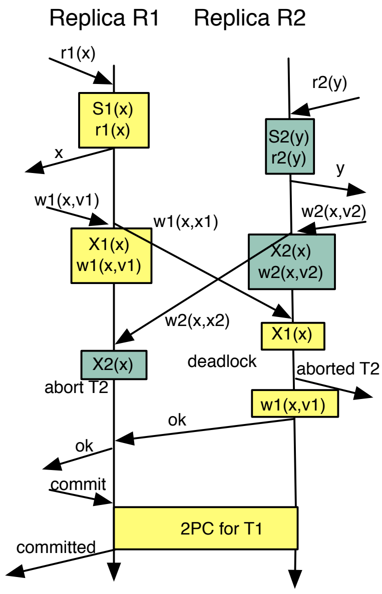

Fig. 12.5 Eager Update Anywhere Example.

too short, many transactions are aborted unnecessarily. If set too long, a few deadlocks can block large parts of the database leading quickly to deterioration. This was one of the reasons, why Gray et. al [20] indicated that traditional replication solutions do not scale.

Comparing the two protocols presented so far, the update anywhere protocol is likely to have longer response times than the primary copy protocol as each write operation has to be executed at all replicas before the next operation can start. However, this is a consequence of the protocol detail, rather than being intrinsic to the update anywhere and primary copy styles. The primary copy protocol could also wait for each write to be executed at the secondaries before proceeding, or the update anywhere protocol could simply multicast the writes as the primary copy protocol without waiting. Conflicts would then be resolved at commit time. One could even change the protocols and send all changes in a single message only at the end of transaction. We will discuss the issue of message overhead and number of messages per transaction in Section 12.4.1. At this point, we only want to point out that we have consciously chosen to present different flavors of protocols to already give an indication that there are many design alternatives.

12.2.3 Lazy Primary Copy

Compared to eager approaches, lazy approaches do not have any communication among replicas during transaction execution. A transaction can completely execute and commit at one replica and updates are only propagated to the other replicas after commit. Combining lazy with primary copy leads to a quite simple replication approach.

Example Protocol

Figure 12.6 presents a simple lock-based lazy primary copy protocol. Read-only transactions are executed as in the eager approach (lines 1-5). Update transactions may only be submitted to the primary which executes both read and write operations locally (lines 1-5 and 6-11). Therefore, when the transaction aborts during execution or when the client requests it (line 14), the abort remains local (lines 12-13). When the client submits a commit (lines 15-18), the transaction commits first locally and only after commit are all write operations multicast within a single message, often referred to as write set. These writesets are multicast in FIFO order. The multicast can be directly after the commit or some time after, e.g., in certain time-intervals. The secondaries, upon receiving such a writeset (lines 19-23) acquire locks in receiving order to make sure that they serialize conflicting transactions in the same way as the primary.

Example Execution

Figure 12.7 shows an example where $T_1 = r_1(x), w_1(x, v1), w_1(y, w1)$ executes at the primary updating x and y. Only after commit the updates are sent to the secondary. At the secondary, the locks for the updates are requested. As there is a deadlock with a local transaction $T_2 = r_2(y), r_2(x)$, the local transaction has to abort in order to apply the updates of T_1.

Discussion

Advantages Being a primary copy approach, concurrency control remains simple while serializability is provided. It is similar to the eager primary copy approach as secondaries apply updates in receiving order. In contrast to eager approaches the response times of update transactions are not delayed by communication and coordination overhead among the replicas which can potentially lead to shorter response times.

```
Upon: $r_i(x)$ for local transaction $T_i$
 1: acquire read lock on $x$
 2: if deadlock then
 3:     call $abortHandling(T_i)$
 4: else
 5:     return $x$

Upon: $w_i(x, v_i)$ for local transaction $T_i$    {only at primary replica}
 6: acquire exclusive lock on $x$
 7: if deadlock then
 8:     call $abortHandling(T_i)$
 9: else
10:     $x := v_i$
11:     return ok

Upon: $abortHandling(T_i)$
12: abort $T_i$, release locks of $T_i$
13: return aborted

Upon: abort request for local transaction $T_i$
14: call $abortHandling(T_i)$

Upon: commit request for local transaction $T_i$
15: commit $T_i$, release locks
16: return committed
17: if $T_i$ update transaction then
18:     send all $w_i$ of $T_i$ in single write set
        sometime after commit
        in FIFO order

Upon: receiving write set message of
      remote transaction $T_j$ from primary replica
      in FIFO order
19: for all $w_j(x, v_j)$ in write set do
20:     acquire exclusive lock on $x$
21: for all $w_j(x, v_j)$ in write set do
22:     $x := v_j$
23: commit $T_j$, release locks
```

Fig. 12.6 Lazy Primary Copy Algorithm.

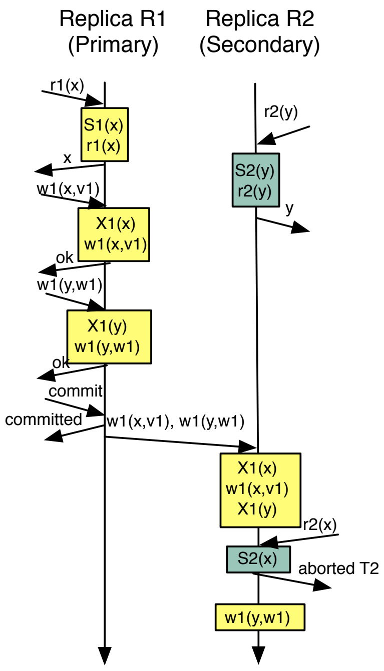

Fig. 12.7 Lazy Primary Copy Example.

Disadvantages While a lazy primary copy approach easily provides serializability or other strong isolation levels, lazy replication provides an inherent weaker consistency than eager replication. Lazy replication does not provide the virtual consistency shown in eager approaches. At the time a transaction commits at the primary, the data at the secondary becomes stale. Thus, read operations that access the secondaries before the writeset is processed read outdated data. Note that serializability is still provided, with read-only transactions that read stale data being serialized before the update transactions although those happened earlier in time.

More severe, if the site executing and committing an update transaction fails before propagating the writeset, the other replicas are not aware of the transaction. If another site takes over as primary, this transaction is lost. When the failed replica recovers it might try to reintegrate the changes into the existing system, but the database might have changed considerably since then. We can consider this a loss of the durability guarantee. We will discuss this later in more detail.

Although there is no communication among replicas during transaction execution, transactions are not necessarily faster than in an eager approach. If the clients and replicas are geographically distributed in a WAN, then clients that are not close to the primary copy experience long response times as they have to interact with a remote primary copy. That is, there can still be considerable communication delay between client and primary copy. To address this issue, many commercial systems partition the database, with each partition having the primary copy on a different replica. In geographically distributed applications, a database can often be partitioned by regions. Clients that are local to one region typically access mostly the partition of the database that is relevant for this region. Thus, the replica of a region becomes the primary for the corresponding partition. Therefore, for most clients the primary of the data they access will be close and client/replica interaction will be fast. A challenge with this approach is to find appropriate partitions. Also, the programmer has to be aware to write code so that each transaction only accesses data for a single partition.

12.2.4 Lazy Update Anywhere

Allowing update transactions to execute at any replica and at the same time propagate changes only after commit combines flexibility with fast execution. No remote communication, neither between replicas nor between client and replica, is necessary.

Example Protocol

Figure 12.8 shows the differences to the lazy primary copy approach. Write operations can now be processed at all replicas and each replica is responsible to multicast the writesets of its local transactions to the other replicas (lines 1-6). When a replica receives such a remote writeset, it applies the changes (lines 7-14). However, as lazy update anywhere allows conflicting transactions to execute and commit concurrently at different replica without detecting conflicts during the life-time of transactions, conflict resolution might be needed. The system has to detect for a write operation on a data item x whether there was a concurrent conflicting operation on the same data item. If such a conflict is detected, conflict resolution has to ensure that the different replicas agree on the same final value for their data copies of x. There are many ways to resolve the conflict; a common choice is the Thomas Write Rule, which discards any update with earlier timestamp than a previously applied update.

