Supporting Linearizable Semantics in Replicated Databases*

Luís Rodrigues
INESC-ID/IST
ler@ist.utl.pt

Nuno Carvalho
INESC-ID/IST
nonius@gsd.inesc-id.pt

Emili Miedes
U. Politecnica de Valencia
emiedes@iti.upv.es

Abstract

This paper proposes a novel database replication algorithm that offers strong consistency (linearizable semantics) and allows reads and non-conflicting writes to execute in parallel in multiple replicas. The proposed algorithm supports the use of quorums to trade the availability/efficiency of read and write operations, making a bridge between consensus-based and quorum based solutions for database replication. Furthermore, the algorithm offers better performance for linearizable read-only transactions with a negligible impact on write transactions.

1 Introduction

Database replication is an important technique to increase the availability of data. Furthermore, by allowing read-only and non-conflicting write transactions to execute in parallel in multiple copies, database replication can also improve the performance of a distributed database system. Not surprisingly, the cost of ensuring the consistency of the database replicas in the presence of concurrent writes is one of the bottlenecks of replication algorithms.

*This work was partially supported by project Pastramy (PTDC/EIA/72405/2006). Selected sections of this report will be published in the Proceedings of the 7th IEEE International Symposium on Network Computing and Applications (IEEE NCA08), Cambridge, MA, USA, July 2008. (short paper)

In most cases, the weaker the consistency model one tries to enforce, the more efficient is the algorithm that supports that model. Due to this reason, many database management systems only enforce snapshot isolation [3], which is weaker than serializability [2]. Still, even serializability allows for reads to be executed "in the past".

Linearizable [14] or atomic semantics is a stronger consistency criteria that ensures that all transactions appear to execute atomically at a single instant in time and, furthermore, if a transaction T_y is started after transaction T_x commits, T_y is ensured to be linearized after T_x. Given that atomic semantics are needed in several application areas, this raises the interesting question of whether is possible to design replication algorithms that enforce such strong semantics and allow one to use quorums to trade the availability of writes with the efficiency of reads.

This paper proposes a replication protocol that combines the consensus-based replication [15, 20, 17] and the quorum-based replication approaches [10, 9] to achieve the goal above. Our protocol has the advantages of: *i*) not enforcing read operations to contact a write quorum of replicas in order to commit (namely, readers are not required to "write-back" the value that has been read nor to be serialized by an uniform total order primitive); *ii*) not slowing down the write operations; *iii*) allow reads and non-conflicting writes to execute in parallel in different replicas. Besides having practical relevance, our proposal makes a bridge between consensus-based and quorum based solutions for database replication.

The rest of the paper is structured as follows. Section 2 motivates our work and surveys the related work. Our replica-consistency algorithm is described in Section 3 and its performance is discussed in Section 4. Section 5 concludes the paper.

2 Related Work

2.1 Consistency Criteria for Databases

In the paper, we are concerned with the two stronget consistency criteria for concurrent databases, namely serlializability [2] and linearizability [14], that can be defined as follows:

Serializability The serializability property property can be defined on a history of transactions. A history is serializable if (*i*) its invocations and responses can be reordered to yield a sequential history, (*ii*) that sequential history is correct according to the sequential definition of the data. Note

that if each transaction is correct by itself, then any serial execution (at any transaction order) of these transactions is correct. As a result, any execution that is equivalent (in its outcome) to a serial execution, is correct.

Linearizability The linearizable property can be defined on a history of transactions. A history is linearizable if (*i*) its invocations and responses can be reordered to yield a sequential history, (*ii*) that sequential history is correct according to the sequential definition of the data and (*iii*) if a response preceded an invocation in the original history, it must still precede it in the sequential reordering. Note that the first two points here match serializability: the operations appear to happen in some order. It is the last point which distinguishes linearizability from serializability.

2.2 Replica Consistency Algorithms

Multiple algorithms have been proposed to manage the consistency of replicated data. These can be classified according to many different criteria. In this paper, we classify previous work according to the following aspects of replica consistency algorithms:

- If they are only suitable to replicate individual data item (also known as *registers*) or are also suitable for replicating collection of data item (such as databases) accessed by transactions. Transaction are defined as atomic sequences of multiple read and/or write operations.
- If they use quorums or consensus as the main technique to ensure consistency.
- If they require the use of locking at each individual replica.

