Introduction to Transactional Replication

Tadeusz Kobus, Maciej Kokociński, and Paweł T. Wojciechowski

Institute of Computing Science, Poznan University of Technology
Poznań, Poland
{Tadeusz.Kobus,Maciej.Kokocinski,
Pawel.T.Wojciechowski}@cs.put.edu.pl

Abstract. Transactional replication is a new enabling technology for service replication. Service replication means that a service runs on a group of processes (service replicas) that work together to execute requests issued by external clients. The characteristic feature of transactional replication is that client requests can be processed on a single replica concurrently as atomic transactions that can read or modify local state. Our goal is to provide an introduction to the transactional replication algorithms. We begin by discussing state machine replication and then present several algorithms that provide full transactional semantics such as deferred update replication and many variants of thereof. Finally, we compare their properties and performance as well as show their strong and weak points.

1 Introduction

Replication is a popular method to increase service reliability and accessibility. It means deployment of a service on several interconnected server machines, each of which may fail independently, and coordination of the service replicas, so that each replica maintains a consistent state view despite failures of communication links or crashes of other replicas. The state is kept by every replica in its local store (the main memory and, optionally, nonvolatile memory).

In this chapter, we survey distributed algorithms that can be used for full replication of services with strong consistency guarantees, without resorting to any central coordinator. An example application is a geo-replicated storage system that ensures strong consistency among all service replicas. Distribution and replication can improve locality and availability of a service by, respectively, moving data closer to the users and processing many requests in parallel. The common feature of the presented algorithms is that they all rely on the fault-tolerant total order (atomic) broadcast primitive (defined in Section 3) that is used to make the state updates consistent among all the replicas despite any crashes. We begin with a simple algorithm of this sort that implements the classical replication scheme relying on atomic broadcast, called *state machine replication (SMR)* [34]. In this approach, a stream of client requests is agreed among all service replicas (that must be deterministic state machines) and processed sequentially by each replica.

Next, we describe example replication algorithms that fall into a different category which we call *transactional replication (TR)* [37]. They can be used to implement a replicated storage system which is then used by replicated services (processes) to process multiple client requests as concurrent transactions. Transactions can read or modify local state and they are executed atomically—completely and successfully or not

R. Guerraoui and P. Romano (Eds.): Transactional Memory, LNCS 8913, pp. 309–340, 2015.

at all, much like in transactional memory systems, but the local state is replicated and kept consistent on many servers. We only focus on one TR scheme relying on atomic broadcast, which is called *deferred update replication (DUR)* [7], and present several algorithms that optimize DUR in various ways. In this approach, a client request can be processed optimistically by any one replica using an atomic transaction, in parallel with other requests (transactions) processed on the same or other replicas. Any state updates are deferred and consistently applied on all replicas on transaction commit.

In the TR systems that provide transactional memory and support full transactional semantics, many transactions can be executed on a single node in parallel. They can perform arbitrary operations (with possible restrictions, such as the use of irrevocable operations) and may also commit or abort (and possibly restart) on demand. On the other hand, pure SMR does not offer full transactional semantics—on each node there is only one transaction executed at a time (spawned for processing a client request) that is only allowed to commit and its code must be deterministic. However, we largely ignore the semantic differences and give, in Section 3, a common specification of the SMR and TR schemes in terms of properties describing the inter-replica and client-replica interactions.

In the chapter, we discuss the following replication algorithms:

1. SMR – state machine replication based on total order broadcast; the algorithm resembles the original idea proposed in [17,33] but was modified to optimize the read-only transactions, as in [29];
2. DUR – deferred update replication that follows the idea presented in [7];
3. MvDUR – deferred update replication with multiversioning; it extends the previous algorithm with an optimization technique that dates back to first database systems [4];
4. HTR – a hybrid state-machine-based and deferred-update replication scheme proposed in [14], which seamlessly combines SMR and DUR with multiversioning into one replication scheme;
5. Postgres-R – an algorithm proposed in [12] that aims at improving DUR by reducing the amount of data transmitted via a network;
6. EDUR – executive deferred update replication proposed in [15], which uses a leader of the broadcast protocol to streamline transaction certification.

Interestingly, replication schemes designed for database systems often are not suitable for replication of services (processes) or software transactional memory (see e.g., [30,25]). This is because typical workloads in TR systems are much different than those in SQL database systems, in which transactions are relatively long, require time-consuming optimizations of queries, and perform costly I/O operations. On the contrary, transactions performed by a replicated service are usually very short (take a fraction of a millisecond to execute) and access few shared objects. The crucial observation here is that, usually, the transaction execution times are much shorter than the latency caused by the network communication. Hence, the main design aspect of TR systems is minimizing the inter-replica synchronization footprint, both in terms of the communication steps as well as the amount of data that need to be communicated between replicas.

We aim our chapter at developers of replication frameworks and all those who would like to learn about transactional replication. We therefore explain the algorithms and

their properties in detail. Each algorithm is presented by giving its pseudocode. We then compare the algorithms taking into account their semantics, the overhead due to concurrency control and transaction certification, the number of communication steps, the number and volume of network transmissions, and the expected performace under different workload types.

The structure of the chapter is as follows. We begin by defining the system model and transactional replication properties in Section 2 and discuss problems that face the designers of such systems in Section 3. Then, we present the SMR algorithm in Section 4. Next, we describe DUR in Section 5 and various ways this scheme can be optimized (the MvDUR, HTR, Postgres-R, and EDUR algorithms) in Sections 6–9. Then, we compare the presented algorithms in Section 10, and finally conclude and give references to related work in Section 11.

2 System Model and Properties

In this section, we describe the system model and properties. A *replicated process* $P = \{P_1, ..., P_n\}$ consists of n service processes (replicas) P_i ($i = 1..n$) running on independent machines (nodes) connected via a network. Each process P_i has access to its own volatile memory and stable storage; the combined content of the two constitutes *local state*. $S = \{S_1, ..., S_n\}$ is a *replicated state*, where S_i is a local state of process P_i ($i = 1..n$). A transaction executed by process P_i can only access objects that belong to local state S_i.

We assume a crash-recovery failure model in which processes may crash independently and later on recover and rejoin computation. Processes can recover its local state either from stable storage or other replicas (as in e.g. JPaxos [16]). However, the recovery algorithms are beyond the scope of this chapter. A process is said to be *up* if it correctly executes its program. Upon crash, a process fails by ceasing communication with any other processes and becomes a *down* process. It can rejoin distributed computation upon a recovery event which requires executing a recovery procedure. A process is said to be *unstable* if it crashes and recovers infinitely many times. A process is *correct* if it is eventually permanently up (there is a time after which it never crashes). Otherwise, it is *faulty*, i.e. unstable or eventually permanently down (there is a time when it crashes and later never recovers).

Our replication algorithms are aimed at distributed asynchronous systems which can be characterized as follows. There is no central coordinator and the processes communicate solely by exchanging messages using bidirectional *fair-loss links* [2]. For simplicity, however, all presented algorithms use *perfect links* (no messages are lost) since they can be easily implemented on top of fair-loss links (see e.g., [5]). Messages may be lost and no upper bound on message transmission is known. The failure pattern of messages is independent from the one of processes. No assumption is also made on the relative computation speeds of the processes. However, we assume availability of a failure detector Ω [6], which is the weakest failure detector capable of solving consensus in a distributed asynchronous system in which processes or communication links may fail.

In addition to service processes we consider an unspecified number of external client processes. We assume that the clients are independent and they do not communicate

Properties of a replicated process P:

R1: *Validity:* If a process P_j modifies object o with v during state update, then $\mathrm{w}(o^k)v$ was executed by some process P_i ($i = j$ or $i \neq j$) as part of some transaction that commits.

R2: *Termination:* On commit of a transaction T, every correct process P_i eventually applies $T's$ updates (modified objects) to its local state S_i.

R3: *Integrity:* No process updates its state twice as the result of executing a transaction T.

R4: *Agreement:* No two correct processes update their state differently as the result of executing a transaction T.

R5: *Atomicity:* Operations of a transaction T and any T's updates to S are performed atomically.

R6: *Causal order:* No process P_i updates state S_i as the result of request r_2 unless P_i has already updated S_i as the result of any update request r_1, such that $r_1 \xrightarrow{c} r_2$.

R7: *Total order:* Let r_1 and r_2 be any two requests. Let P_i and P_j be any two processes that update state as the result of r_2. If P_i updates state on r_1 before r_2 then P_j updates state on r_1 before r_2.

Properties of client-P interaction:

C1: *Validity:* If a client sends a request r to a correct process P_i then replicated process P executes T and eventually returns the response to r to the client.

C2: *No creation:* If a request r is handled by some process P_i, then r was previously sent by some client.

C3: *No duplication:* No response is delivered more than once.

C4: *Causal order:* Let r_1 and r_2 be any two requests such that $r_1 \xrightarrow{c} r_2$. If res_1 and res_2 are responses to these requests (r_1 and r_2, respectively) delivered to the client, then res_1 is delivered before res_2.

Fig. 1. Properties of transactional replication

with each other directly. The only possible client interaction is through the replicated service. They can submit requests to any of the service processes and await responses. A client may submit only one request at a time. If a client does not receive any response after submitting a request, it can choose a different replica and issue the request again. Such a situation can occur if a replica is down or a timeout was reached due to high communication latency. Each request is processed by an atomic transaction. In case of optimistic replication schemes, such as DUR, transactions are executed in parallel and some of them may conflict. The conflicting transactions are reexecuted until they finally commit (or explicitly abort). However, the clients are not aware of transaction reexecutions.

We assume a simple communication interface: to communicate with a replicated service, a client sends a request message $\langle \texttt{Request} \mid (id, LC, code, args) \rangle$, denoted r, which is then handled by a replicated process P by executing an atomic transaction T identified by $r.code$, where T can use arguments $args$; id is the message identifier and LC will be explaned in Section 3. Then, replicated process P will return to the client a response to request r using a message $\langle \texttt{Response} \mid (r.id, LC, res) \rangle$, where result res

depends on the local state read by transaction T. We use a notation *r.a* to denote a record field a of message r.