Example Execution

Figure 12.9 shows an example execution under this protocol indicating the special case of a conflict. Both $T_1 = r_1(x), w_1(x, v1)$, local at $R1$ and $T_2 = r_2(y), w_2(x, v2)$,

```
Upon: $w_i(x, v_i)$ for local transaction $T_i$
 1: acquire exclusive lock on $x$
 2: if deadlock then
 3:     call $abortHandling(T_i)$
 4: else
 5:     $x := v_i$
 6:     return ok

Upon: receiving write set message of
      remote transaction $T_j$ from replica $R'$
 7: for all $w_j(x, v_j)$ in write set do
 8:     acquire exclusive lock on $x$
 9: for all $w_j(x, v_j)$ in write set do
10:     if Conflict detected then
11:         resolve conflict for eventual consistency
12:     else
13:         $x := v_j$
14: commit $T_j$, release locks
```

Fig. 12.8 Lazy Update Anywhere Algorithm.

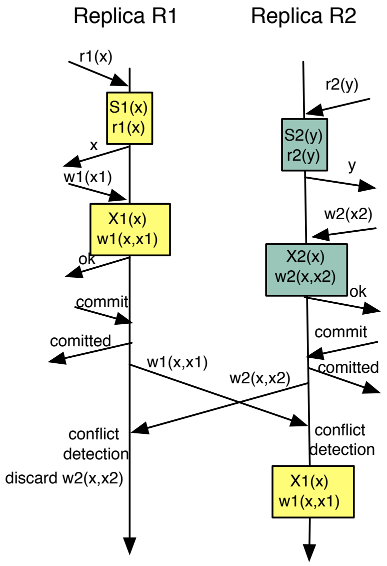

Fig. 12.9 Lazy Update Anywhere Example.

local at $R2$ update x and commit. After commit, the writesets are propagated. If both $R1$ and $R2$ simply applied the update they receive from the other replica, $R1$ would eventually have the value written by T_2, and $R2$ would have the value written by T_1. Upon receiving the writeset of T_2, $R1$ has to detect that T_1 was concurrent to T_2, conflicts with T_2, and already committed. $R2$ has to detect this conflict when receiving the writeset of T_1. A resolution mechanism, e.g., via timestamps, has led both replicas to decide that T_1's update wins out, and so $R1$ discards the write of x from T_2, while $R2$ overwrites x using T_1's update. In this way, it is guaranteed that the data copies converge towards a common value.

Discussion

Advantages Lazy update anywhere provides flexibility and fast execution for all transactions. These are two very strong properties. In some situations the approach is possible and necessary, e.g., if WANs have frequent connection loss, forbidding updates would lead to large revenue loss, and either conflicts are rare or easy to resolve.

Disadvantages One has to be aware that the fundamental properties of transactions are violated. Durability is not guaranteed as a transaction might commit from the user perspective but the updates are finally lost during conflict resolution. Atomicity might be lost, if conflict resolution is done on an per-object basis. If a transaction T_i updates data items x and y, and a conflict with T_j exists only on x, then it could be possible that T_i's update on x is not considered while its update on y succeeds.

The traditional concept of serializability or other isolation levels is also not longer guaranteed. Finally, conflict resolution is potentially very complex; depending on the semantics of data objects and the application architecture, different resolution mechanisms might be needed for different parts of the database. Therefore, such an approach is appropriate in very controlled environments with exact knowledge of the application. It is likely not suitable for a general replication solution.

12.2.5 Eager vs. Lazy

Since Gray et. al. [20] categorized protocols into eager and lazy many new replica protocols have been developed, and it is not always clear whether they are eager or lazy. In eager algorithms using a 2PC, the transaction at the local replica only commits once the transaction has been executed at all replicas. With this, all data copies are virtually consistent. This results in the disadvantage that the response time perceived by a client is determined by the slowest machine in the system.

Many recent protocols that also define themselves as eager do not run a 2PC. Instead, they allow a transaction to be committed at the local replica once the local replica knows that all remote replicas will "eventually" commit it (unless the remote replica fails). This typically requires that all replicas have received the write operations or the writeset, and that it is guaranteed that each replica will decide on the same global serialization order. It is not necessary that remote replicas have actually already executed the write operations at the time the local replica commits. This means, some "agreement" protocol is executed among the replicas but it does not necessarily include processing the transaction or writing logs to disk. We use this weaker form of "eagerness" in order to accommodate many of the more recent replication solutions.

Many of these approaches use the delivery guarantees of group communication systems [13] to simplify the agreement protocol. Group communication systems provide multicast primitives with ordering guarantees (e.g., FIFO order or total order of all messages) and delivery guarantees (see Chapter 3). In particular, a multicast with uniform reliable delivery guarantees that whenever a message is delivered to any replica, it is guaranteed that it will be delivered to all replicas that are available. Now assume that transactions multicast their writesets using uniform reliable delivery. In case a replica receives a writeset of a local transaction and commits the transaction, uniform reliable delivery guarantees the writeset will be received by all other replicas, even if the local replica crashes immediately after the commit. Therefore, assuming an appropriate concurrency control method, the transaction will also commit at all other replicas. This provides the "eager" character of these protocols offering atomicity and durability without the need of a 2PC. Uniform reliable delivery itself performs some coordination among the group members before delivering a message to guarantee atomicity in the message delivery, though this is not visible to the replication algorithm. There are some subtle differences in the properties provided by uniform reliable delivery multicast compared to 2PC as the first assumes a crash-stop failure model where nodes never recover, while 2PC has a crash-recovery model that assumes sites to rejoin.

In contrast, we use the term lazy for protocols where write operations are not sent at all before commit time, or where the sending multicast occurs earlier but is not reliable. Thus, if a local replica fails after committing a transaction but before propagating its write operations successfully, then the remote replicas have no means to commit the transaction. If the others don't want to block until the local replica has recovered and sent the write operations, the transaction can be considered lost.

12.3 Correctness Criteria

So far, we have only informally reasoned about the differences in correctness that the protocols provide. In fact, the research literature does not have a single, generally agreed on understanding of what "correctness" and data consistency mean. Terms such as strong consistency, weak consistency, 1-copy-equivalence, serializability, and snapshot isolation are used but definitions vary and it is not always clear which failure assumptions are needed for a protocol to provide its properties (see Chapter 1 for consistency models for replicated data). In this section, we discuss different aspects of correctness, and how they relate to one another.

In our discussion above, the eager protocols provided a stronger level of consistency than the lazy ones. This is, however, only true in regard to atomicity. But there does not simply exist a stack of consistency levels, from a very low level to a very high level. Instead, correctness is composed of different orthogonal issues, and a replica control protocol might provide a high level of consistency in one dimension and a low level for another dimension. In regard to lazy vs. eager, all lazy protocols are weaker than eager protocols in regard to atomicity. But given two particular protocols, one lazy and one eager, the lazy protocol could provide stronger consistency than the eager protocol in regard to a different correctness dimension.

In the following, we look at several correctness dimensions individually, extract what are the possible levels of consistency for this dimension and discuss to what degree replication protocols can fulfill the criteria depending on their category.

12.3.1 Atomicity and Consistency

Atomicity in a replicated environment means that if an update transaction commits at one replica it has to commit at all other replicas and its updates are executed at all replicas. If a transaction aborts, none of the replicas may have the updates reflected in its database copy. Considering a failure-free environment only, this means that the replica and concurrency control system has to ensure that each replica takes the same decision on commit/abort for each transaction. For instance, in the primary copy protocols described in the previous section, this was achieved via a FIFO multicast of write operations or writesets and strict 2PL.

Considering a system that is able to tolerate failures, atomicity means that if a transaction commits at a replica that fails after the commit, the remaining available replicas also need to commit the transaction in order for the transaction to not be "lost". Note that available replicas can continue committing transactions while some replicas are down and thus, don't commit these transactions. Recovery has to ensure that a restarted replica receives the missing updates. In summary, available replicas commit the same set of transactions and failed replicas have committed a subset of these transactions.