We start by considering algorithms to replicate individual registers. Two main approaches can be used to provide atomic (or linearizable) semantics: quorum based and consensus based. Furthermore, quorum-based approaches can be non-blocking or rely on locking to ensure consistency. Each of these approaches is discussed below. Finally, we discuss how the algorithms used to replicate individual registers can be extended to replicate databases accessed by transactions.

2.3 Non-Blocking Quorum-Based Atomic Register

The non-blocking quorum-based approach operates as follows [12]. Let Ω be the set of replicas of the register. There are two types of operations

associated with the register: read and write operations. To execute an operation, not all replicas need to participate. We assume that two types of quorum sets are defined, *write quorums* and *read quorums* such that read quorums intersect with write quorums, and write quorums intersect with each other. Furthermore, each replica maintains a version number associated with the register; every-time the value of the register is updated, the version number is increased. The reader must be aware that the description of the algorithm is necessarily concise. For a more detailed description see, for instance, [12].

A write operation is initiated by multicasting a ReadRequest and waiting for a ReadReply(v, t) from a read quorum of replicas. The ReadReply(v, t) includes the value v and version number t of the replica. Then, a new version number is created by incrementing the largest version number from the replies. Subsequently, a WriteRequest(v', t') is multicast with the new value and version number. When a WriteRequest(v', t') is received, a replica replies with a WriteReply and updates its local value (and version) if the version is larger than the version stored locally. The operation concludes when a WriteReply has been received from a write quorum of replicas.

The read operation is initiated by multicasting a ReadRequest and waiting for a ReadReply(v, t) from a read quorum of replicas. The value to be returned is the one associated with the largest version number from all the values received. However, before returning, the read operation proceeds by *writing back* the value to be returned, by exchanging WriteRequest(v', t') and WriteReply with a write quorum of replicas as described above for the write operation.

A drawback of the write back phase is that it forces every read operation to be as expensive as a write operation (in fact, both a read and a write quorum of replicas are required for both operations). One may wonder if there are non-blocking solutions (i.e., that do not resort to some form of locking) that can avoid expensive read operations (i.e., to allow *fast reads*). In [8] it is shown that non-blocking fast reads are only possible in the case of *single writer* registers.

2.4 Consensus-Based Atomic Register

This solution relies on the use of an atomic broadcast [13] primitive which, in turn, can be implemented on top of a consensus primitive [19]. In simple terms, an atomic broadcast primitive ensures that all correct processes receive the same messages in exactly the same order (see Section 3.1 for a

detailled definition). Atomic broadcast can be used to implement any form of deterministic object using a technique that has been coined the *state-machine* approach [18].

An atomic register can be implemented using the state-machine approach as follows. Both write and read operations are implemented by (atomically) broadcasting a request and waiting for a reply of at least one replica. Note that all correct replicas receive and process all requests in exactly the same order. Unfortunately, in this solution read and writes are equally expensive (and, in fact, any consensus protocol uses a write quorum internally).

2.5 Locking Quorum-Based Atomic Register

The quorum-based approach can be modified to use locking. In this approach, a lock is associated with the register. The algorithm is similar to the non-blocking quorum-based atomic solution described before with the following differences. When a read or write request is received, it is only processed if the register is unlocked. If the register is locked the request is queued until the register is unlocked. Also, when a process updates its local value in response to a WriteRequest it grants the lock to the writer. Furthermore, the write operation is augmented to include the multicast of a LockRelease message as soon as a WriteReply is received from a write quorum. As the name implies, the LockRelease message releases a lock that has been granted by a previous WriteRequest, allowing for pending (write or read) request to be processed. Obviously, this approach requires some form failure detection to release the lock when the process that holds the lock fails. Furthermore, in face of concurrent write operations, different writers may acquire write locks in different orders in different replicas, causing deadlocks. In fact, this sort of conflicts may hinder the efficiency of the replicated system [11].