In general, transactions can execute any legal program containing operations $\mathrm{r}(o^k)v$, $\mathrm{w}(o^k)v$, **abort**, and **retry**, which respectively, read or write a value v to object o in version k, and abort or retry T. Writes of transaction T to object versions on replica P_i can be seen by other transactions on P_i only after P_i *updates* state S_i with the modified objects. All transactions (including retried) eventually *commit* or *abort*. On commit, T updates a replicated state S (with modified objects) and returns result *res*. On abort, T returns $\emptyset$.

In Figure 1, we define the properties of a transactional replication system, taking into account the handling of requests by a replicated process P (rules **R1-R7**) and the interaction between the clients and P (rules **C1-C4**). All algorithms described in this chapter guarantee these properties. In the specification, we use the symbol $\xrightarrow{c}$ to denote a causal order relation defined as follows: if $r_1 \xrightarrow{c} r_2$, then request r_2 depends on result *res* returned by r_1.

3 Replicated Algorithm Design Problems

Replication of a service means maintaining the service's code and state on a number of machines, so when some of them fail, others can continue to provide the service and process clients' requests. The *service's state* consists of all data which the service and the replication protocol operate on and their current status of execution. Developing replication frameworks is challenging due to some known fundamental problems in distributed systems. Below we discuss the problems which are related to inter-replica and client-replicas synchronization, and fault-tolerance.

Inter-replica synchronization. In order to guarantee consistency of state updates, replicas must synchronize, which is inherently difficult in a distributed system. Formally, many such problems can be reduced to the problem of *consensus*, i.e. reaching agreement among a group of distributed processes on a single value proposed by one of them. It has been proven that this problem is impossible to solve in a fully asynchronous distributed system [10]. However, some additional assumptions can be made about the system (e.g., the existence of partial synchrony and failure detectors) which make this problem solvable. Solving the consensus problem efficiently is essential for performance of TR schemes described in this chapter. The best known algorithm of this sort is Paxos [18], which solves the consensus problem assuming that the majority of processes is not faulty (meaning not down). If a "faulty process" is as defined in Section 2, then also some additional mechanism is required to support process recovery (see e.g., [16]). In fact, Paxos can solve an infinite sequence of consensus instances. Thus, distributed processes can use this protocol to propose (in multiple consensus instances) and agree upon a common set and order of messages. This semantics is captured by *Total Order Broadcast (TOB)* [5,9]. This primitive enables reliable broadcast of messages with a guarantee that all messages are delivered by all non-faulty processes in the same order. All algorithms discussed in this chapter rely on TOB or protocols derived from it. This, in turn, allowed us to directly compare them.

Client-replicas synchronization. The interaction between the external clients and the replicated service is not trivial. Imagine a client who issues a request r_1 to one of

the replicas, say P_i. P_i handles r_1 and sends a response back to the client. The response to request r_1 can only be sent after the request is *stable* in the system, which means that P_i has updated its local state and it is sure that all other non-faulty replicas will also *eventually* update state. Therefore, some of the replicas may lag behind others. It is possible, then, that upon receiving a response to r_1, the client issues a new request r_2 to a replica P_j that lags behind P_i. If P_j subsequently executes r_2 and r_2 is causally dependent on r_1 (which is typical), then inconsistencies may be introduced to the system.

Fortunately, this problem can be easily solved using logical clocks LC_i that are maintained by replicas P_i ($i = 1..n$). Every replica P_i will increment LC_i each time it has updated local state S_i. A replica P_i which is handling a client request r_1 will return to the client the current value of LC_i in response to r_1 just after r_1 is stable. The client can attach the obtained clock value to a subsequent request r_2 (in a field $r_2.clock$). Since the clock values are monotonically increased, a replica handling r_2 can check whether its state is up-to-date and so it can execute the request, or it has to postpone its processing until it synchronizes with the rest of the replicas. All algorithms presented in this chapter feature this mechanism.

Consider yet another troublesome scenario. A client sends a request r to a replica P_i and awaits the response. P_i crashes before sending reply to the client, or it takes exceptionally long time for the replica to reply to the client. The client may become impatient and issue request r again, but this time to another replica. In effect, the request may be executed twice. To prevent this undesirable behavior, a history could be maintained (and garbage collected after some time) of all requests sent by each client, which will allow detection of duplicates. However, for brevity, we omit this code in the presentation of the algorithms.

Fault-tolerance. The transactional replication systems must be robust against failures. Ideally, a replicated service should be operational when all machines except one crash. However, this requirement is usually too strong since systems fulfilling it cannot be implemented efficiently. It is because the replicas would have to extensively use stable storage in order to be able to recover in the event of failures. On the other hand, if majority of processes is up and running at any time, recovery of failed processes can be very efficient and does not require replicas to access stable storage during the normal (non-faulty) operation. All of the replication schemes discussed in this chapter fall into the latter category.

4 State Machine Replication

State Machine Replication (SMR) [17,33] is one of the simplest replication schemes. It does not support full transactional semantics, but we included SMR in our discussion as it serves as a base for some optimized TR schemes. In this replication scheme, a service replica (process) begins execution on every server from the same initial state and advances by processing all client requests sequentially. Note that each process has to be deterministic. Otherwise, consistency among replicas could not be preserved as the replicas might diverge. Then, the crucial element of SMR is the protocol which is used for dissemination of requests to be executed by all processes in the same order. The required semantics is provided by the Total Order Broadcast (TOB) protocol defined in Section 3.

Algorithm 1. State Machine Replication for process p_i

```
1: integer LC ← 0

Thread q on request r from client c (executed on one replica)
2: response res ← ⊥
3: upon INIT
4:     if r.code is read-only then
5:         wait until LC ≥ r.clock
6:         lock { res ← execute r.code with r.args }
7:     else
8:         TO-BROADCAST r
9:         wait for res
10:    return (r.id, LC, res) to client c

The main thread of SMR (executed on all replicas)
11: response res ← ⊥
12: upon TO-DELIVER (request r)
13:     lock { res ← execute r.code with r.args
14:            LC ← LC + 1 }
15:    if request with r.id handled locally by thread q then
16:        pass res to thread q
```

Algorithm. In Algorithm 1, we show an optimized version of SMR which differentiates between updating and read-only requests [29], thus allowing for some level of parallelism in the execution of requests. For simplicity, we assume that each incoming request is handled in a separate thread. Depending on whether the request is read-only or not, the replica either executes it locally, or broadcasts it to all processes. As for broadcast, a replica uses the *TO-Broadcast* primitive of TOB (line 8). The request is delivered by each replica (through the *TO-Deliver* event) and independently executed by the replica's main thread (line 13). Finally, the replica that received the request, sends the response back to the client (line 10).

On the other hand, if the request is read-only, the replica has to first make sure that it is aware of the changes performed by all requests issued by the same client earlier (this procedure pertains to the problem described in Section 3). For this purpose, each replica stores a logical clock variable *LC* and attaches its current value to every response message that is sent to the client. This value is then enclosed in the subsequent request message issued by the client (in the field *clock*) and is used to check whether the replica that handles the request is up-to-date, so that its execution will not result in any inconsistencies (line 5).

In the presented algorithm, replicas do not perform the above check for updating requests since the execution order of the updating requests is determined in SMR by TOB and so it is the same at every replica. In effect, if r_1 and r_2 are any two requests, such that r_2 causally depends on r_1, then r_2 can be executed at each replica only after the replica executed r_1.

In our SMR algorithm, the execution of requests is performed within a critical section, guarded by a lock (lines 6 and 13). It is because read-only requests cannot be processed concurrently with updating requests. Otherwise, they could encounter inconsistencies, since the updating requests do not operate on copies of objects they modify, as it is in other schemes described in this chapter, but instead they perform write operations in place of the old values. It would be possible to run several read-only requests

in parallel, but this optimization would require using readers-writers locks to protect critical sections of lines 6 and 13, respectively.

Discussion. The advantages of SMR are obvious. This replication scheme is simple and can handle machine failures well. However, the performance of SMR is limited by the capacity of any replica to process the updating requests sequentially. It can therefore neither benefit from modern multicore architectures nor scale with the increasing number of replicas. Furthermore, the semantics of SMR is not as rich as the one available in the TR scheme, e.g., processing of requests cannot be rolled back or wait for a condition to be met.

5 Deferred Update Replication

In the rest of the chapter, we focus on *multi primary-backup replication* [7] (also called *multi-master replication*), where each request is executed by only one single replica that processes the request and issues updates to other replicas, but all replicas can process requests in parallel. In this approach, we have to be able to resolve any conflicts which take place between concurrent threads that access the same set of objects and at least one of the threads modifies the shared object. Here is where the transaction abstraction comes into play. Then each request is executed as an atomic transaction whose operations logically occur at a single instant in time, so the intermediate states are not visible to other transactions. Furthermore, atomicity prevents updates to the state from occurring only partially.

For efficiency, it is important to limit the amount of synchronization among threads and replicas. Hence, we focus on replication schemes featuring *optimistic concurrency control*. They require much less synchronization than those relying on the *pessimistic* one. It is because transactions are executed without upfront locking of objects that are to be accessed by these transactions but, instead, they operate on their own local copies of the objects. Any object modifications are then applied to the replica state on transaction commit.

Deferred Update Replication (DUR) [7] is the simplest transactional replication scheme of this sort. Typically, DUR supports full replication, and each replica can handle multiple requests in separate threads using optimistic transactions. The transactional semantics ensures that the requests are processed atomically and in isolation. The transaction's execution phase is followed by the committing phase in which the replicas synchronize and certify transactions.

Transaction certification means checking if a committing transaction does not conflict with concurrent transactions. It is the only moment in a transaction's lifetime that requires replica and thread synchronization. Upon successful certification, replicas update their state. Otherwise, the transaction is rolled back and restarted. Many different protocols can be used for transaction certification. In this chapter, we discuss *DUR relying on Total Order Broadcast* (see e.g., [27,26,1] among others). Using TOB avoids blocking and limits the number of costly synchronization steps [1,13,26] (see also [32,11]).