Atomicity in the presence of failures can only be achieved by eager protocols as all replicas are guaranteed to receive the writeset information and all other information needed to decide on the fate of a transaction before the local replica commits the transaction. Thus, the available replicas will eventually commit the transaction. In a lazy protocol, available replicas might not know about the existence of a transaction that executed at a replica that fails shortly afterwards. In a lazy primary copy protocol, atomicity can be guaranteed if no new primary is chosen when the current primary fails. Then, upon recovery of the primary, the missing writesets can eventually be propagated. However, this would severely reduce the availability of the system. If a failover takes place to a new primary, or in a lazy update anywhere approach, one can still attempt to send the writesets after recovery. However, the transaction might conflict with other transactions that committed in the meantime, and therefore, no smooth integration of the "lost" transaction into the transaction history is possible.

In summary, while eager protocols can provide atomicity, lazy approaches can be considered non-atomic.

In the research literature, one often finds the term *strong consistency* associated with eager protocols and *weak consistency* associated with lazy protocols. The use of these terms usually remains vague. One way to define strong consistency is by what we have called virtual consistency, requiring all data copies to have the same value at transaction commit time. Only eager protocols with a 2PC or similar agreement provide virtual consistency; the weaker forms of eagerness described in Section 12.2.5 allow a transaction to commit before all replicas have executed the write operations. Nevertheless, protocols based on this weaker eager definition are usually also associated with providing strong consistency. Strong consistency is different than atomicity as it refers to the values of data items and not the outcome of transactions. It typically implies that all replicas apply conflicting updates in the same order. In theory, one might have a replica control protocol that provides atomicity (guaranteeing that all replicas commit the same set of transactions) but the execution order of conflicting transactions is different at the different replicas. However, we are not aware of such a protocol.

Weak consistency generally means that data copies can be stale or even temporarily inconsistent. Staleness arises in lazy primary copy approaches. As long as the primary has not propagated the writeset to the secondaries, the data copies at the secondaries are outdated. If the secondaries apply updates in the same serialization order as they primary, the data copies at secondaries do not contain any incorrect data but simply data from the past. A system can be designed to limit the staleness experienced by a read operation on a secondary site. For instance, for numeric values the difference between the value read and the value at the primary might be kept below a threshold like 100. Other systems use different forms of limiting the divergence, for example, the secondary copy might be required to have missed no more than a fixed number of writes which were already applied at the primary, or the limit might be on the time between when a write is done at the primary and when it gets to the secondary. These staleness levels, also referred to as freshness levels allow one to bound the discrepancy between replicas visible to the outside world. These intermediate consistency levels can be achieved by refreshing secondary copies at appropriate time points.

Lazy update anywhere protocols allow the copies to be inconsistent. As our example in Figure 12.9 has shown, each replica might have changes of a local committed transaction while missing conflicting changes from a concurrent transaction committed at a different replica. In such a scenario, the most important property to provide is *eventual consistency* [42]. It indicates that, assuming the system reaches a quiescent state without any further write operations, all copies of a data item eventually converge to the same value. Note that eventual consistency is normally defined outside the scope of transactions. As such, it is possible that if two conflicting transactions T_i and T_j update both x and y, all copies of x will eventually contain T_i's update while all copies of y will have T_j's update. One way to define eventual consistency in the context of transactions is as follows: there must exist a subset of the committed transactions and an order on this subset, such that data copies converge to the same values as if the write operations of these transactions had been executed in the given serial order.

12.3.2 Isolation

Isolation in a Non-replicated System

In non-replicated database systems, the level of isolation indicates the degree to which concurrently executing transactions are allowed to be seen by another one. The most well-known correctness criteria is *serializability*: the interleaved execution of transactions is equivalent to a serial execution of these transactions. Typically, two executions are considered equivalent if the order of any two conflicting operations is the same in both executions. Two operations conflict if they access the same data item and at least one is an update operation. The most well-known concurrency control mechanisms providing serializability are strict 2-phase-locking and optimistic concurrency control. Weaker levels of isolation are often defined by specifying a set of anomalies that are allowed to occur during the execution. For instance, *snapshot isolation* allows an anomaly that may not occur in a serializable execution[1]. Snapshot isolation can be implemented very efficiently and provides much better concurrency in applications with a large read proportion. Transactions read from a snapshot of the database that represents the committed version of the database as of start of transaction. Conflicts only exist between write operations. If two concurrent transactions want to update the same data item only one of them may

[1] Note that strictly speaking snapshot isolation and serializability are incomparable since snapshot isolation disallows some executions allowed by serializability (concurrent blind writes, e.g. consider two transactions, $r_1(x); r_2(y); w_1(y); w_2(x); c_1; c_2$, being the subscripts the transaction identifier) and vice versa (write skew: $r_1(x); r_2(x); w_1(y); w_2(y); c_1; c_2$).

succeed, the other has to abort. Snapshot isolation typically uses multiple versions to provide snapshots.

Global Isolation Levels

Ideally, a replicated system should provide exactly the same level of isolation as a non-replicated system. For that, definitions for isolation in a replicated system have to reduce the execution over data copies onto an execution over a single logical copy. For instance, serializability in a replicated system is provided if the execution is equivalent to a serial execution over a single logical copy of the database.

Apart from serializability, snapshot isolation has also been well studied in replicated systems. All transactions must read from snapshots that can also exist in a non-replicated system and writes by concurrent committed transactions must not conflict, even if they are executed at different replicas. In a replicated environment, snapshot isolation is very attractive due to its handling of read operations.

Atomicity vs. Isolation

In principle, isolation is orthogonal to atomicity. Both eager and lazy protocols can provide serializability or snapshot isolation across the entire system. However, this only holds if there are no failures. If there are failures, then the problem of lost transactions occurs in lazy protocols, as we have discussed before. It is not clear how these lost transactions and transactions that have read values written by these lost transactions, can be placed in the execution history to show that it is equivalent to a serial history or fulfills the snapshot isolation properties.

1-Copy-Equivalence

1-copy-equivalence requires the many physical copies to appear as one logical copy. It was introduced with failures in mind, that is, the equivalence must exist even when copies are temporarily not available; in this view, lazy protocols do not provide 1-copy-equivalence. 1-copy-equivalence can then be combined with an isolation level to consider isolation in a failure-prone environment. For example, 1-copy-serializability requires the execution over a set of physical copies, some of them possibly unavailable, to be equivalent to a serial execution over a single logical copy.

Linearizability and Sequential Consistency

Linearizability and sequential consistency are two correctness criteria defined for the concurrent execution on replicated objects. They include the notion of the execution over the replicated data to be equivalent to an execution on a single image of the object. However, none of the two has the concept of transactions which requires to take operations on different objects into account (although sequential consistency takes the order within a client program into account). Different to serializability and snapshot isolation, linearizability requires an order that is consistent with real time.

12.3.3 Session Consistency

Session consistency is yet another dimension of correctness that is orthogonal to atomicity, data consistency or isolation. It defines correctness from the perspective of a user. Users typically interact with the system in form of sessions. For instance, a database application opens a connection to the database and then submits a sequence of transactions. These transactions build a logical order from the user's perspective. Therefore, if a client first submits transaction T_i and then T_j, and T_i has written some data item x that T_j reads, then T_j should observe T_i's write (unless another transaction has overwritten x since T_i's commit). This means, informally, session consistency guarantees that a client observes its own writes.

Definitions like serializability and 1-copy-serializability do not include session consistency, since they require the execution to be equivalent to a serial order, but that may not match the order of submission within a session. In the usual non-replicated platforms, built with locking or SI, session consistency is observed. Thus a truly transparent replicated system should provide session consistency, too.