2.6 From Registers to Transactional Databases

Transactions typically perform an atomic sequence of multiple read and/or write operation in several data items. When moving from register replication to database replication, two main approaches are possible: to perform inter-replica coordination every time a given data item is accessed (what has been called linear interaction mode [20]) or to execute the entire transaction optimistically in a single replica and perform a single inter-replica coordination step at the end (also called constant interaction approach [20]). Given that there is usually a high cost in the communication, we will concentrate

on the later approach. In particular, we will describe an approach that is known as the certification approach with voting [15]. The general outline of the algorithm is as follows:

1. A transaction t is submitted to a given replica of the database. Let this replica be denoted the delegate node. All the transaction's operations are executed locally on the delegate node, obtaining (local) read locks on read data items (in this approach we assume that, in order to be written, an object must be previously read. i.e., we exclude blind writes).

2. When the application requests a commit, a PREPARE message with the set of written data items is sent to all nodes using atomic broadcast.

3. When the write set of a transaction t is delivered by atomic broadcast, all nodes try to obtain local write locks on all items in the set. If there is a transaction that holds a write lock on any item of the write set of t, t is placed on hold until that write lock is relinquished. Transactions holding read locks on any item of the write set of t are aborted: in this case the delegate node broadcasts an ABORT message to all replicas. When the delegate node has obtained all write locks, sends a COMMIT message to all servers, through (uniform) reliable broadcast.

4. Upon the reception of the COMMIT message, a node applies the transaction's write set to the local database and subsequently releases all locks held on behalf of that transaction. Upon the reception of an abort message, nodes abort the transaction and releases all its locks.

This algorithm requires the exchange of an atomic broadcast plus an (uniform) reliable broadcast message during the inter-replica coordination phase, also called the transaction certification phase. There is a variant of this algorithm that requires only the exchange of an atomic broadcast (avoiding the dissemination of the final commit/abort vote to all replicas) [20]. However, that variant requires the delegate node to send both the read and the write set to the other replicas. In any case, as in the consensus-based atomic register approach introduced above, all transactions (including read-only transactions) need to go through the global certification phase if linearizable semantics are to be provided. Thus, all transactions are required to use a write quorum to commit.

Note that, in this algorithm, both write and read transactions must use the atomic broadcast and the totally ordered certification procedure in

order to ensure linearizable semantics. In fact, if one attempts to optimize the system and execute read transaction in a single replica linearizability is not ensured. For instance, replica p receives a transaction t^w updating value of some data item fromv v to v' via the atomic broadcast primitive; p immediately executes a local read-only transaction t_1^r that reads the new value v'; another read transaction t_2^r, initiated after the commit of t_1^r, that is submitted to another replica may still read the old value v, given that the atomic broadcast message may be delivered at different times in different replicas. Actually, if read transactions from the same client are allowed to be submitted to different replicas (for instance, due to the use of a load balancer), this optimization does not even ensures serializability (see [16] for an interesting discussion on the problem and solutions based on atomic broadcast).

3 The WCRQ Algorithm

In the following, we present our novel algorithm, that we have named "Write-Consensus Read-Quorum" (WCRQ), that combines consensus and quorum based approaches to achieve the advantages of both schemes, namely:

i) It exhibits constant interaction [20], since a single coordination phase is executed at commit time. Thus, contrary to classical quorum based approaches, there is no need to communicate every time a data item is accessed.

ii) It allows read-only transactions to coordinate with only a read-quorum of replicas, while still providing linearizable semantics. Therefore, unlike classical consensus-based approaches, one can trade the efficiency of read and write transactions without weakening the semantics. In particular, one does not need to send all read-only transactions to a single primary copy.

3.1 Building Blocks

We consider an asynchronous message passing system model augmented with a failure detector [6]. Our algorithm uses the following broadcast primitives: regular reliable broadcast (*R-broadcast*), uniform reliable broadcast (*UR-broadcast*), and uniform total order (*UTO-broadcast*). Examples of implementations of these primitives can be found in [4, 6, 12]. In the following definitions, let Ω be the set of replicas that need to perform inter-replica coordination.

The perfect failure detector is denoted by $\mathcal{P}$, and it outputs, at every process, the set of processes that are detected to have crashed (we simply say *detected*). A perfect failure detector is characterized by the following

properties: (1) (accuracy) if a process p is detected by any process, then p has crashed and; (2) (completeness) eventually every process that crashes is permanently detected by every correct process. Once the crash of a process p is detected by some process q, the detection is permanent.