Algorithm. In Algorithm 2, we give pseudocode for DUR that builds on [14]. Each replica maintains two global variables. The first one, *LC*, represents the logical clock

Algorithm 2. Deferred Update Replication for process p_i

```
1: integer LC ← 0
2: set Log ← ∅
3: function GETOBJECT(txDescriptor t, objectId oid)
4:     if (oid, obj) ∈ t.updates then
5:         value ← obj
6:     else
7:         lock { value ← retrieve object oid }
8:     return value
9: function CERTIFY(integer start, set readset)
10:    lock { L ← {t ∈ Log : t.end > start} }
11:    for all t ∈ L do
12:        writeset ← {oid : ∃(oid, obj) ∈ t.updates}
13:        if readset ∩ writeset ≠ ∅ then
14:            return failure
15:    return success
```

```
Thread q on request r from client c (executed on one replica)
16: txDescriptor t ← ⊥                          // type: record (id, start, end, readset, updates)
17: response res ← ⊥
18: upon INIT
19:     wait until LC ≥ r.clock
20:     raise TRANSACTION
21:     return (r.id, LC, res) to client c
22: upon TRANSACTION
23:     t ← (a new unique id, 0, 0, ∅, ∅)
24:     lock { t.start ← LC }
25:     res ← execute r.code with r.args
26:     COMMIT()
27: upon READ(objectId oid)
28:     t.readset ← t.readset ∪ {oid}
29:     if CERTIFY(t.start, {oid}) = failure then
30:         raise RETRY
31:     else
32:         return GETOBJECT(t, oid)
33: upon WRITE(objectId oid, object obj)
34:     t.updates ← {(oid', obj') ∈ t.updates : oid' ≠ oid} ∪ {(oid, obj)}
35: procedure COMMIT
36:     if t.updates = ∅ then
37:         return to INIT
38:     if CERTIFY(t.start, t.readset) = failure then
39:         raise RETRY
40:     TO-BROADCAST t
41:     wait for outcome
42:     if outcome = failure then
43:         raise RETRY
44:     else                                        // outcome = success
45:         return to INIT
46: upon ROLLBACK
47:     stop executing r.code and return to INIT
48: upon RETRY
49:     stop executing r.code
50:     raise TRANSACTION
```

```
The main thread of DUR (executed on all replicas)
51: upon TO-DELIVER (txDescriptor t)
52:     outcome ← CERTIFY(t.start, t.readset)
53:     if outcome = success then
54:         lock { t.end ← LC
55:                Log ← Log ∪ {t}
56:                apply t.updates
57:                LC ← LC + 1 }
58:     if transaction with t.id executed locally by thread q then
59:         pass outcome to thread q
```

which is used in a similar way as in SMR, i.e., *LC* is incremented every time a replica changes its state (line 57) and enables the replica to track whether its state is recent enough to execute the client's request (line 19). Additionally, *LC* is used to mark the start and the end of the transaction execution (lines 24 and 54). The transaction's start and end timestamps, stored in the *transaction descriptor* (line 16), allow us to reason about the precedence order between transactions. Let t_1 and t_2 be transaction descriptors of two transactions T_1 and T_2. We say that transaction T_1 *precedes* transaction T_2 (denoted $T_1 \rightarrow T_2$) iff $t_1.end < t_2.start$. If neither $T_1 \rightarrow T_2$ nor $T_2 \rightarrow T_1$, we say that T_1 and T_2 are *concurrent*. The second variable, *Log*, is a set used to store the transaction descriptors of committed transactions. Maintaining this set is necessary to perform transaction certification.

The DUR algorithm detects any conflicts among transactions by checking whether a given transaction T that is being certified read any stale data. The latter occurs when T read any shared objects that have been modified by a concurrent but already committed transaction. For this purpose, DUR traces the accesses to shared objects independently for each transaction. The identifiers of objects that were read and the modified objects themselves are stored in private, per transaction, memory spaces: *readset* and *updates*. On every read, an object's identifier is added to the *readset* (line 28). Similarly, on every write a pair of the object's identifier and the corresponding object is recorded in the *updates* set (line 34). Then, the CERTIFY function compares the given *readset* against the *updates* of all the committed transactions in *Log* that are concurrent with the tested transaction. If it finds any non-empty intersection of the sets, the outcome is negative. Otherwise, it is positive (no conflicts detected, the transaction is certified successfully). Note that every time a transaction reads some shared object, a check against conflicts is performed (line 29). This way T is guaranteed to always read from a consistent snapshot. When a conflict is detected, T is forced to retry.

When a transaction's code completes, the COMMIT operation (line 35) is used to end the transaction and initiate the committing phase, which can be explained as follows. If T is a read-only transaction (T did not modify any objects), it can commit straight away, without performing any further conflict checks or replica synchronization, similarly as in SMR (lines 36–37). A read-only transaction does not need to perform certification as the possible conflicts would have been detected earlier, upon read operations (line 29). For update transactions, first, the local certification takes place (line 38), which is not mandatory but allows the replica to detect conflicts earlier, and thus sometimes avoid costly network communication. Next, the transaction's descriptor containing *readset* and *updates* is broadcast to all processes using TO-BROADCAST (line 40). The message is delivered in the main thread, where the final certification takes place (line 52). Upon successful certification of transaction T, replicas apply the updates performed by T and commit it (lines 54–57). Otherwise, T is rolled back and reexecuted by the same replica.

To manage the control flow of a transaction, the programmer can use two additional operations: ROLLBACK and RETRY, whose semantics is similar as in transactional memory systems. The ROLLBACK operation (line 46) stops the execution of a transaction and revokes all the changes it performed so far. The RETRY operation (line 48) forces a transaction to rollback and restart.

For clarity, we made several simplifications. Firstly, note that the operations on *LC* (lines 24, 54, 57), *Log* (lines 10 and 55) and the accesses to transactional objects (lines 7 and 56) have to be synchronized. For simplicity, a single global lock is used. For better performance, the implementation can rely on fine-grained locks. Secondly, in our pseudocode, *Log* can grow indefinitely. In reality, *Log* can easily be kept small by garbage collecting information about the already committed transactions that ended before the oldest live transaction started in the system.

In the presented algorithm, we use the same certification procedure for both the certification test performed upon every read operation (line 29) and the certification test that happens after a transaction descriptor is delivered to the main thread (line 52). In practice, however, doing so would be very inefficient. It is because for every read operation, we check for the conflicts against all concurrent transactions (line 10), thus performing much of the same work again and again. However, this repeated actions can be easily avoided by associating the accessed shared objects with version numbers—the value of *LC* at the time the objects were most recently modified.

Discussion. It is easy to see that, at least theoretically, DUR has the potential to perform much better than SMR. The capability of executing requests in parallel is especially valuable for CPU-intensive workloads. Unfortunately, there are also factors that limit the robustness of DUR. Firstly, the system has to monitor transactional accesses to all shared objects, which is costly. This overhead cannot be avoided unless we know *a priori* the conflict pattern of all transactions. Secondly, the volume of data exchanged via a network is high, mainly due to, usually large, *readsets* that have to be broadcast alongside *updates*. Thirdly, transaction certification, which can be a costly operation, is performed independently for every transaction by each process, thus limiting scalability. In the next sections, we present several replication algorithms that address some of the above problems.

6 Deferred Update Replication with Multiversioning

Multiversioning [4] in an important optimization technique which allows for multiple versions of transactional objects that are transparent to the programmer. Only one object version is accessible by a transaction at any time. Object versions are immutable, thus they can be accessed concurrently without any synchronization. Furthermore, since read-only transactions accessing object versions are abort-free, the system does not need to trace accesses to shared objects for transactions *a priori* known to be read-only. The latter feature can greatly improve the overall performance and scalability of the transactional system when workloads are dominated by read-only transactions [30]. All TR algorithms described in this chapter can benefit from this optimization technique.

Algorithm. In Algorithm 3, we present the DUR scheme extended with multiversioning, which we call *Multiversion DUR (MvDUR)*. In MvDUR, the information about already committed transactions is no longer stored in *Log*, and each object can have many *object versions obj*, each one paired with their corresponding *version numbers ver*. When a transaction commits, the system creates new versions of all objects modified by the transaction (lines 56–57), all having the same version number assigned, which is equal to the current value of logical clock *LC*.

Algorithm 3. Deferred Update Replication with Multiversioning for process p_i

```
1: integer LC ← 0
2: function GETVERSION(objectId oid, integer notNewerThan)
3:     lock { return (obj, ver) such that obj is a version of object oid whose version number ver
4:                                 is the highest available such that ver ≤ notNewerThan }
5: function GETOBJECT(txDescriptor t, objectId oid)
6:     if (oid, obj) ∈ t.updates then
7:         value ← obj
8:     else
9:         (obj, ver) ← GETVERSION(oid, t.start)
10:        value ← obj
11:    return value
12: function CERTIFY(integer start, set readset)
13:    for all id ∈ readset do
14:        (obj, ver) ← GETVERSION(id, ∞)
15:        if ver > start then
16:            return failure
17:    return success
```

```
Thread q on request r from client c (executed on one replica)
18: txDescriptor t ← ⊥                          // type: record (id, start, end, readset, updates)
19: response res ← ⊥
20: upon INIT
21:     wait until LC ≥ r.clock
22:     raise TRANSACTION
23:     return (r.id, LC, res) to client c
24: upon TRANSACTION
25:     t ← (a new unique id, 0, 0, ∅, ∅)
26:     lock { t.start ← LC }
27:     res ← execute r.code with r.args
28:     COMMIT()
29: upon READ(objectId oid)
30:     obj ← GETOBJECT(t, oid)
31:     if r.code is not read-only then
32:         t.readset ← t.readset ∪ {oid}
33:     return obj
34: upon WRITE(objectId oid, object obj)
35:     t.updates ← {(oid', obj') ∈ t.updates : oid' ≠ oid} ∪ {(oid, obj)}
36: procedure COMMIT
37:     if t.updates = ∅ then
38:         return to INIT
39:     if CERTIFY(t.start, t.readset) = failure then
40:         raise RETRY
41:     TO-BROADCAST t
42:     wait for outcome
43:     if outcome = failure then
44:         raise RETRY
45:     else                                       // outcome = success
46:         return to INIT
47: upon ROLLBACK
48:     stop executing r.code and return to INIT
49: upon RETRY
50:     stop executing r.code
51:     raise TRANSACTION
```

```
The main thread of MvDUR (executed on all replicas)
52: upon TO-DELIVER (txDescriptor t)
53:     outcome ← CERTIFY(t.start, t.readset)
54:     if outcome = success then
55:         lock { LC ← LC + 1
56:                for all (oid, obj) ∈ t.updates
57:                    add obj as a new version of object oid with version number LC }
58:     if transaction with t.id executed locally by thread q then
59:         pass outcome to thread q
```

Compared to DUR, there is also a new function GETVERSION which takes two arguments *oid* and *notNewerThan* and retrieves a version *obj* of an object identified with *oid* that is the most recent among all those object versions that have a version number lower than or equal to *notNewerThan* (lines 2–4). The function can be used to read from a consistent snapshot of the system and return the newest object versions that existed in the system up to a given moment in time. This way all reads which are performed by a transaction are consistent and no conflict checks are necessary. Therefore, read-only transactions are guaranteed to always commit. For this reason, as stated earlier, if a transaction is *a priori* known to be read-only, it does not need to record its accesses in the *readset* (line 31).