In a replicated system, without special mechanisms, replica control may not ensure session consistency For instance, in a lazy primary copy approach, the client could submit an update transaction to the primary, and then submit a read-only transaction to a secondary before the writeset of its update transaction has been propagated to the secondary. In this case, it does not observe its own writes. In order to provide session consistency, such a protocol needs to be extended. For instance, transactions can receive global transaction identifiers which are monotonically increasing within a session. The driver software at the client then keeps track of the transaction identifiers. Whenever it submits a new transaction to a replica it piggybacks the identifier of the last transaction that was committed on behalf of this client. Then, the replica to which the new transaction was submitted will make sure that the new transaction will see any state changed performed by this last or older transactions.

Other protocols provide session consistency automatically, e.g., an eager protocol with 2PL and 2PC. Assume again a primary copy approach and a client submits first update transaction T_i to the primary and then read-only transaction T_j to a secondary. Although T_i might not yet be committed at the secondary when the first operation of T_j is submitted, T_i is guaranteed to be in the prepared state or a later state holding all necessary locks. Thus, T_j will be blocked until T_i commits and will see its writes. Eager protocols that only guarantee "eventual commit" typically need a special extension, e.g., a special driver as described above, to provide session consistency.

12.4 Other Parameters

We have already seen that eager protocols do not necessarily always provide higher guarantees than lazy protocols. In the same way, lazy protocols do not always perform better than eager protocols. In fact, performance depends on many issues. Some fundamental techniques can be applied to most replica control algorithms to speed-up processing. In this section, we discuss some of them. We also discuss some other fundamental design choices for a replicated database architecture that have a great influence on the performance, feasibility, and flexibility of the replication solution.

12.4.1 Message Management

The number of message rounds within a transaction are an important parameter for a replica control protocol. Looking at our examples of Section 12.2, the eager update anywhere protocol has a message round per write operation of a transaction (writeset and acknowledgement) plus the 2PC. With this, the number of messages within a transaction is linear with the number of write operations of the transaction. In contrast, the presented lazy protocols send one message per transaction, independently of the number of operations.

The number of messages per transaction depends on protocol details rather than simply on the category. For example, eager protocols can have a constant number of messages and lazy protocols can send a message per write operation. As an example, let's have a look at two further eager update anywhere protocols. The first alternative (*Alternative 1*) to the protocol presented in Figure 12.4 (*Original protocol*) executes first all operations only on the local database copy. Only when the client submits the commit request, the local replica sends the writeset with all write operations to all other replicas. The other replicas acquire the locks, execute the operations and return when they have completed. Finally the 2PC is performed. This model has one message round for the writeset and acknowledgements plus the overhead for the 2PC. The second alternative (*Alternative 2*) also executes the transaction first locally and sends the writeset at commit time. The remote replicas acquire the locks and send the acknowledgement once they have all locks. The local replica commits the transaction once it has received all acknowledgements. No 2PC takes place. The remote replicas execute the write operations in the writeset and commit the transaction in the meantime. That is, transaction execution contains only a single message round.

It is often assumed that transaction response time increases with the number of message rounds which occur during the transaction. In WANs, where messages take a long time, this means it is usually unacceptable to include more than one message round. In LANs, however, message latency might not play such a big role, and message throughput is often high. In such an environment, response time may be influenced more by other aspects rather than rounds of message exchange.

We illustrate this along the eager update anywhere protocol of Figure 12.4 and the two alternatives presented above. Figure 12.10 shows an example execution of a transaction $T_1 = w_1(x, v1), w_1(y, w1)$ updating x and y under these three variant protocols. In this diagram, we show time by the vertical distance, and we pay special attention to the possible concurrency between activities. The original protocol of Figure 12.4 multicasts each write operation and then executes it locally. That is, in the ideal case, the write operations on the different physical copies occur concurrently, and the local replica receives all acknowledgements shortly after it has

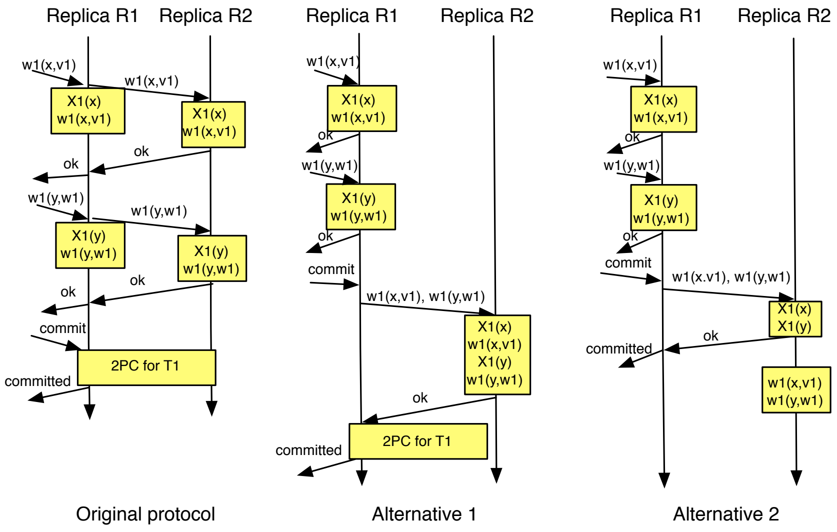

Fig. 12.10 Example execution with three different eager update anywhere protocols.

completed the operation itself. As such, execution is concurrent (this execution is also called conservative in Chapter 13). What is added is the latency of *n* messages rounds if there are *n* write operations and the latency of the 2PC. In the first alternative described above, the local replica first executes locally, then sends one message, then the remote replicas execute the write operations and then the 2PC occurs. Thus, while the number of message rounds is lower, the pure execution time is actually longer than in the original protocol as execution at the local replica and remote replicas is not performed in parallel. Finally, the last algorithm has the local execution, then one writeset message, then the time to acquire the locks successfully and finally the acknowledgement phase within the response time of the transaction. This approach has the lowest number of messages rounds and the actual execution time at the remote replicas is not included in the response time. These two alternatives are executed optimistically at the local replica (called optimistic execution in Chapter 13).

12.4.2 Executing Writes

Write operations have to be executed at all replicas. This can be done in two ways. In *statement* replication, also called *symmetric* replication, each replica executes the complete write operation, e.g., the SQL statement (update, delete, insert). In contrast, in *object* replication, also called *asymmetric* replication, only the local replica executes the operation and keeps track of the tuples changed by the operation. Then, the changed tuples are sent to the remote replicas which only apply the changes.

Applying the changes has usually much less overhead than executing the statement itself. For instance, experiments with PostgreSQL have shown that even for simple statements (update on primary key), applying the change takes only 30% of the resources compared to executing the entire statement. The reasons are the cost in parsing the SQL statement, building the execution tree, etc. However, if a statement changes many data records, then sending and processing them might be costly because of message size. In this case, statement replication is likely to be preferable.

A challenge of statement replication is determinism. One has to make sure that executing the statement has the same results at all replicas. If statements include setting the current time, generating a random number, etc. determinism is no more given.

An extreme case of statement replication would actually not only execute the write operations of an update transaction at all replicas, but the entire update transaction. This might be appropriate in WANs in order to keep the message overhead low.

12.4.3 Concurrency Control Mechanisms

Our example protocols so far all used standard strict 2PL as concurrency control mechanism. Clearly, replica control can be combined with various concurrency control mechanisms, not only 2PL. In this section, we look at optimistic and multi-version concurrency control.

Concurrency Control in a Non-replicated System With optimistic concurrency control, a transaction's writes are done in a private workspace, and then, at the end of the transaction, a validation phase checks for conflicts, and if none are found, then the private workspace is written into the shared database. One mechanism is backward validation, where the validation of transaction T checks whether there was any concurrent transaction that already performed validation and wrote a data item that was read by T. If this is the case, T has to abort.