The primitive *R-broadcast(m)* satisfies the following three properties: (1) (agreement) if a correct process in Ω has *R-delivered*(m), then every correct process in Ω eventually *R-delivers*(m); (2) (validity) if a correct process *R-broadcasts(m)*, then every correct process in Ω eventually *R-delivers*(m); (3) (integrity) for any message m, every correct process delivers m only if m was previously broadcast by some process $p \in \Omega$.

The primitive *UR-broadcast(m)* satisfies the following three properties: (1) (uniform agreement) if a process in Ω has *UR-delivered*(m), then every correct process in Ω eventually *UR-delivers*(m); (2) (validity) as for the regular version; (3) (integrity) as for the regular version.

The properties of the primitive *UTO-broadcast* [13] are the same as the uniform reliable broadcast with the additional property: (4) (uniform total order) let m_1 and m_2 be two messages that are *UTO-broadcast*. We note $m_1 < m_2$ if and only if a process (correct or not) *UTO-delivers* m_1 before m_2.

We assume that a quorum system has been defined on the set of participants. The quorum system defines two types of quorums: *read quorums* and *write quorums*. The quorum system respects the following constraints: *i*) A read quorum always intersects with a write quorum. *ii*) A write quorum always intersects with (another) write quorum. Note that our algorithm is independent on the mechanisms used to define the quorum system: one may use weighted voting [10, 9] or some other form of quorums [7, 1]. The quorum system used by our replication algorithm does not need to be the same as the one used to implement uniform total order, but the overall efficiency of the system may benefit from the use of a consistent set of quorum system at both layers.

3.2 Algorithm Description

As the name "Write-Consensus Read-Quorum" implies, our algorithm uses different approaches to execute update transactions and read-only transactions.

Update transactions are executed in a manner that is similar to the certification algorithm with voting (CAV) of [15]. The transaction is executed optimistically in a single replica, called the *delegate* replica, and, at commit time, uniform total order is used to serialize a (globally consistent) certifi-

cation procedure. However, the certification procedure is different from the CAV: the main difference between CAV and the WCRQ algorithm is that, in WCRQ, an explicit acknowledgment is sent back to the delegate node and that a write quorum of acknowledgments needs to be collected before the transaction is certified.

In detail, the delegate replica in order to certify a transaction sends a PREPAREW(t, WS_t) message with the transaction write-set WS_t through the uniform total order primitive. When the PREPAREW(t, WS_t) message is delivered, each replica obtains a write lock for each item i in WS_t[1]; when all write locks have been acquired, it sends back an acknowledgment message ACKW(t) to the delegate node using a point-to-point channel. When the delegate node has acquired all the locks for t and it has received a ACKW(t) from a write-quorum of replicas, it sends a COMMIT(t) message using uniform reliable broadcast. When the COMMIT(t) message is delivered, t is committed, writing all its updates in the database and releasing all locks held by t. All transactions t'waiting to obtain write locks on an object written by t are aborted (the delegate node for that transaction sends a ABORT(t) message is sent using uniform broadcast). When a ABORT(t) message is delivered, t is aborted, releasing all its locks.

This protocol, in conjunction with the read algorithm to be described below, ensures that no transaction commits before obtaining write locks at a write-quorum of replicas on the data items that are about to be changed. As a result, subsequent read-only transactions are guaranteed to read the fresh data.

On the other hand, read-only transactions do not need to be broadcast using the uniform total order primitive. Instead, they are executed optimistically in a single replica (the delegate replica) and, at commit time, only a read-quorum of replicas need to be contacted to execute the certification procedure (note that the delegate node is part of this read-quorum). Replicas in the read-quorum check if the read items are still up-to-date in the local database. If the transaction is detected to read data "in the past" (because the data has been locked and/or updated by a more recent write-transaction), a notification is sent to the delegate node to abort the transaction.