The transaction certification phase in MvDUR is different and much more efficient than in DUR. Instead of checking a transaction's *readset* against the *update* sets of (possibly many) committed concurrent transactions, the certification procedure just compares the version numbers of the objects that were read. If the most recent version of a read object has a version number which is higher than the transaction's *start* timestamp, then a conflict exists—i.e., a new version was created after the transaction had already started execution.

The committing phase in MvDUR is similar to DUR's one. Both algorithms differ in the way each replica applies transaction updates. In MvDUR, replicas update their state by adding new object versions (lines 56–57). For this, we have to use locks since these operations must be done atomically. However, in practice MvDUR can be implemented in a way that avoids using locks altogether.

In our pseudocode, no object version is ever removed from the system. However, a simple garbage collection mechanism can be proposed, as follows. Let us consider a replica R, and let t be the transaction descriptor of the oldest live transaction in R ($t.start$ is equal to the the lowest value among all descriptors of live transactions in R). Let d be the set of all object versions in R whose version numbers are lower than or equal $t.start$. Then, for each shared object, all its versions in d but the most recent one can be safely dropped.

7 Hybrid SM-DUR Algorithm

The SMR and DUR (or MvDUR) replication schemes presented in previous sections are based on different premises. In SMR, any sequential program implementing some service can be replicated, and the replication framework simply broadcasts requests using TOB. On the other hand, DUR requires the service's program to be transaction-oriented, but it offers potentially much better scalability due to its capability of processing requests in parallel. The two schemes were compared both theoretically and experimentally in [37]. The main corollary drawn from this comparison is that no scheme can be considered superior.

In SMR, all requests are executed sequentially by all replicas, which does not leave much room for performance improvement. Therefore, it might seem that DUR, which supports parallelism, should always outperform SMR. However, this is not the case for

several reasons. Firstly, the size of messages broadcast can be an order of magnitude larger than in SMR since a message contains not only the *updates* that result from the transaction execution but also the *readset*, necessary for transaction certification. Especially the latter set can be of significant size. Also, the cost of bookkeeping *readset* and *updates* is not negligible. On the other hand, a message broadcast in SMR usually only contains a client request with a reference (with some arguments) to a function that executes this request. Therefore, it is sometimes more efficient to broadcast a client request, as in SMR, rather than broadcast the state changes, as in DUR, even at the cost of executing the request n times independently on each replica. Secondly, there is also the aspect of concurrency control and its inherent cost in the optimistic replication schemes. In DUR, transactions may be forced to retry due to conflicts, so a transaction can be executed multiple times before it eventually commits. If the contention level is high, the benefits of parallel execution in DUR may not only be overshadowed but even completely outweighed by the costly transaction reruns. This, in turn, causes the performance of the system to diminish. On the contrary, in SMR no conflicts ever occur.

The SMR and DUR (or MvDUR) replication schemes also differ in the semantics offered to the programmer. Unlike DUR, SMR only supports deterministic services. Otherwise, replicas could diverge when processing the same request and eventually cause the system to run into inconsistencies. On the other hand, the fact that each request (transaction) is executed in SMR exactly once by each process, and is never forcefully retried can be an advantage. For instance, it allows SMR to support operations with side-effects that cannot be easily undone, such as I/O, system calls, etc. On the other hand, the basic DUR scheme cannot deal with irrevocable operations well because each transaction may execute multiple times before it eventually commits. In transactional memory systems, various techniques were developed to deal with this problem, such as buffering or executing irrevocable transactions sequentially w.r.t. other transactions. They can be used to extend DUR accordingly.

These insights led us to merge SMR and DUR into *Hybrid Transactional Replication (HTR)* [14]. In this replication scheme, the programmer can use transactional constructs to encode handlers of client requests as atomic transactions, similarly as in DUR. However, each transaction is executed in one of two execution modes that are selected dynamically: a pessimistic one (*SM mode*) and an optimistic one (*DU mode*). A transaction which is executed in the SM mode is guaranteed an abort-free execution, but its code has to be deterministic. Moreover, HTR makes sure that only one such a transaction is run in the system at a time. On the other hand, a transaction which is executed in the DU mode can run in parallel with any SM transaction and any other DU transactions. Because a DU transaction is executed only by one replica process, it can also contain non-deterministic operations. However, a DU transaction may abort, so the client requests that require irrevocable operations should only be executed as SM transactions.

Algorithm. Before we dive into the details of HTR, let us discuss the key idea of how the two transaction execution modes can coexist. The way SM and DU transactions are executed in HTR closely resembles how the client requests are handled, respectively, by SMR and DUR, but objects are not modified in place as it is in SMR. HTR manages the two modes by serializing the execution of SM transactions with the certification of DU transactions. Therefore, during the execution of a SM transaction, no other transaction

can modify the system state. This way a SM transaction operates on consistent state and is guaranteed an abort-free execution. Note that DU transactions execute in isolation on copies of shared objects, so no interference with other transactions is possible. The order in which the main HTR thread certifies DU transactions and executes SM transactions is determined by TOB. Therefore, each replica advances exactly in the same way.

The pseudocode of the HTR algorithm (see Algorithm 4) shares many parts with MvDUR, on which HTR is based. [1] HTR features an abstraction called the *transaction oracle*. After a replica receives a request, the oracle is queried to asses whether to execute the request as a DU or SM transaction (line 25). In practice, the decision made by the oracle relies on hints declared by the programmer as well as on dynamically collected data regarding various aspects of system's performance. Note that, the request execution mode is determined on per transaction execution basis. It means that a request can be first executed multiple times as a DU transaction (due to aborts) and then as a SM transaction (which is guaranteed to always commit).

The execution and committing phases of DU transactions are almost identical as in MvDUR. The only difference lies in feeding the oracle with the statistics regarding transaction execution (lines 54, 57 and 61) which, in turn, allow the oracle to adjust its future decisions. On the other hand, if the oracle determines that a request is to be executed as a SM transaction, it is first broadcast using TOB (line 32). When the request is delivered, it is processed by the same thread that certifies DU transactions and applies their updates (lines 76–79). A SM transaction does not execute directly on the shared objects as in SMR. Instead, it uses shared object copies as a DU transaction does. By doing so, a SM transaction can be easily rolled back on demand at any time. Moreover, SM transactions produce versions of objects that can be used by other transactions (including the read-only ones) exactly the same way as the versions produced by regular DU transactions. For this purpose, HTR features the appropriate upon statements (TRANSACTION, READ, WRITE, ROLLBACK, and RETRY) in the main thread section. Since a SM transaction is guaranteed to commit, it does not need to maintain *readset* (line 86). A SM transaction commits by simply applying the updates it produced (lines 90–92) and returning the result to the thread that originally received the request (lines 78–79).

Discussion. HTR brings together the best features of both SMR and DUR. It offers rich transactional semantics, also when the client requests are executed in the SM mode. Additionally, it supports irrevocable operations, which is not typical in replication schemes featuring optimistic concurrency control. In terms of performance, HTR is at least as good as either SMR or DUR. Moreover, HTR can dynamically adapt to a changing workload because the oracle can monitor the system's performance and adjust its decisions accordingly. However, for HTR to perform well, the oracle has to be tailored to the application in question. In [14], we outline the most important aspects of a good oracle design and describe example oracles for several benchmark applications.

[1] HTR does not require multiversioning in order to work. However, the only existing implementation of HTR is based on MvDUR [14].

Algorithm 4. Hybrid Transactional Replication for process p_i (part 1)

```
1: integer LC ← 0
2: function GETVERSION(objectId oid, integer notNewerThan)
3:     lock { return (obj, ver) such that obj is a version of object oid whose version number ver
4:                              is the highest available such that ver ≤ notNewerThan }
5: function GETOBJECT(txDescriptor t, objectId oid)
6:     if (oid, obj) ∈ t.updates then
7:         value ← obj
8:     else
9:         (obj, ver) ← GETVERSION(oid, t.start)
10:        value ← obj
11:    return value
12: function CERTIFY(integer start, set readset)
13:    for all id ∈ readset do
14:        (obj, ver) ← GETVERSION(id, ∞)
15:        if ver > start then
16:            return failure
17:    return success
```

```
Thread q on request r from client c (executed on one replica)
18: txDescriptor t ← ⊥                          // type: record (id, start, end, readset, updates, stats)
19: response res ← ⊥
20: upon INIT
21:    wait until LC ≥ r.clock
22:    raise TRANSACTION
23:    return (r.id, LC, res) to client c
24: upon TRANSACTION
25:    mode ← TransactionOracle.query()
26:    if mode = DU then
27:        t ← (a new unique id, 0, 0, ∅, ∅, ∅)
28:        lock { t.start ← LC }
29:        res ← execute r.code with r.args
30:        raise COMMIT()
31:    else                                                          // mode = SM
32:        TO-BROADCAST r
33:        wait for (outcome, res, t)
34:        UPDATEORACLESTATISTICS(t)
35:        if outcome = retry then
36:            raise TRANSACTION
37: upon READ(objectId oid)
38:    obj ← GETOBJECT(t, oid)
39:    if r.code is not read-only then
40:        t.readset ← t.readset ∪ {oid}
41:    return obj
42: upon WRITE(objectId oid, object obj)
43:    t.updates ← {(oid′, obj′) ∈ t.updates : oid′ ≠ oid} ∪ {(oid, obj)}
44: procedure COMMIT                                                // for DU transactions
45:    if t.updates = ∅ then
46:        return to INIT
47:    if CERTIFY(t.start, t.readset) = failure then
48:        raise RETRY
49:    TO-BROADCAST t
50:    wait for outcome
51:    if outcome = failure then
52:        raise RETRY
53:    else                                                          // outcome = success
54:        UPDATEORACLESTATISTICS(t)
55:        return to INIT
56: upon ROLLBACK                                                   // for DU transactions
57:    UPDATEORACLESTATISTICS(t)
58:    stop executing r.code and return to INIT
59: upon RETRY                                                      // for DU transactions
60:    stop executing r.code
61:    UPDATEORACLESTATISTICS(t)
62:    raise TRANSACTION
63: procedure UPDATEORACLESTATISTICS(txDescriptor t)
64:    TransactionOracle.feed(t.stats)
```