Multi-version concurrency control is used in connection with snapshot isolation. Each write operation generates a new version of a data item. We say a data version commits when the transaction that created the version commits. Versions can be used to provide read operations with a snapshot. The read operation of a transaction T_i reads the version of a data item that was the last to commit before T_i started. With this, a transaction reads from a snapshot as of transaction start time. Snapshot isolation has to abort one of two concurrent transactions that want to update the same data item. Commercial systems set write locks to detect conflicts when they occur and abort immediately. However, conflicts can also be detected at commit time similar to the mechanisms for optimistic concurrency control.

Concurrency Control in a Replicated System The challenge of distributed concurrency control is to ensure that all replicas decide on the same serialization order. Primary copy approaches can simply rely on the (non-replicated) concurrency control mechanism at the primary and then forward write operations or writesets in FIFO order. The concurrency control tasks at secondaries are then quite straightforward. The extensions for an update anywhere approach are often more complicated.

In the particular case of strict 2PL, no extensions to the protocol itself are needed. However, distributed deadlock can occur. For optimistic and snapshot isolation concurrency control the question arises how to perform validation. Validation of all transactions could be performed at one central site. Alternatively, each replica could perform validation. However, in the latter case, the validation process needs to be deterministic to make sure that all replicas validate transactions in the same order and decide on the same outcome. For that purpose, many replication approaches use a total order multicast to send the relevant validation information to all replicas. Total order multicast is provided by group communication systems [13]. It guarantees that all members of a group receive all messages sent to the group in the same total order.

Optimistic Concurrency Control Figure 12.11 sketches a replica control protocol based on optimistic concurrency control and a central scheduler that performs validation for all transactions. A transaction is submitted to any replica and executed locally according to standard optimistic techniques. A read operation (lines 1-2) accesses the last committed version of the data item. Data items are tagged with the transaction that was the last to write them. A transaction keeps track of all data versions read in the read set RS. A write (lines 3-5) creates a local copy which is added to the transaction's writeset WS. An abort (lines 6-7) simply means to discard both read and writeset. Upon a commit request, the read and writesets are sent to the scheduler (line 8) which performs validation (lines 9-12). It checks whether the readset of the currently validated transaction overlaps with the writesets of any concurrent transaction that validated before. If yes, it tells the local replica to abort the transaction. Otherwise it forwards the writeset to all replicas using a FIFO multicast. The replicas apply them (lines 13-20). A write $w(x)$ of this transaction becomes now the last committed version of x (line 17). Validation and applying the writeset is performed in the same serial order. The protocol description hides several technical challenges when such an approach should really be implemented in a database system. Firstly, one has to determine whether two transactions are concurrent. For that some timestamp mechanism must to be used, which can compare transactions that are local at different replicas.

Snapshot Isolation Figure 12.12 sketches a replica control protocol based on snapshot isolation and using total order multicast. A transaction executes locally (lines 1-4) reading the last committed snapshot as of start time and creating new versions upon write operations. Abort simply means to discard the writes (lines 5-6). At the end of transaction only the writeset is multicast in total order (line 7). Validation now checks whether this writeset overlaps with the writesets of any concurrent transaction that validated before (line 8). No information about reads needs to be sent, since in SI conflict, leading to abort (lines 9-11), is only considered between write operations. If validation succeeds, remote transactions have to create the new versions (lines 13-16). Transactions are committed serially to guarantee that all replicas go through the same sequence of snapshots. The advantage over the optimistic concurrency control protocol is that read operations remain completely

```
Upon: $r_i(x)$ for local transaction $T_i$   {let $T_j$ be the last to update $x$ and commit}
 1: add $x^j$ to read set $RS_i$
 2: return $x^j$

Upon: $w_i(x,v_i)$ for local transaction $T_i$
 3: create local copy $x^i$ of $x$ and add to write set $WS_i$
 4: $x^i := v_i$
 5: return ok

Upon: abort request for local transaction $T_i$
 6: discard $RS_i$ and $WS_i$
 7: return abort

Upon: commit request for local transaction $T_i$
 8: send $(RS_i, WS_i)$ to central scheduler

Upon: receiving $(RS_i, WS_i)$ from replica $R$ {validation at central scheduler}
 9: if $\exists T_j, T_j || T_i \wedge WS_j \cap RS_i$ then
10:     send $abort(T_i)$ back to $R$
11: else
12:     multicast $WS_i$ to all replicas in FIFO order

Upon: receiving $WS_i$ for any transaction $T_i$ from central scheduler in FIFO order
13: for all $w_i(x,v_i)$ in $WS_j$ do
14:     if $T_i$ remote transaction then
15:         create local copy $x^i$ of $x$
16:         $x^i := v_i$
17:     write $x^i$ to database
18: commit $T_i$
19: if $T_i$ local transaction then
20:     return ok

Upon: receiving $abort(T_i)$ for local transaction from central scheduler
21: discard $RS_i$ and $WS_i$
22: return abort
```

Fig. 12.11 Update Anywhere Protocol based on Optimistic Concurrency Control and Central Scheduler.

```
Upon: $r_i(x)$ for local transaction $T_i$
 1: return committed version $x^j$ of $x$ as of start time of $T_i$

Upon: $w_i(x,v_i)$ for local transaction $T_i$
 2: create version $x^i$ of $x$ and add to write set $WS_i$
 3: $x^i := v_i$
 4: return ok

Upon: abort request for local transaction $T_i$
 5: discard $WS_i$
 6: return abort

Upon: commit request for local transaction $T_i$
 7: multicast $WS_i$ to all replicas in total order

Upon: receiving $WS_i$ for any transaction $T_i$ in total order
 8: if $\exists T_j, T_j || T_i \wedge WS_j \cap WS_i$ then
 9:     discard $WS_i$
10:     if $T_i$ local transaction then
11:         return abort
12: else
13:     if $T_i$ remote transaction then
14:         for all $w_i(x,v_i)$ in $WS_i$ do
15:             create version $x^i$ of $x$
16:             $x^i := v_i$
17:     commit $T_i$
18:     if $T_i$ local transaction then
19:         return ok
```

Fig. 12.12 Update Anywhere Protocol based on Snapshot Isolation and Total Order Multicast.

local. The local replica makes sure that all reads are from a committed snapshot. For validation they don't play any role.

Fault-Tolerance Both the optimistic and the pessimistic protocol above use multicast primitives. If the multicast primitive provides uniform reliable delivery, then we can consider these protocols as eager: a transaction only commits locally when it is guaranteed that the writeset will be delivered at all replicas and when the global serialization order of the transaction is determined. Therefore, when a transaction

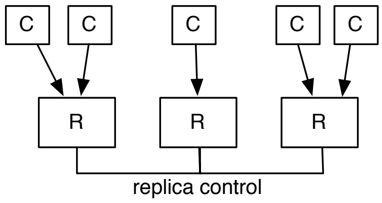

Fig. 12.13 Kernel-based Architecture.

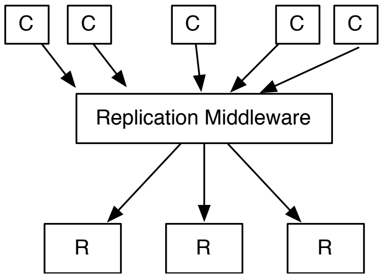

Fig. 12.14 Central Middleware.

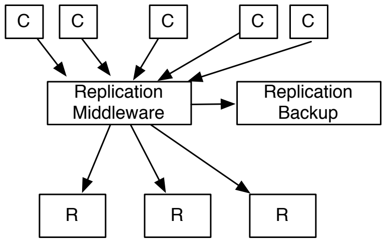

Fig. 12.15 Central Middleware with Backup.

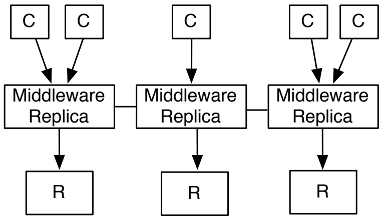

Fig. 12.16 Middleware Replica for each Database Replica.

commits locally it will commit in the same order at all other available replicas. If the above protocols use a multicast without uniform reliable delivery, they have the characteristics of a lazy protocol: a transaction might be committed at a replica that fails and the other replicas do not receive the writeset.