In detail, the delegate node performs the following steps to certify a

[1]If there is one or more read locks on i, every t' not yet serialized by the total order primitive that has that read lock is aborted (by sending an ABORT(t') message using total order broadcast), and the write lock on o is granted to t. If there is a write lock on i, or if some read locks on i are from transactions t' that have been serialized before t, t will be placed on hold until those write/read locks are released.

read transaction. It sends a PREPARER(t, RS_t) message with the transaction read-set RS_t to a read-quorum of replicas. When the message PREPARER(t, RS_t) is delivered, each node certifies the transaction as follows. It obtains a read lock for all the data items in RS_t (if there is a write lock on the item, the transaction waits until the write lock is released). When the read locks have been aquired, if the data items version is the same as the one read at the delegate node, it sends back a ACKR(t) acknowledgement to the delegate node; otherwise, it sends back a NACKR(t) notification to the delegate node. In any case, the read locks are released as soon as the transaction has been certified. When the delegate node has received an ACKR(t) acknowledgement from a read-quorum of replicas, it commits the transaction t. Otherwise, if it receives a NACKR(t) notification, it aborts the transaction.

Given that the protocol requires the delegate node to send a second message to commit or abort the transaction, this protocol uses a perfect failure detector to signal the failure of the delegate node and initiate an inter-replica coordination phase to decide on the outcome of the affected transactions. This protocol step is only executed when nodes fail and does not affect the performance of the protocol in steady-state. The reader must note that most practical implementations of database replication systems that are based on the certification approach rely on some form of group communication [4] service (see, for instance, the GORDA architecture [5]); in this case, the failure of a replica is followed by the installation of a new view (with the up to date membership of active replicas); the view installation procedure introduces a global consistency point in the communication flow that makes the inter-replica coordination trivial [4]).

4 Discussion

In this section, we first discuss the correctness of our algorithm and later we discuss its cost in terms of communication steps and number of messages.

4.1 Correctness

We now discuss the correctness of the algorithm. Assume that a given data item as an initial value v. Assume also that an update transaction t^w updates the value of that data item to v'. An execution would violate if a transaction t_1^r would commit reading the value v' and a subsequent transaction t_2^r, initiated after the commit of t_1^r, would still read value v. We show that this is impossible by contradiction. Assume that the delegate

node for transaction t_1^r is node p. In order to read v' at node p, t^w must have committed at node p. Let q be the delegate node for transaction t^w. In order to q to send a COMMIT(t^w) message it must have collected ACKW(t^w) messages from a write quorum Q^w of replicas. Also, ACKW(t^w) messages are only sent after a write lock has been obtained for the data item. Thus, when transaction t_1^r commits, a write quorum Q^w of replicas has either: *i*) the item locked or *ii*) the new value v' for that item (if they have also received the commit message).

Assume that the delegate node for transaction t_2^r is node r. In order to commit t_1^r and return the value v, node r must contact a read quorum of replicas Q^r. Furthermore, in order to send back an ACKR(t_2^r), all nodes $n \in Q^r$ must have the old version of the data and no write lock on the item. However, given that transaction t_2^r is initiated after transaction t_1^r, there is at least one node $i \in Q^w \cap Q^r$ that either has the item locked or the value v', thus the contradiction.

4.2 Cost

We analyze the cost of our algorithm in terms of the number of messages and communication steps needed to provide the service. This measure is less ambiguous than the usual number of "phases". To give an example, the classical two phase commit protocol (2PC) has three communication steps [3]: (1) PREPARE sent from the coordinator to the participants, (2) REPLY of the participants sent to the coordinator, and (3) DECISION sent by the coordinator to the participants. We consider in our analysis only the best case scenario, i.e. runs with no failure suspicions. This is the most frequent case in practice. Using this metrics, we now analyze the cost of our building blocks and that of the certification algorithm with voting (CAV) of [15]. These costs are summarized in Table 1. In the table, N is the total number of replicas, W is the size of a write quorum, and R is the size of a read quorum.