Algorithm 4. Hybrid Transactional Replication for process p_i (part 2)

```
The main thread of HTR
65: txDescriptor t ← ⊥                                   // type: record (id, start, end, readset, updates, stats)
66: enum outcome ← ⊥                                     // type: enum {committed, rolledback, retry, success, failure}
67: response res ← ⊥
68: upon TO-DELIVER (txDescriptor t)
69:     outcome ← CERTIFY(t.start, t.readset)
70:     if outcome = success then
71:         lock { LC ← LC + 1
72:                for all (oid, obj) ∈ t.updates
73:                    add obj as a new version of object oid with version number LC }
74:     if transaction with t.id executed locally by thread q then
75:         pass outcome to thread q
76: upon TO-DELIVER (request r)
77:     raise TRANSACTION
78:     if request r handled locally by thread q then
79:         pass (outcome, res, t) to thread q
80: upon TRANSACTION                                     // for SM transactions
81:     t ← (a new unique id, 0, 0, ∅, ∅, ∅)
82:     lock { t.start ← LC }
83:     res ← execute r.code with r.args
84:     COMMIT()
85: upon READ(objectId oid)                              // for SM transactions
86:     return GETOBJECT(t, oid)
87: upon WRITE(objectId oid, object obj)                 // for SM transactions
88:     t.updates ← t.updates ∪ {(oid, obj)}
89: procedure COMMIT                                     // for SM transactions
90:     lock { LC ← LC + 1
91:            for all (oid, obj) ∈ p.updates
92:                add obj as a new version of object oid with version number LC }
93:     outcome ← committed
94:     return to TO-DELIVER
95: upon ROLLBACK                                        // for SM transactions
96:     outcome ← rolledback
97:     stop executing r.code and return to TO-DELIVER
98: upon RETRY                                           // for SM transactions
99:     outcome ← retry
100:    stop executing r.code and return to TO-DELIVER
```

8 Postgres-R

In the previous sections we explained that the great strength of the algorithms such as DUR (or MvDUR) is the fact that there is only one communication step for each transaction's run. However, there is no such thing as free lunch. DUR trades low communication latency for a high volume of data to be broadcast and transaction certification which has to be performed independently by each replica. In this section we present Postgres-R [12], an algorithm originally proposed for database replication, which appears similar to DUR but is able to compensate some of its limitations. Postgres-R has also been used in distributed TM [8]. Unlike in DUR, in Postgres-R no *readset* is broadcast after a transaction completes its execution. Also, in total, all processes perform less certification, thus saving resources. Postgres-R, however, requires an additional communication phase—a process that executed the transaction broadcasts to all replicas the final decision on whether to commit or abort the transaction. This additional broadcast is performed after the process broadcasts and delivers the updates produced by the transaction.

Algorithm 5. Postgres-R for process p_i (part 1)

```
1: integer LC ← 0
2: set AbortedTx ← ∅, DecidedTx ← ∅
3: function GETOBJECT(txDescriptor t, objectId oid)
4:     if (oid, obj) ∈ t.updates then
5:         value ← obj
6:     else
7:         lock { acquire read lock on oid for transaction t.id }
8:         value ← retrieve object oid
9:     return value
```

Thread q **on request** r **from client** c (executed on one replica)

```
10: txDescriptor t ← ⊥                          // type: record (process, id, start, end, updates)
11: response res ← ⊥
12: upon INIT
13:     wait until LC ≥ r.clock
14:     raise TRANSACTION
15:     return (r.id, LC, res) to client c
16: upon TRANSACTION
17:     t ← (p_i, a new unique id, 0, 0, ∅)
18:     lock { t.start ← LC }
19:     res ← execute r.code with r.args
20:     COMMIT()
21: upon READ(objectId oid)
22:     return GETOBJECT(t, oid)
23: upon WRITE(objectId oid, object obj)
24:     lock { acquire write lock on oid for transaction t.id }
25:     t.updates ← {(oid', obj') ∈ t.updates : oid' ≠ oid} ∪ {(oid, obj)}
26: procedure COMMIT
27:     if t.updates ≠ ∅ then
28:         TO-BROADCAST t
29:         wait for outcome
30:     lock { release all the locks held by transaction t.id }
31:     return to INIT
32: upon ROLLBACK
33:     lock { release all the locks held by transaction t.id }
34:     stop executing r.code and return to INIT
35: upon RETRY
36:     lock { release all the locks held by transaction t.id }
37:     stop executing r.code
38:     raise TRANSACTION
39: upon ABORT
40:     raise RETRY
```

Algorithm. The pseudocode for Postgres-R is given in Algorithm 5. Similarly to DUR, Postgres-R executes a transaction on copies of shared objects. Unlike DUR, however, Postgres-R does not maintain *readsets* for executed transactions and extensively relies on the read-write locks associated with each shared object (lines 7 and 24). The locks prevent live transactions from reading an inconsistent snapshot. In this sense, the locks fulfill the same function as the local certification procedure performed upon every read operation in DUR.

Once a transaction T finishes execution, the transaction's descriptor containing the process ID, the transaction ID, start timestamp and the updates that T produced, is broadcast to all replicas using TOB (line 28). Since the message does not contain *readset* (as in DUR), replicas cannot independently certify T. In Postgres-R certification happens somewhat indirectly and is driven by TOB. Similarly as in DUR, TOB is used to establish the serialization order on all (updating) transactions in the system.

Algorithm 5. Postgres-R for process p_i (part 2)

```
The main thread of Postgres-R (executed on all replicas)
41: upon TO-DELIVER (txDescriptor t) lock
42:     if t.id ∈ AbortedTx then
43:         return
44:     if transaction with t.id executed locally by thread q then
45:         outcome ← commit
46:         R-BROADCAST (t.id, outcome)
47:         DecidedTx ← DecidedTx ∪ {t.id}
48:         apply t.updates
49:         LC ← LC + 1
50:         pass outcome to thread q
51:     else
52:         for all (oid, obj) ∈ t.updates do
53:             if read or write lock acquired on oid by some transaction t'.id executed locally by thread q then
54:                 AbortedTx ← AbortedTx ∪ {t'.id}
55:                 R-BROADCAST (t'.id, abort)
56:                 raise ABORT on thread q
57:             enqueue write lock request on oid for transaction t.id
58: upon R-DELIVER(integer id, decision d) lock
59:     if id ∈ DecidedTx then                          // transaction executed locally
60:         return
61:     if d = commit then
62:         DecidedTx ← DecidedTx ∪ {id}
63:     else                                            // d = abort
64:         AbortedTx ← AbortedTx ∪ {id}
65:         release all the locks held by transaction id
66: upon GRANTED ALL LOCKS ENQUEUED FOR TRANSACTION t.id AND t.id ∈ DecidedTx lock
67:     apply t.updates
68:     release all the locks held by transaction t.id
69:     LC ← LC + 1
70: upon PROCESS p_j CRASH                  // reliable information from group membership mechanism
71:     lock { release all locks/dequeue all lock requests for transactions t.id such that t.process = p_j }
```

Transactions in committing state that are TO-Delivered preempt earlier transactions whose updates are not yet TO-Delivered. This is done in the following way: upon delivery of a new transaction descriptor (line 41) a replica tries to acquire write locks for every object in the update set on behalf of the incoming transaction; if the lock is held by a local transaction whose updates were not yet broadcast and delivered, the local transaction is aborted (lines 54–56) and its locks are released (line 36). At this point the replica is the sole process which has the knowledge about the outcome of this local transaction. Because the aborted transaction might have already broadcast its transaction descriptor, which other processes will eventually deliver, the replica needs to inform them of its decision to abort the transaction.[2] For this purpose the *reliable broadcast (RB)* is used (line 55). It is sufficient because decision messages do not need to be ordered.

If a committing transaction is not preempted and it gets to the point where it is TO-Delivered by the replica which initiated it, then the transaction can finally commit (lines 45–50). Similarly as in case of an aborted transaction, only one process knows about the outcome, so it has to inform others of the decision to commit (line 46). The next step is to apply the updates and increment *LC*. The updates can be applied straight

[2] If the aborted transaction was still in the executing phase, i.e. it did not reach the commit phase, then this step can be ignored. However, this optimization is not reflected in the pseudocode.

away, because in case of a local transaction we are sure that it holds the locks for every modified object since its execution phase.

The commitment of a foreign transaction (initiated by a different replica) is more complicated (lines 52–57). As previously stated, first, the transaction needs to acquire write locks for every updated item. If they are held by local (executing or committing) transactions, the local transactions need to be preempted. However, locks may also be held by other foreign committing transactions which wait to be committed. Therefore, the replica enqueues lock requests on behalf of the incoming transaction (line 57). The operation of acquiring locks and enqueuing lock requests for individual objects must be atomic and the lock requests need to respect FIFO order. Note, that three **upon** statements handled by the main Postgres-R thread feature a **lock** in its declaration (lines 41, 58 and 66) meaning the whole statement is guarded by a global lock. Therefore, all accesses to read/write locks in the pseudocode are protected from interleaving with each other. Besides acquiring the locks one more condition needs to be met for a transaction to be able to commit. The replica that initiated it must take the actual decision to commit it and then broadcast this decision. Only when the appropriate decision is R-Delivered (line 58) and all the required locks are granted (line 66) the process of committing can be finished (lines 67–69). Naturally, if the R-Delivered decision is to abort, then the waiting transaction is dropped and all the locks it managed to acquire are released (line 65).