12.4.4 Architectural Alternatives

There exist two major architectural design alternatives to implement a replication tool. Our description of protocols so far followed a *kernel-based* or *white box* approach, where the replica control module is part of the database kernel and tightly coupled with the existing concurrency control module. A client connects to any database replica which then coordinates with the other replicas. The database system is replication-aware. Figure 12.13 depicts this architecture type.

Alternatively, replica control can be implemented outside the database as a *middleware* layer. Clients connect to the middleware that appears as a database system. The middleware then controls the execution and directs the read and write operations to the individual database replicas. Some solutions work with a purely *black-box* approach where the underlying database systems that store the database replicas do not have any extra functionality. Others use a *gray-box* approach where the database system is expected to export some minimal functionality that can be used by the middleware for a more efficient implementation of replica control. For instance, the database system could collect the writeset of a transaction in form of the set of records the transaction changed and provide it to the middleware on request. A

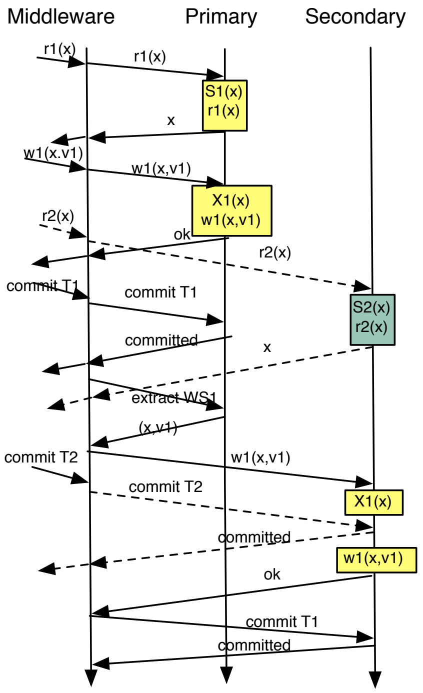

Fig. 12.17 Execution of a lazy primary-copy protocol with a central middleware.

middleware-based approach has typically its own concurrency control mechanism which might partially depend on the concurrency control of the underlying database systems. There might be a single middleware component (centralized approach) as in Figure 12.14, or the middleware might be replicated itself. For example, the middleware could have a backup replica for fault-tolerance (Figure 12.15). Other approaches have one middleware instance per database replica, and both together build a replication unit (Figure 12.16). A transaction can then connect to any middleware replica.

Figure 12.17 depicts an example execution of a lazy, primary copy protocol based on a central middleware. There is an update transaction T_1 and a read-only transaction T_2. All requests are sent to the middleware. T_1 (solid lines) is executed and committed at the primary. T_2 (dashed lines) is executed at any replica (in the example the secondary). After T_1 commits, the middleware extracts the writeset, and executes the write operations at the secondaries. The proper order of execution of these write operations and the local concurrency control mechanisms at the database replicas (here strict 2PL) guarantees the same serialization order at all replicas.

Discussion Kernel-based approaches have the advantage that they have full access to the internals of the database. Thus, replica control can be tightly coupled with concurrency control and it is easy to provide concurrency control at the record level across the entire replicated system. Writeset extraction and application can be made highly efficient. In contrast, middleware-based protocols often have to partially re-implement concurrency control. Before execution of a particular statement, they often only have partial information of which records are exactly accessed (as SQL statements can contain predicates). This results often in a coarser concurrency level (e.g, table-based). If the database system does not export writeset functionality, writesets need to be extracted via triggers or similar procedures, which often is much less efficient than a kernel-based extraction. Fault-tolerance of the middleware is a further major issue. Depending on how much state the middleware manages, it can be complicated. Finally, the middleware represents a level of indirection, increasing the total number of messages.

However, middleware-based systems have many advantages. They can be used with 3rd-party database systems that do not provide replication functionality and whose source-code is not available. Furthermore, they can potentially be used in heterogeneous environments with different database systems. They also present a nicer separation of concerns. In a kernel-based approach, any changes to the concurrency control implementation might directly affect the replica control module. For middleware-based approaches this is likely only the case with major changes in the functionality of the underlying system.

12.4.5 Cluster vs. WAN Replication

We have already mentioned in the introduction that replication is done for different purposes, and in different settings. When replication is used for scalability, then the replicas are typically located in a single cluster. Read access to data items can then be distributed across the existing replicas while write operations have to be performed on all replicas. If the read ratio is high, then an increasing load can be handled by adding more replicas to the system. In a LAN, message latency is low and bandwidth high. Thus, the number of messages does not play a major role. Hence, a transaction can likely have several message rounds and a middleware can be interposed without affecting performance too much. Furthermore, there is no need for lazy update anywhere replication as the gain in efficiency is not worth the low degree of consistency that it provides. Finally, the write overhead should be kept as low as possible as it determines how much the system can be scaled. Thus, asymmetric replication will be better than symmetric replication.

Often, replication serves the purpose of fast local access when the application is geographically distributed. In this case, replicas are connected via a WAN. Thus, message latency plays an important role. In this context, lazy update anywhere might be preferable as it does not have any message exchange within the response time of the transaction. The price for that, namely conflict resolution and temporary inconsistency, might be acceptable. It might also be possible to split the application into partitions and put a primary copy of each partition close to the clients that are most likely to access it. This would provide short response times for most transactions without inconsistencies. One has to make sure that clients don't have to send several rounds of messages to a remote replica or a remote middleware. They always should be able to interact with their local site or send transaction requests in a single message to remote sites. The influence of symmetric vs. asymmetric replication will likely play a minor role in a WAN setting. The larger message sizes of asymmetric replication might be of disadvantage.

Replication for fault-tolerance can deploy replicas both in a LAN and a WAN. In LANs, typically eager protocols are used to keep replicas consistent. When a replica fails, another replica can take over the tasks assigned to the failed replicas in a transparent manner. Replicas can also be distributed across a WAN, typically with lazy propagation. When a catastrophic failure occurs that shuts down an entire location, a replica in a different location can take over. As catastrophic failures seldom occur, weaker consistency is acceptable for the advantage of having better performance during normal processing.

12.4.6 Degree of Replication

So far, we have assumed that all replicas have a full copy of the database, referred to as full replication. In *partial* replication, each data item of the database has a physical copy only on a subset of sites. The replication degree of a data item is the number of physical copies it has (see Chapter 5 for partial replication).

Partial replication serves different purposes. We mentioned above that for cluster replication, the ratio of write operation presents a scalability limit. Ideally, if a single system can handle C transactions per time unit, than a n-node system should be able to handle nC transactions. However, write operations have to be executed at all replicas. Thus, if write operations constitute a fixed fraction of the submitted workload, increasing the workload means increasing the absolute number of write operations each replica has to perform. This decreases the capacity that is available to execute local read operations. At some level of load, adding new sites will not increase the capacity available for further operations, and thus, throughput cannot be increased beyond this point. In the extreme case with 100% write operations and symmetric replication a replicated system does not provide any scalability as it can handle C transactions just as the non-replicated system.

When using partial replication in a cluster environment, read operations are executed at a single replica, as in full replication. Write operations now only need to be executed at replicas that have a copy of the data item accessed. For instance, if each data item has only two copies, then only two write operations need to be performed. Assuming again 100% write operations and symmetric replication, n replicas can handle $nC/2$ transactions (assuming data copies are distributed and accessed uniformly). With less write operations, it scales appropriately better. The important point is that when the replication degree is fixed to a constant (e.g., 2 or 4), then the system can scale without facing a limit from contention for writing. In contrast, if the replication degree increases with the number of sites in the system (e.g., n, $n/2$), then there is a scalability limit.

In a WAN environment, having a data item replicated at a specific geographic location decreases communication costs for read operations but increases communication and processing costs for update transactions. In this context, the challenge is to place data copies in such a way to find a trade-off between the different factors.