The cost of a regular reliable broadcast is just the cost of sending the message to each recipient, i.e., one multicast step or N point-to-point messages. Uniform reliable multicast requires the exchange of acknowledgments among recipients, thus it requires one additional communication step and N multicast acknowledgments (or N^2 point-to-point acknowledgments). Total order typically requires 3 communication steps, as one additional step and one additional multicast message is required to assign a sequence number to the replicas (this step may be omitted in some cases, but this is not relevant to make the comparative performance analysis). The costs for the

Algorithm	*Steps*	*p2p messages*	*Multicast messages*
Regular reliable broadcast	1	N	1
Uniform reliable broadcast	2	$N + N^2$	$1 + N$
Uniform total order	3	$2N + N^2$	$2 + N$
CAV-Write	5	$3N + 2N^2$	$3 + 2N$
WCRQ-Write	6	$3N + 2N^2 + W$	$3 + 2N + W$
WCRQ-Write (cross-layer)	5	$3N + 2N^2$	$3 + 2N$
CAV-ReadOnly	5	$3N + 2N^2$	$3 + 2N$
WCRQ-ReadOnly	2	$2R$	$1 + R$

Table 1: Communication Steps

consistency protocols are just the sum of the cost of their building blocks.

As it can be observed, using the WCRQ algorithm provides much more efficient linearizable read-only transactions than the voting certification algorithm. Furthermore, most of the additional cost of W messages for update transactions can be eliminated using cross-layer design techniques. In fact, the implementation of the uniform total order primitive requires the exchange of acknowledgments among the participants. If we piggy-back the AckW message with the acknowledgment generated at the level of the uniform total order primitive, the cost of the update transaction can approximate the cost of the voting certification algorithm. In fact, generating an acknowledgment at the replication layer is slightly less efficient than generating the acknowledgment at the (lower) uniform total order layer, but this cost increment is negligible when compared with the full cost of sending an additional message.

5 Conclusions

In this paper we have proposed a new replication algorithm, that we have named "Write-Consensus Read-Quorum". The algorithm combines the advantages of certification-based replication protocols, namely constant interaction cost, with the advantages of quorum-based replication (faster reads) when providing strong consistency, in particular, linearizable semantics. Interestingly, the algorithm achieves these qualities with negligible overhead for write transactions in terms of number of messages or communication steps when compared with competing algorithms such as [15], that require the use of a write-quorum to certify a read-only transaction in order to offer linearizable semantics. Furthermore, contrary to primary-backup style

solutions, that redirect all read operations to a single replica, we allow the parallel execution of read-only transactions in multiple replicas.

References

[1] D. Agrawal and A. El-Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transaction on Computer Systems*, February 1991.

[2] P. Bernstein and N. Goodman. Serializability theory for replicated databases. *Journal of Computer and System Sciences*, 31:355–374, 1985.

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

[4] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Comm. ACM*, 36(12):37–53, December 1993.

[5] N. Carvalho, A. Correia Jr., J. Pereira, L. Rodrigues, R. Oliveira, and S. Guedes. On the use of a reflective architecture to augment database management systems. *Journal of Universal Computer Science*, 13(8):1110–1135, 2007.

[6] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 34(1):225–267, 1996. A preliminary version appeared in the *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[7] S. Cheung, M. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. In *Proceedings of the 6th Internation Conference on Data Engineering*, pages 438–445, 1990.

[8] P. Dutta, R. Guerraoui, R. Levy, and A. Chakraborty. How fast can a distributed read be? In *Proc. of the ACM Symposium on Principles of Distributed Computing (PODC 2004)*, pages 236–245, 2004.

[9] H. Garcia-Molina and D. Barbara. How to assign votes in distributed system. *Journal of the ACM*, 32(4):841–860, 1985.

[10] D. Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating System Principles*, pages 150–162, USA, December 1979.

[11] J. Gray, P. Helland, P. O'Neal, and D. Shasha. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Quebec, Canada, June 1996.

[12] Rachid Guerraoui and Luís Rodrigues. *Reliable Distributed Systems*. Springer Verlag, 2006.

[13] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems*, pages 97–145. ACM Press, 1993.

[14] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[15] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the 18th International Conference on Distributed Computing Systems (ICDCS)*, The Netherlands, May 1998.

[16] R. Oliveira, J. Pereira, A. Correia Jr, and E. Archibald. Revisiting 1-copy equivalence in clustered databases. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 728 – 732, 2006.

[17] M. Patiño Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of the 14th International Symposium on Distributed Computing (DISC)*, Toledo, Spain, October 2000.

[18] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[19] J. Turek and D. Shasha. The Many Faces of Consensus in Distributed Systems. *IEEE Computer*, 25(6):8–17, June 1992.

[20] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In

Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000), Germany, October 2000.