Sometimes the decision message for some transaction T may arrive at some process before the message with T's transaction descriptor. Postgres-R, therefore, maintains two sets *AbortedTx* and *DecidedTx*, so it knows whether to apply or drop the updates once they arrive. Now, consider a scenario in which the decision message for a transaction T never arrives because of a replica crash. In such a case, every replica would indefinitely hold locks for all objects modified by T. It is easy to show that a simple timeout-based mechanism running independently on each replica is not sufficient. Therefore, replicas need to abort such transactions in a coordinated fashion. For this purpose, Postgres-R utilizes *group communication services*. Whenever processes leave (because of failures or shutdowns) or join (recovering processes), the group communication module creates different views in the computation. A view gives an illusion of a stable configuration consisting of only operational processes. All messages sent within a view are confined to that view.

In case of failures, upon a view change, we can identify active transactions originating at the failed site and we can safely abort them (line 71) without compromising consistency of the non-faulty processes. Even if the crashed replica has broadcast a commit decision just before the crash, this message will not be delivered to any of the processes. This is because a new view is established, and all the messages from previous view were already delivered or are discarded.

Discussion. As described above, the process of certification in Postgres-R is somewhat indirect. Incoming transactions, whose order is established with TOB, invalidate live transactions that are local to specific replicas. Therefore, the certification is distributed and replicas need additional synchronization to disseminate the result of certification. Instead of certifying each transaction directly, the processes have to rely on others to broadcast the final decision in a second phase. The additional broadcast greatly

increases the latency of a transaction's commit. It means that the concurrent transactions may have to wait significant amount of time on locks held by a transaction waiting for the decision message. In turn, these concurrent transactions are more prone to abort induced by transactions executed by other replicas. One can see, then, that even low contention is problematic to Postgres-R. It seems, therefore, that Postgres-R is not suitable for transactional replication where transactions are usually short and access few objects but may conflict often. One has to remember, though, that Postgres-R was originally designed for database systems, not distributed TM.

So what types of workloads does Postgres-R handle well? Transactions in Postgres-R have to be long and access many objects. Only then the potential gains that stem from not having to broadcast *readset* (as in DUR) are worth the cost of an additional communication phase.

9 Executive Deferred Update Replication

In this section we present yet another DUR-based algorithm, called *Executive Deferred Update Replication (EDUR)* [15]. The key idea behind EDUR lies in an observation regarding some distributed agreement protocols, such as Paxos. These algorithms feature a distinguished process, *the leader*, which is responsible for coordination of message broadcast. It means that a message broadcast by some process is first received by the leader who, essentially, stamps it with a sequence number before sending it to the rest of the processes. This way each process knows the final message delivery order. Since all messages pass through the leader, we can use the leader to perform some additional work before it forwards the messages to the rest of the replicas. In particular, EDUR uses the leader to certify transactions on behalf of all replicas. Streamlining transaction certification with the broadcast protocol has several advantages. Firstly, certification is performed only by one process, not by all process as in DUR. Secondly, the network traffic is greatly reduced which can be explained as follows. Once a transaction is certified successfully, only the set containing the updates resulting from transaction execution has to be forwarded to all replicas. The often large readset required for transaction certification is no longer needed. In case a transaction fails certification, the leader only needs to inform the process that executed the transaction that it has to be restarted. Finally, unlike Postgres-R, EDUR does not increase the number of communication steps for each transaction's run. It means that EDUR can be implemented efficiently.

It is worth to note that the load of the leader in EDUR not only does not increase compared to DUR but even can be lower. Both in DUR and EDUR the leader certifies transactions but in the latter case the certification procedure occurs earlier and the size of messages broadcast is often much smaller, which attribute to lower load.

Let us focus for a while on a broadcast protocol that serves as a base for EDUR. It turns out that it is insufficient to simply extend this protocol so that the leader executes some routine before a message is forwarded to the rest of the processes. It is because the leader, by processing the messages and possibly changing their content, establishes a prefix order on the sequence of messages it sends. In other words, the messages which were concurrently issued by different replicas and pass through the leader are no longer independent with regard to each other. Any message m that appears later in the sequence

is logically dependent on any message m' that appears in the sequence prior to m.[3] This would not be problematic if the leader coordinated only one consensus instance at a time. Then, a new transaction did not undergo a certification until the message regarding the previously certified transaction would not be delivered by the leader (in the total order broadcast sense). This way, upon leader change, the new leader would be aware of all transactions certified by the previous one, thus preserving consistency. However, for performance reasons, TOB protocols such as Paxos allow for concurrent processing of several consensus instances. This means that a different solution is required.

In [15], we point out that it would be possible to build EDUR on top of Extended Virtual Synchrony (EVS) [22]. In EVS, processes are organized within groups of processes that maintain dynamic views of processes that are considered to be operational. As noted in Section 8, a process view gives an illusion of a stable group configuration consisting of only correct processes that never crash. Whenever a process is suspected to have crashed or voluntarily joins or leaves the group a new view is formed. Messages sent within a view are confined to that view. It is therefore possible to safely elect some process in each view and make it responsible for transaction certification. However, EVS limits the performance of EDUR in several ways. Most importantly, EVS requires a system to pause computation upon every view installation event. The overhead should not be noticeable if views do not change often. Unfortunately, a new view has to be installed every time any process begins to be suspected of a failure by any other process from the same group. If a group is large such a situation can be a commonplace. For these reasons, EDUR uses a new broadcast protocol called *Executive Order Broadcast (EOB)*.

Below we characterize EOB informally (see [15] for a formal specification). EOB extends TOB in two aspects. Firstly, EOB introduces a number of new primitives that allow the programmer to define actions to be undertaken by the leader before a message is forwarded to the rest of the replicas (see below). Secondly, in EOB the total order property of TOB is substituted by the *executive order*. This property guarantees that not only all messages are delivered by each replica in the same order but also it ensures that the prefix order imposed by the leader is always preserved. The definition of EOB accounts for multiple concurrent leaders, so it is possible to devise an EOB-enabled algorithm similar to Paxos. In fact, the implementation of EOB in [15] is based on Paxos.

Let us review the primitives and events of EOB. EO-BROADCAST(id, mc) and EO-DELIVER(id, mc') correspond to the ones of TOB. In addition they account for the fact that the content *mc* of the broadcast message can be changed by the leader. Therefore, the unique identifier *id* is used to distinguish between messages. The next four primitives are characteristic for EOB: EO-LEADERELECT and EO-LEADERRECALL are used by a local failure detector to inform the process that it has to, respectively, take on or relinquish the duties of the leader process (and we say that during the time periods between these events the process *is a leader*). A leader receives EO-LEADERDELIVER(id, mc)

[3] Naturally, all messages issued by replicas as a result of processing requests from the same client form a sequence of logically dependent messages. However, a client cannot issue a new request, until the previous one returns.

events, so it can process the incoming messages. To broadcast a (possibly) modified message, the leader invokes the EO-LEADERBROADCAST(id, mc') primitive.

When the leader promptly forwards all messages that it received through the EO-LeaderDeliver events, with no additional action, EOB is reduced to TOB. In fact, EOB is strictly stronger than TOB. One can also show that EOB is strictly weaker than EVS. It is because, unlike EVS, EOB does not feature the group membership service. Most importantly, however, under stable conditions, EOB can operate as efficiently as TOB but, unlike EVS, it requires reconfiguration only when the current leader is suspected to have crashed (groups in EVS are reconfigured each time any process is suspected).

Algorithm. Once we understand how EOB works, we can describe pseudocode for EDUR, given in Algorithm 6. It is based on MvDUR presented in Section 6. The most apparent difference between MvDUR and EDUR lies in the fact that EDUR features a *leader thread* running on each replica (lines 54–78). During the time between the EO-LeaderElect and EO-LeaderRecall events (lines 62 and 68), the thread performs transaction certification on behalf of other replicas (line 72). Note that, the EO-LEADERELECT primitive takes as an argument $initialHistory$. It is an ordered set which represents the initial (unreliable) knowledge of the leader about the EO-Broadcast but not yet EO-Delivered transaction descriptors. The order in $initialHistory$ is consistent with the order in which the transaction descriptors were TO-LeaderBroadcast by previous leaders and in which they will most probably be TO-Delivered soon. It allows the leader to start certifying incoming transactions as soon as possible, i.e. without waiting for the appropriate EO-Deliver events. In case the set contains incorrect information, e.g., it does not include a transaction successfully certified by the previous leader, which was agreed on by majority of processes, EOB guarantees to invalidate all decisions made by the new leader, thus preventing any inconsistencies.[4] The leader thread maintains its own tentative logical clock *TLC*, which is incremented every time a new transaction descriptor is EO-LeaderDelivered and the transaction is successfully certified (line 73). The information about successfully certified transactions that are not yet EO-Delivered is stored in the *ProcessedTx* set.

The certification procedure performed by the leader (lines 54–61) is a bit different from the standard one, featured in MvDUR, and also used in EDUR for local transaction certification (lines 14–19). It is because each transaction T needs to be certified by the leader also against all transactions T' which are (a) concurrent with respect to T, (b) have been successfully certified by the leader, and (c) are not yet EO-Delivered (line 57). After the certification, the transaction descriptor is transformed before it is EO-LeaderBroadcast. Since the certification is already performed, *readset* is no longer needed. Moreover, if the transaction failed certification, the $updates$ set also need not be broadcast. In such a case, only the transaction identifier is included in the forwarded message, so that the replica that executed the transaction knows to restart it.[5]

[4] In this sense, the EOB primitives give a leader an impression of being the sole leader in the system, capable of making authoritative decisions on behalf of the rest of the processes. Obviously, this makes the work of the programmer much easier.

[5] In fact, only a unicast message would suffice in such circumstances. This optimization, however, would require extending EOB with new primitives, thus making the protocol unjustifiably more complicated [15].