Partial replication has several challenges. Finding an appropriate replication degree and the optimal location for the replicas is difficult. Concurrency control has to be adjusted. When a client accesses a data item, a replica needs to be located. This is not necessarily the local replica. Also, partial replication might lead to distributed queries if no site has data copies of all data items accessed by the query.

12.4.7 Recovery

Recovering failed replicas and letting new replicas join the system is an important task. A joining replica has to get the up-to-date state of the database. This can be done by transferring a copy of the entire database to the joining replica. For a replica that had failed and now rejoins, it is also possible to only receive fresh copies of the data items that were actually changed during the downtime. Alternatively, it could receive and apply the writesets of the transactions it has missed during its downtime. This state transfer can take place offline or online. With *offline* transfer, transaction processing is interrupted until the transfer is completed. Using *online* recovery, transaction execution continues during state transfer. In this case, one has to make sure that the recovering replica does not miss any transactions. For a given transaction, either its changes are contained in the state transfer, or the joining replica receives its updates after the transfer is complete.

12.5 Existing Systems

12.5.1 Early Work

Database replication had its first boom in the early 80s. The book "Concurrency Control and Recovery in Database Systems" [7] provides a formalism to reason about correctness in replicated database systems. The term 1-copy-serializability was created and is still used today. Early work on replication took as baseline concurrency control mechanisms of non-replicated systems, extended them and combined them with replica control [7, 9]. Failure handling – both site and network failures – were a major research issue [6, 1]. Basically all these approaches used eager replication and provided strong correctness properties in terms of atomicity and isolation. In 1996, Gray et al. [20] indicated that these traditional approaches provide poor performance and do not scale as they commit transactions only if they have executed all their operations on all (available) physical data copies. Also, execution is often serially, leading to extremely long response times.

12.5.2 Commercial Systems

Since then, many new replication solutions have been developed. Commercial systems often provide a choice of replication solutions. High-availability solutions often implement a simplified version of primary-copy. In these approaches, all transactions (update and read-only) are submitted to the primary. The secondary only serves as a backup. Writeset propagation to the backups can be eager or lazy. In case the primary fails, clients are automatically redirected to the backup which becomes the new primary. Typically, any active transaction is aborted. Otherwise, clients can continue their requests as if no failure had occurred.

Lazy replication solutions, which allow looser consistency when reading at a replica, are often provided for WAN replication. Sophisticated reconciliation techniques are offered for update anywhere, based on timestamps, site priority, values or arithmetic functions. Both distributed and centralized reconciliation mechanisms exist. Eager update anywhere protocols are rarely found in commercial systems.

12.5.3 Lazy Replication Made Serializable

Some research efforts analyzed the correctness of lazy primary copy protocols where different data items have their primary copies on different sites [14]. In such a scenario, global serializability can be violated even if each site implements strict 2PL. In order to avoid incorrect executions some solutions restrict the placement of primary and secondary copies to avoid irregularities [8, 34]. The main idea is to define the set of allowed configurations using graphs where nodes are the sites and there are edges between sites if one site has a primary copy and the other a secondary copy of a given data item. Serializability can be provided if the graph has certain properties (e.g., it is acyclic). Others require to propagate updates along certain paths in the graph.

12.5.4 Cluster Replication

Group Communication Work in this direction started with approaches that explore the use of group communication and was based on kernel-replication (such as Postgres-R [25, 47] or the state machine approach [37]). Different tasks, such as transaction execution and data storage, can further be distributed [16]. Many other followed, e.g., [2, 22, 23, 26, 47, 27]. They provide different concurrency control mechanisms, differ in the interface they provide to the clients of the database system (JDBC interface vs. procedural interface), the way they interact with the group communication system, etc. They also consider recovery and failover mechanisms.

Middleware-Based Systems A lot of work has designed replication protocols that are especially targeted for middleware-based replication. There exist several approaches based on group communication [35, 10, 29, 36]. They often assume one middleware replica for each database replica and middleware replicas communicate with each other via multicast. They are typically all eager protocols.

Other solutions have a single middleware, possibly with a backup [3, 39]. Both eager and lazy approaches have been proposed. There is also considerable work that focuses less on the replica control itself but on issues such as load distribution and query routing [41, 32, 18, 4, 17, 48]. In lazy approaches one has a wide range of options when to actually propagate updates to other replicas, e.g., only when the freshness level goes below a threshold acceptable for queries. Load can be distributed according to many different strategies. Dynamically deciding on the number of replicas needed to handle a certain load has also been considered [19].

In [11], the authors provide an interesting discussion of the gap between the replica control protocols proposed by the research community, and the technical challenges to make them work in an industrial setting.

12.5.5 Other Issues

Approaches such as [40, 46, 28, 30, 44] take the specifics of WAN replication into account. They attempt to keep the number of message rounds low or accept weaker levels of consistency. Many approaches touch on partial replication, such as [17, 46, 45, 43, 44]. The particular issue of session consistency is discussed in [15].

12.5.6 Related Areas of Research

Many other techniques widely used in database systems can actually be considered some form of replication although they are not identified as such by research. In regard to scalability, materialized views are internal replicas of data that have been reorganized and processed in such a way so as to speed up certain queries that no longer need to be processed but can be answered directly from the materialized view. Materialized views can be seen as a special form of lazy primary copy replication (in some cases even update anywhere replication), where a materialized view is not a copy of a specific data item but an aggregation over many data items (i.e, table records) of the database. Thus, this makes change propagation considerably more complex [21] .

Parallel databases use both partitioning and redundant data allocation across disks and memory. Replica control algorithms look somewhat different as there is not an independent database engine at each node but the system is treated as a single logical unit (regardless of whether the hardware is intrinsically parallel such as a multi core processor or it is an actual cluster of machines).

In regard to fault-tolerance, the log of a database is a form of replication [33]. All changes to the database are replicated onto stable storage in form of redo and undo logs. When a server fails, a new server instance is started, reading the log in order to recreate the state of the database. Fault tolerance is also achieved through redundant hardware and RAID disks [12] which provide replication at a lower level.

Database caching has been explored extensively for performance improvements [31, 5, 38]. The database cache usually resides outside the database system and caches the most frequently used data items. It is used for fast query execution while updates typically go directly to the database backend. Consistency mechanisms are in place but often involve discarding outdated copies.

12.6 Conclusions

This chapter provides a systematic overview of replica control mechanisms as they occur in replicated databases. We started with a two-parameter characterization providing example protocols based on 2-phase-locking for each of the categories that help to understand the trade-offs between the different categories. Furthermore, we provided an overview of correctness criteria that are important in the context of database replication. Finally, we discuss several other parameters of the replica control design space such as the number of message rounds, writeset processing, concurrency control mechanism, the replication architecture and the degree of replication assumed. We provide a comparative analysis how these parameters influence the performance, design and applicability of a given replica control protocol for certain application and execution environments.

Acknowledgements This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Spanish National Science Foundation (MICINN) under grant TIN2007-67353-C02, the Madrid Regional Research Council (CAM) under the AUTONOMIC project (S-0505/TIC/000285), and the European Commission under the NEXOF-RA project (FP7-216446).

References

1. Abbadi, A.E., Toueg, S.: Availability in partitioned replicated databases. In: ACM Int. Symp. on Principles of Database Systems (PODS), pp. 240–251 (1986)
2. Amir, Y., Tutu, C.: From Total Order to Database Replication. In: IEEE Int. Conf. on Distributed Computing Systems (ICDCS), pp. 494–506 (2002)
3. Amza, C., Cox, A.L., Zwaenepoel, W.: Distributed Versioning: Consistent Replication for Scaling Back-End DBs of Dynamic Content Web Sites. In: Endler, M., Schmidt, D.C. (eds.) Middleware 2003. LNCS, vol. 2672, pp. 282–302. Springer, Heidelberg (2003)
4. Amza, C., Cox, A.L., Zwaenepoel, W.: A comparative evaluation of transparent scaling techniques for dynamic content servers. In: IEEE Int. Conf. on Data Engineering (ICDE), pp. 230–241 (2005)
5. Bernstein, P.A., Fekete, A., Guo, H., Ramakrishnan, R., Tamma, P.: Relaxed-currency serializability for middle-tier caching and replication. In: ACM SIGMOD Int. Conf. on Management of Data, pp. 599–610 (2006)
6. Bernstein, P.A., Goodman, N.: An algorithm for concurrency control and recovery in replicated distributed databases. ACM Transactions on Database Systems (TODS) 9(4), 596–615 (1984)
7. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
8. Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S., Silberschatz, A.: Update propagation protocols for replicated databases. In: ACM SIGMOD Int. Conf. on Management of Data, pp. 97–108 (1999)
9. Carey, M.J., Livny, M.: Conflict detection tradeoffs for replicated data. ACM Transactions on Database Systems (TODS) 16(4), 703–746 (1991)
10. Cecchet, E., Marguerite, J., Zwaenepoel, W.: C-jdbc: Flexible database clustering middleware. In: USENIX Annual Technical Conference, FREENIX Track, pp. 9–18 (2004)
11. Cecchet, E., Candea, G., Ailamaki, A.: Middleware-based database replication: the gaps between theory and practice. In: ACM SIGMOD Int. Conf. on Management of Data, pp. 739–752 (2008)

12. Chen, P.M., Lee, E.L., Gibson, G.A., Katz, R.H., Patterson, D.A.: Raid: High-performance, reliable secondary storage. ACM Comput. Surv. 26(2), 145–185 (1994)
13. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Computer Surveys 33(4), 427–469 (2001)
14. Chundi, P., Rosenkrantz, D.J., Ravi, S.S.: Deferred updates and data placement in distributed databases. In: IEEE Int. Conf. on Data Engineering (ICDE), pp. 469–476 (1996)
15. Daudjee, K., Salem, K.: Lazy database replication with snapshot isolation. In: Int. Conf. on Very Large Data Bases (VLDB), pp. 715–726 (2006)
16. Elnikety, S., Dropsho, S.G., Pedone, F.: Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In: EuroSys Conference, pp. 117–130 (2006)
17. Elnikety, S., Dropsho, S.G., Zwaenepoel, W.: Tashkent+: memory-aware load balancing and update filtering in replicated databases. In: EuroSys Conference, pp. 399–412 (2007)
18. Gançarski, S., Naacke, H., Pacitti, E., Valduriez, P.: The leganet system: Freshness-aware transaction routing in a database cluster. Information Systems 32(2), 320–343 (2007)
19. Ghanbari, S., Soundararajan, G., Chen, J., Amza, C.: Adaptive learning of metric correlations for temperature-aware database provisioning. In: Int. Conf. on Autonomic Computing, ICAC (2007)
20. Gray, J., Helland, P., O'Neil, P.E., Shasha, D.: The dangers of replication and a solution. In: ACM SIGMOD Int. Conf. on Management of Data, pp. 173–182 (1996)
21. Gupta, A., Mumick, I.S.: Maintenance of materialized views: Problems, techniques, and applications. IEEE Data Engineering Bulletin 18(2), 3–18 (1995)
22. Holliday, J., Agrawal, D., Abbadi, A.E.: The performance of database replication with group multicast. In: IEEE Int. Conf. on Fault-Tolerant Computing Systems (FTCS), pp. 158–165 (1999)
23. Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B., Alonso, G.: Improving the scalability of fault-tolerant database clusters. In: IEEE Int. Conf. on Distributed Computing Systems (ICDCS), pp. 447–484 (2002)
24. Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G., Kemme, B.: Are quorums an alternative for data replication? ACM Transactions on Database Systems (TODS) 28(3), 257–294 (2003)
25. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In: Int. Conf. on Very Large Data Bases (VLDB), pp. 134–143 (2000)
26. Kemme, B., Alonso, G.: A new approach to developing and implementing eager database replication protocols. ACM Transactions on Database Systems (TODS) 25(3), 333–379 (2000)
27. Kemme, B., Pedone, F., Alonso, G., Schiper, A., Wiesmann, M.: Using optimistic atomic broadcast in transaction processing systems. IEEE Transactions on Knowledge and Data Engineering (TKDE) 15(4), 1018–1032 (2003)
28. Leff, A., Rayfield, J.T.: Alternative edge-server architectures for enterprise javaBeans applications. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 195–211. Springer, Heidelberg (2004)
29. Lin, Y., Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R.: Middleware based data replication providing snapshot isolation. In: ACM SIGMOD Int. Conf. on Management of Data, pp. 419–430 (2005)
30. Lin, Y., Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R.: Enhancing edge computing with database replication. In: Int. Symp. on Reliable Distributed Systems (SRDS), pp. 45–54 (2007)
31. Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B.G., Naughton, J.F.: Middle-tier database caching for e-business. In: ACM SIGMOD Int. Conf. on Management of Data, pp. 600–611 (2002)
32. Milan-Franco, J.M., Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B.: Adaptive middleware for data replication. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 175–194. Springer, Heidelberg (2004)

33. Mohan, C., Haderle, D.J., Lindsay, B.G., Pirahesh, H., Schwarz, P.M.: Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems (TODS) 17(1), 94–162 (1992)
34. Pacitti, E., Minet, P., Simon, E.: Fast Algorithm for Maintaining Replica Consistency in Lazy Master Replicated Databases. In: Int. Conf. on Very Large Data Bases (VLDB), pp. 126–137 (1999)
35. Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: MIDDLE-R: Consistent database replication at the middleware level. ACM Transactions on Computer Systems (TOCS) 23(4), 375–423 (2005)
36. Pedone, F., Frølund, S.: Pronto: A fast failover protocol for off-the-shelfcommercial databases. In: Symposium on Reliable Distributed Systems (SRDS), pp. 176–185 (2000)
37. Pedone, F., Guerraoui, R., Schiper, A.: The Database State Machine Approach. Distributed and Parallel Databases 14(1), 71–98 (2003)
38. Perez-Sorrosal, F., Patiño-Martinez, M., Jimenez-Peris, R., Kemme, B.: Consistent and scalable cache replication for multi-tier J2EE applications. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 328–347. Springer, Heidelberg (2007)
39. Plattner, C., Alonso, G.: Ganymed: Scalable replication for transactional web applications. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 155–174. Springer, Heidelberg (2004)
40. Rodrigues, L., Miranda, H., Almeida, R., Martins, J., Vicente, P.: Strong Replication in the GlobData Middleware. In: Proceedings Workshop on Dependable Middleware-Based Systems (part of DSN02), pp. 503–510. IEEE Computer Society Press, Los Alamitos (2002)
41. Röhm, U., Böhm, K., Schek, H.J., Schuldt, H.: FAS - a freshness-sensitive coordination middleware for a cluster of OLAP components. In: Int. Conf. on Very Large Data Bases (VLDB), pp. 754–765 (2002)
42. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1), 42–81 (2005)
43. Schiper, N., Schmidt, R., Pedone, F.: Optimistic algorithms for partial database replication. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 81–93. Springer, Heidelberg (2006)
44. Serrano, D., Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B.: An autonomic approach for replication of internet-based services. In: Int. Symp. on Reliable Distributed Systems (SRDS), pp. 127–136 (2008)
45. Serrano, D., Patiño-Martínez, M., Jiménez, R., Kemme, B.: Boosting database replication scalability through partial replication and 1-copy-SI. In: IEEE Pacific-Rim Conf. on Distributed Computing (PRDC), pp. 290–297 (2007)
46. Sivasubramanian, S., Alonso, G., Pierre, G., van Steen, M.: Globedb: autonomic data replication for web applications. In: Int. World Wide Web Conf (WWW), pp. 33–42 (2005)
47. Wu, S., Kemme, B.: Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In: IEEE Int. Conf. on Data Engineering (ICDE), pp. 422–433 (2005)
48. Zuikeviciute, V., Pedone, F.: Conflict-aware load-balancing techniques for database replication. In: ACM Symp. on Applied Computing (SAC), pp. 2169–2173 (2008)