Algorithm 6. Executive Deferred Update Replication for process p_i (part 1)

```
1: integer LC ← 0, TLC ← 0
2: set ProcessedTx ← ∅
3: boolean IsLeader ← false
4: function GETVERSION(objectId oid, integer notNewerThan)
5:     lock { return (obj, ver) such that obj is a version of object oid whose version number ver
6:                              is the highest available such that ver ≤ notNewerThan }
7: function GETOBJECT(txDescriptor t, objectId oid)
8:     if (oid, obj) ∈ t.updates then
9:         value ← obj
10:    else
11:        (obj, ver) ← GETVERSION(oid, t.start)
12:        value ← obj
13:    return value
14: function CERTIFY(integer start, set readset)
15:    for all id ∈ readset do
16:        (obj, ver) ← GETVERSION(id, ∞)
17:        if ver > start then
18:            return failure
19:    return success
```

```
Thread q on request r from client c (executed on one replica)
20: txDescriptor t ← ⊥                          // type: record (id, start, end, readset, updates)
21: response res ← ⊥
22: upon INIT
23:    wait until LC ≥ r.clock
24:    raise TRANSACTION
25:    return (r.id, LC, res) to client c
26: upon TRANSACTION
27:    t ← (a new unique id, 0, 0, ∅, ∅)
28:    lock { t.start ← LC }
29:    res ← execute r.code with r.args
30:    COMMIT()
31: upon READ(objectId oid)
32:    obj ← GETOBJECT(t, oid)
33:    if r.readOnly = false then
34:        t.readset ← t.readset ∪ {oid}
35:    return obj
36: upon WRITE(objectId oid, object obj)
37:    t.updates ← {(oid', obj') ∈ t.updates : oid' ≠ oid} ∪ {(oid, obj)}
38: procedure COMMIT
39:    if t.updates = ∅ then
40:        return to INIT
41:    if CERTIFY(t.start, t.readset) = failure then
42:        raise RETRY
43:    EO-BROADCAST t
44:    wait for outcome
45:    if outcome = failure then
46:        raise RETRY
47:    else                                       // outcome = success
48:        return to INIT
49: upon ROLLBACK
50:    stop executing r.code and return to INIT
51: upon RETRY
52:    stop executing r.code
53:    raise TRANSACTION
```

Algorithm 6. Executive Deferred Update Replication for process p_i (part 2)

```
The leader thread of EDUR (executed on all replicas)
54: function LEADERCERTIFY(integer start, set readset)
55:     if CERTIFY(start, readset) = failure then
56:         return failure
57:     lock { conflictingTx ← {(id, updates, clock) ∈ ProcessedTx :
58:                                 clock > start ∧ ∃(oid, obj) ∈ updates : oid ∈ readset} }
59:     if conflictingTx = ∅ then
60:         return success
61:     return failure
62: upon EO-LEADERELECT (ordered set initialHistory) lock
63:     TLC ← LC
64:     for all t ∈ initialHistory : t.updates ≠ ∅ do
65:         TLC ← TLC + 1
66:         ProcessedTx ← ProcessedTx ∪ {(t.id, t.updates, TLC)}
67:     IsLeader ← true
68: upon EO-LEADERRECALL lock
69:     IsLeader ← false
70:     ProcessedTx ← ∅
71: upon EO-LEADERDELIVER(txDescriptor t)
72:     if LEADERCERTIFY(t.start, t.readset) = success then
73:         TLC ← TLC + 1
74:         lock { ProcessedTx ← ProcessedTx ∪ {(t.id, t.updates, TLC)} }
75:     else
76:         t.updates ← ∅
77:     t.readset ← ∅
78:     EO-LEADERBROADCAST t

The main thread of EDUR (executed on all replicas)
79: upon EO-DELIVER(txDescriptor t)
80:     if updates ≠ ∅ then
81:         outcome ← success
82:         lock { if IsLeader = true then
83:                     ProcessedTx ← {(id, updates, clock) ∈ ProcessedTx : id ≠ t.id}
84:                 LC ← LC + 1
85:                 for all (oid, obj) ∈ t.updates
86:                     add obj as new version of object oid with version number LC }
87:     else
88:         outcome ← failure
89:     if transaction with t.id executed locally by thread q then
90:         pass outcome to thread q
```

The rest of the pseudocode of EDUR is very similar to MvDUR's. In fact, the execution phase of EDUR differs from MvDUR only in using EOB to broadcast messages (line 43). Naturally, in EDUR processes do not perform certification upon delivering the message (line 79). Instead, they only update their state if the transaction successfully passed certification (lines 81–86).

Discussion. It is easy to see why EDUR introduces no inconsistencies during stable periods, i.e., when a leader process does not change. All messages pass through the leader which certifies, transforms and finally forwards them to all processes. The leader does not wait for a transaction it successfully certified to be committed before it certifies other transactions. It means that implicit order on message delivery is introduced. Since the leader does not change, each process EO-Delivers messages in the order the leader sent them. The consistency is therefore preserved. During unstable periods the consistency is preserved as well. It is because EOB makes sure that the prefix order established on the messages EO-LeaderBroadcast by the leader is always respected, even when the leader changes. The system performance during the leader transition periods is comparable to DUR's since in EOB the changes of the leader occur smoothly (thanks

to *initialHistory* passed to EO-LEADERELECT and the fact that multiple concurrent leaders are allowed). In fact, the new leader starts just when the old one is suspected to have crashed, and not only after a distributed agreement is reached to elect a new leader or establish a new view.

Having only one process to certify the transactions enables us to devise all kinds of interesting optimizations, not possible with standard DUR/MvDUR [15]. One of the most interesting involves using a multithreaded certification procedure to improve the throughput of the leader.

10 Comparison

In Table 1, we compare replication algorithms discussed in this chapter, looking at their selected features and performance characteristics. We excluded DUR with multiversioning (MvDUR). This powerful optimization technique boosts DUR's performance but does not change the characteristics of DUR in any aspect that we consider in our comparison. Below we discuss and explain our results.

Semantics. All discussed replication algorithms (except SMR) support full transactional semantics, so the programmer can use additional constructs to manage the flow of control, such as *abort* and *retry* (and possibly also *commit*). In DUR, Postgres-R and EDUR, a transaction is always executed optimistically. Therefore, these algorithms do not support irrevocable operations. Naturally, requests executed with SMR may include irrevocable operations, because SMR always executes all (updating) requests sequentially. Similarly, abort-free execution of irrevocable transactions is guaranteed in HTR for transactions executed in the SM mode. Additionally, DUR, HTR, Postgres-R and EDUR guarantee abort-free execution of read-only transactions if only they support multiversioning.

Complexity. We consider three aspects in the quantitative evaluation of the algorithms. Firstly, we compare the overhead due to the used concurrency control mechanisms. All replication schemes featuring transactional semantics require some additional computation steps and data structures, which result in some extra overhead during request processing. DUR, HTR, Postgres-R and EDUR do not update the accessed shared objects directly. Instead, the updates are performed on copies of shared objects and stored in the *updates* set. Additionally, DUR, HTR in DU mode and EDUR maintain *readset* containing object IDs of all shared objects read by the transaction.[6] Postgres-R does not maintain *readset* but acquires locks on accessed shared objects. Similarly, all algorithms but SMR feature a transaction certification phase. Depending on the algorithm, certification is performed by all replicas (DUR, HTR in DU mode), by all replicas but the one that executed the transaction (Postgres-R) or by a single replica (EDUR). Transaction certification differs between the algorithms. Its complexity depends either on the size of *readset* (DUR, HTR in DU mode and EDUR) or *updates* (Postgres-R).

[6] *Readset* does not need to be maintained for read-only transactions.

Table 1. Comparison of transactional replication schemes

	SMR	**DUR**	**HTR**	**Postgres-R**	**EDUR**
Semantics:					
- control flow management	*no*	*yes*			
- support for irrevocable operations	*yes*	*no*	*yes*	*no*	*no*
Overhead due to concurrency control	*none*	*tracking accesses to shared objects, writes performed on object copies* [a]			
Commit-time transaction certification:					
- number of times performed	0	n	n or 0 [a]	$n-1$	1
- complexity	n/a	$O(\|readset\|)$	$O(\|readset\|)$ or n/a [a]	$O(\|updates\|)$	$O(\|readset\|)$
Number of communication steps	3 (TOB)	3 (TOB)	3 (TOB)	5 ($TOB+RB$)	3 (EOB)
Number of network transmissions:					
- client's request message	n	1	1 or n [a]	1	1
- transaction readset	0	n	n or 0 [a]	0	1
- transaction updates	0	n	n	n	n or 0 [b]
Sensitivity to a workload type:					
- high contention	*none*	*high*	*medium*	*very high*	*medium*
- CPU intensive workload	*high*	*low*	*low*	*low*	*low*
- many read operations	*none*	*high*	*medium*	*low*	*low*

[a] Depends on the transaction execution mode (HTR only).
[b] Depends on the outcome of transaction certification.

Secondly, we compare the number of communication steps per transaction run which are required for replica synchronization. Naturally, the least number of communication steps is two: the processes send data in the first phase and, to ensure reliable communication, exchange acknowledgments in the second phase. Additionally, if the order of messages is important, the message needs to be first forwarded to the leader/sequencer process which then orders and broadcasts it. Thus under stable conditions two broadcast protocols featured in this chapter, i.e. TOB and EOB, require three communication steps, and third one, RB (reliable broadcast), requires only two. Hence, for each transaction's run SMR, DUR, HTR and EDUR need three communication steps while Postgres-R needs five communication steps.

Thirdly, we check the amount of data that replicas need to exchange in order to synchronize. Typically, SMR requires the least data to be transferred. It is because SMR broadcasts only the request's code and data needed for request execution. On the other hand, other algorithms require to broadcast the updates resulting from the local request execution, and usually some metadata that are necessary for transaction certification. Of course, when using the SM execution mode in HTR, the amount of data needed to be broadcast is the same as in SMR. EDUR reduces the network traffic by performing certification only on one process—this reduction is particularly significant in case of transactions that failed certification.

Finally, we compare three different types of workloads and discuss how they influence the performance and scalability of the algorithms. Replication schemes featuring optimistic concurrency control typically do not tolerate high contention well (i.e., when multiple concurrent requests access the same data). It is because under such workloads many transactions are rolled back and restarted, thus wasting resources. This type of workload is particularly troublesome for Postgres-R because it requires two broadcasts to be performed for each transaction's run. In HTR and EDUR, the negative aspects of high contention can be compensated. HTR allows for transaction execution with abort-free guarantees thus reducing the overall contention. In EDUR, conflict detection is streamlined with message broadcast, thus reducing the total amount of computation and the volume of data transferred through the network. Moreover, other processes do not need to bother with processing transactions that failed certification. On the other hand, in SMR, no conflicts can occur, because all (updating) requests are executed sequentially. However, for the same reason, SMR is not suitable for CPU intensive workloads. On the contrary, DUR, HTR, Postgres-R and EDUR perform better under CPU intensive workloads because they allow for the concurrent execution of all requests, not necessarily the read-only ones.

DUR does not handle well requests that execute multiple read operations. It is because DUR gathers the information about read objects in *readset* and later broadcasts it alongside *updates* to all replicas. Large *readsets* put strain on the network stack and so limit the system's scalability. In HTR, a transaction accessing a large number of objects can be executed in the SM mode (thus no *readset* need to be broadcast). Such a workload is also not problematic for EDUR or Postgres-R as well (in EDUR *readset* is only sent to the leader process; in Postgres-R replicas do not exchange any information about

objects read by transactions). On the other hand, the type of operations (read/write) executed within a request does not influence the performance of SMR because it does not feature transactional semantics.

11 Conclusion and Further Reading

In this chapter, we studied distributed algorithms for full transactional replication. We defined the properties of transactional replication in terms of the rules that define the replicated process as well as the interaction between the replicated process and external clients. Then we described and discussed several core algorithms. They included basic schemes, such as state machine replication (SMR) and deferred update replication (DUR), as well as optimized variants that use multiversioning (MvDUR), combine SMR and DUR (HTR), optimize broadcast data (Postgres-R), and optimize the broadcast protocol itself (EDUR).

We then compared their main features and complexity, taking into account concurrency control, computation overhead, network communication overhead, and the application workload type. One can see from this comparison that there is no one solution that fits all purposes. The results of experimental evaluation (see e.g., [37,14]) show that a simple scheme such as SMR performs surprisingly well compared to DUR, even though it provides limited parallelism. However, the optimizations of DUR make it a lot more viable, especially given its full transactional semantics which basic SMR lacks.

We only presented selected SMR and DUR-like algorithms whose main feature is that they all rely on the total order broadcast to serialize the execution of transactions or state updates. There exist many other transactional replication methods and algorithms that differ in a number of ways, e.g., they use pessimistic concurrency control or speculative executions, build the replication protocols on top of non-distributed transactional memory, or explore other models of data space and failure. Below we give some example references to the recent work that is close to the work discussed in this chapter, but they are by no means complete.

Romano, Palmieri, Quaglia, Carvalho, and Rodrigues [31] (see also [24]) explore speculative replication protocols for transactional systems. The key idea is to run an *optimistic atomic broadcast (OAB)* algorithm to provide an early, possibly erroneous, guess on transactions' serialization order, in parallel with the algorithm that is used to determine the actual order.

Marandi, Primi, and Pedone [21] optimize the SMR scheme by using speculative execution to reduce the response time and state partitioning to increase the throughput of SMR. In the follow-up paper [19], the authors propose *parallel state-machine replication (P-SMR)*, which optimizes SMR by exploiting service semantics to determine when commands can execute concurrently and when serial execution is needed (see also [20], where a more aggressive speculative strategy is used).

Arun, Hirve, Palmieri, Peluso, and Ravindran [3] observe that in DUR even in case when remote transactions rarely conflict with each other, the conflicts among local transactions (on the same replica) can significantly decrease performance. They explore speculation to optimize this scenario and prevent some local transactions from aborting each other.

Sciascia, Pedone, and Junqueira [36] propose *scalable deferred update (S-DUR)* aimed at increasing scalability of DUR through optimizing the execution of update transactions. The key idea is to divide the state into logical partitions, replicate each one among a group of servers, and orchestrate the execution and termination of transactions across partitions using a 2PC-like protocol. Pacheco *et al.* [23] build on this idea to scale DUR on multicore processors.

In [35], Sciascia and Pedone research the application of DUR to geo-replicated storage systems. The paper discusses two optimizations of DUR for geo-replication which essentially explore delaying and reordering of transactions.

Some researchers investigated transactional replication algorithms considering complex failure models, in which servers mail fail arbitrarily. For example, Pedone and Schiper [28] discuss DUR under Byzantine faults and propose suitable extensions of this replication scheme in this failure model.

Acknowledgements. This work was funded from National Science Centre funds granted by decision No. DEC-2012/06/M/ST6/00463.

References

1. Agrawal, D., Alonso, G., Abbadi, A.E., Stanoi, I.: Exploiting atomic broadcast in replicated databases (extended abstract). In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 496–503. Springer, Heidelberg (1997)
2. Aguilera, M.K., Chen, W., Toueg, S.: Failure detection and consensus in the crash-recovery model. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 231–245. Springer, Heidelberg (1998)
3. Arun, B., Hirve, S., Palmieri, R., Peluso, S., Ravindran, B.: Speculative client execution in deferred update replication. In: Proc. of MW4NG 2014: The 9th Middleware for Next Generation Internet Computing Workshop (December 2014)
4. Bernstein, P.A., Goodman, N.: Multiversion concurrency control—theory and algorithms. ACM Transactions on Database Systems (TODS) 8(4), 465–483 (1983)
5. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming. Springer (2011)
6. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM (JACM) 43(4), 685–722 (1996)
7. Charron-Bost, B., Pedone, F., Schiper, A. (eds.): Replication - Theory and Practice. LNCS, vol. 5959. Springer, Heidelberg (2010)
8. Couceiro, M., Romano, P., Rodrigues, L.: Polycert: Polymorphic self-optimizing replication for in-memory transactional grids. In: Proc. of Middleware 2011: The 12th ACM/IFIP/USENIX International Conference on Middleware (December 2011)
9. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys (CSUR) 36(4), 372–421 (2004)
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) 32(2), 374–382 (1985)
11. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: Proc. of SIGMOD 1996: The ACM SIGMOD International Conference on Management of Data (June 1996)
12. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In: Proc. of VLDB 2000: The 26th International Conference on Very Large Data Bases (September 2000)

13. Kemme, B., Pedone, F., Alonso, G., Schiper, A.: Processing transactions over optimistic atomic broadcast protocols. In: Proc. of ICDCS 1999: The 19th IEEE International Conference on Distributed Computing Systems (1999)
14. Kobus, T., Kokociński, M., Wojciechowski, P.T.: Hybrid replication: State-machine-based and deferred-update replication schemes combined. In: Proc. of ICDCS 2013: The 33rd IEEE International Conference on Distributed Computing Systems (July 2013)
15. Kokociński, M., Kobus, T., Wojciechowski, P.T.: Make the leader work: Executive deferred update replication. In: Proc. of SRDS 2014: The 33rd IEEE International Symposium on Reliable Distributed Systems (October 2014)
16. Kończak, J., Santos, N., Żurkowski, T., Wojciechowski, P.T., Schiper, A.: JPaxos: State machine replication based on the Paxos protocol. Tech. Rep. EPFL-REPORT-167765, Faculté Informatique et Communications, EPFL (July 2011)
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM (CACM) 21(7), 558–565 (1978)
18. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems (TOCS) 16(2) (May 1998)
19. Marandi, P.J., Bezerra, C.E., Pedone, F.: Rethinking state-machine replication for parallelism. In: Proc. of ICDCS 2014: The 34th IEEE International Conference on Distributed Systems, pp. 368–377 (June 2014)
20. Marandi, P.J., Pedone, F.: Optimistic parallel state-machine replication. In: Proc. of SRDS 2014: The 33rd International Symposium on Reliable Distributed Systems (October 2014)
21. Marandi, P.J., Primi, M., Pedone, F.: High performance state-machine replication. In: Proc. of DSN 2011: The 41st IEEE/IFIP International Conference on Dependable Systems and Networks (June 2011)
22. Moser, L.E., Amir, Y., Melliar-Smith, P.M., Agarwal, D.A.: Extended virtual synchrony. In: Proc. of ICDCS 1994: The 14th International Conference on Distributed Computing Systems (June 1994)
23. Pacheco, L., Sciascia, D., Pedone, F.: Parallel deferred update replication. In: Proc. of NCA 2014: The 13th IEEE International Symposium on Network Computing and Applications (August 2014)
24. Palmieri, R., Quaglia, F., Romano, P.: OSARE: Opportunistic speculation in actively REplicated transactional systems. In: Proc. of SRDS 2011: The 30th IEEE International Symposium on Reliable Distributed Systems (October 2011)
25. Palmieri, R., Quaglia, F., Romano, P., Carvalho, N.: Evaluating database-oriented replication schemes in software transactional memory systems. In: The 15th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (April 2010)
26. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distributed and Parallel Databases 14(1) (July 2003)
27. Pedone, F., Guerraoui, R., Schiper, A.: Exploiting atomic broadcast in replicated databases. In: Pritchard, D., Reeve, J.S. (eds.) Euro-Par 1998. LNCS, vol. 1470, pp. 513–520. Springer, Heidelberg (1998)
28. Pedone, F., Schiper, N.: Byzantine fault-tolerant deferred update replication. Journal of the Brazilian Computer Society 18, 3–18 (2012)
29. van Renesse, R.: Paxos made moderately complex, available electronically (2012)
30. Romano, P., Carvalho, N., Rodrigues, L.: Towards distributed software transactional memory systems. In: Proc. of LADIS 2008: The 2nd Workshop on Large-Scale Distributed Systems and Middleware (September 2008)
31. Romano, P., Palmieri, R., Quaglia, F., Carvalho, N., Rodrigues, L.: On speculative replication of transactional systems. Journal of Computer and System Sciences 80(1), 257–276 (2014)

32. Schiper, A., Raynal, M.: From group communication to transactions in distributed systems. Communications of the ACM (CACM) 39(4) (April 1996)
33. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) 22(4), 299–319 (1990)
34. Schneider, F.B.: Replication management using the state-machine approach, pp. 169–197. ACM Press/Addison-Wesley (1993)
35. Sciascia, D., Pedone, F.: Geo-replicated storage with scalable deferred update replication. In: Proc. of DSN 2013: The 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (June 2013)
36. Sciascia, D., Pedone, F., Junqueira, F.: Scalable deferred update replication. In: Proc. of DSN 2012: The 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (June 2012)
37. Wojciechowski, P.T., Kobus, T., Kokociński, M.: Model-driven comparison of state-machine-based and deferred-update replication schemes. In: Proc. of SRDS 2012: The 31st IEEE International Symposium on Reliable Distributed Systems (October 2012)