Fault-Tolerant Partial Replication in Large-Scale Database Systems

Pierre Sutra, Marc Shapiro

▶ To cite this version:

HAL Id: inria-00232662

https://hal.inria.fr/inria-00232662v4

Submitted on 31 Mar 2009

INRIA

Fault-Tolerant Partial Replication in Large-Scale Database Systems

Pierre Sutra   Marc Shapiro

Université Paris VI and INRIA Rocquencourt, France

N° 6440

Janvier 2008

Thème COM

Rapport de recherche

Fault-Tolerant Partial Replication in Large-Scale Database Systems

Pierre Sutra*   Marc Shapiro

Université Paris VI and INRIA Rocquencourt, France

Abstract: We investigate a decentralised approach to committing transactions in a replicated database, under partial replication. Previous protocols either reexecute transactions entirely and/or compute a total order of transactions. In contrast, ours applies update values, and generate a partial order between mutually conflicting transactions only. It results that transactions execute faster, and distributed databases commit in small committees. Both effects contribute to preserve scalability as the number of databases and transactions increase. Our algorithm ensures serializability, and is live and safe in spite of faults.

Key-words: data replication, large-scale, database systems

* LIP6, 104, ave. du Président Kennedy, 75016 Paris, France; mailto:pierre.sutra@lip6.fr

Un algoorithme tolérant aux fautes pour la réplication de bases de données dans les systèmes large-échelle

Résumé :

1 Introduction

Non-trivial consistency problems e.g. file systems, collaborative environments, and databases. are the major challenge of large-scale systems. Recently some architectures have emerged to scale file systems up to thousands of nodes [12, 15, 3], but no practical solution exists for database systems.

At the cluster level protocols based on group communication primitives [4, 11, 16] are the most promising solutions to replicate database systems [22] . In this article we extend the group communication approach to large-scale systems.

Highlights of our protocol:

- Replicas do not re-execute transactions, but apply update values only.

- We do not compute a total order over of operations. Instead transactions are partially ordered. Two transactions are ordered only over the data where they conflict.

- For every transaction T we maintain the graph of T's dependencies. T commits locally when T is transitively closed in this graph.

The outline of the paper is the following. Section 2 introduces our model and assumptions. Section 3 presents our algorithm. We conclude in Section 4 after a survey of related work. An appendix follows containing a proof of correctness.

2 System model and assumptions

We consider a finite set of asynchronous processes or *sites* Π, forming a distributed system. Sites may fail by crashing, and links between sites are asynchronous but reliable. Each site holds a database that we model as some finite set of *data items*. We left unspecified the granularity of a data item. In the relational model, it can be a column, a table, or even a whole relational database. Given a data item x, the replicas of x, noted *replicas(x)*, are the subset of Π whose databases contain x.

We base our algorithm on the three following primitives:[1]

- *Uniform Reliable Multicast* takes as input a unique message m and a *single* group of sites $g \subseteq \Pi$. Uniform reliable multicast consists of the two primitives R-multicast(m) and R-deliver(m). With Uniform Reliable Multicast, all sites in g have the following guarantees:

  - Uniform Integrity: For every message m, every site in g performs R-deliver(m) at most once, and only if some site performed R-multicast(m) previously.
  - Validity: if a correct site in g performs R-multicast(m) then it eventually performs R-deliver(m).

---

[1]Our taxonomy comes from [5].

lock held

		R	W	IW
lock	R	1	0	0
requested	W	0	0	0
	IW	0	0	1

Table 1: Lock conflict table

- Uniform Agreement: if a site in g performs R-deliver(m), then every correct sites in g eventually performs R-deliver(m).

Uniform Reliable Multicast is solvable in an asynchronous systems with reliable links and crash-prone sites.

. *Uniform Total Order Multicast* takes as input a unique message m and a single group of sites g. Uniform Total Order Multicast consists of the two primitives TO-multicast(m) and TO-deliver(m). This communication primitive ensures Uniform Integrity, Validity, Uniform Agreement and Uniform Total Order in g:

- Uniform Total Order: if a site in g performs TO-deliver(m) and TO-deliver(m') in this order, then every site in g that performs TO-deliver(m') has performed previously TO-deliver(m).

. *Eventual Weak Leader Service* Given a group of sites g, a site $i \in g$ may call function *WLeader*(g). *WLeader*(g) returns a *weak leader* of g :

- *WLeader*(g) $\in g$.
- Let ρ be a run of Π such that a non-empty subset c of g is correct in ρ. It exists a site $i \in c$ and a time t such that for any calls of *WLeader*(g) on i after t, *WLeader*(g) returns i.

This service is strictly weaker than the classical eventual leader service Ω [18], since we do not require that every correct site eventually outputs the same leader. An algorithm that returns to every process itself, trivially implements the Eventual Weak Leader Service.

In the following we make two assumptions: during any run, **(A1)** for any data item x, at least one replica of x is correct, and **(A2)** Uniform Total Order Multicast is solvable in *replicas*(x).

2.1   Operations and locks

Clients of the system (not modeled), access data items using read and write operations. Each operation is uniquely identified, and accesses a single data item. A read operation is

a singleton: the data item read, a write operation is a couple: the data item written, and the update value.

When an operation accesses a data item on a site, it takes a lock. We consider the three following types of locks: read lock (R), write lock (W), and intention to write lock (IW).Table 1 illustrates how locks conflict with each other; when an operation requests a lock to access a data item, if the lock is already taken and cannot be shared, the request is enqueued in a FIFO queue. In Table 1, 0 means that the request is enqueued, and 1 that the lock is granted.

Given an operation o, we note:

- $item(o)$, the data item operation o accesses,

- $isRead(o)$ (resp. $isWrite(o)$) a boolean indicating whether o is a read (resp. a write),

- and $replicas(o) \stackrel{\triangle}{=} replicas(item(o))$;

We say that two operations o and o' *conflict* if they access the same data item and one of them is a write:

$$conflict(o, o') \stackrel{\triangle}{=} \left\{ \begin{array}{l} item(o) = item(o') \\ isWrite(o) \vee isWrite(o') \end{array} \right.$$

2.2 Transactions

Clients group their operations into *transactions*. A transaction is a uniquely identified set of read and write operations. Given a transaction T,

- for any operation $o \in T$, function $trans(o)$ returns T,

- $ro(T)$ (respectively $wo(T)$) is the subset of read (resp. write) operations,

- $item(T)$ is the set of data items transaction T accesses: $item(T) \stackrel{\triangle}{=} \bigcup_{o \in T} item(o)$.

- and $replicas(T) \stackrel{\triangle}{=} replicas(item(T))$.

Once a site i grants a lock to a transaction T, T holds it until i commits T, i aborts T, or we explicitly say that this lock is released.

3 The algorithm

As replicas execute transactions, it creates precedence constraints between conflicting transactions. Serializability theory tell us that this relation must be acyclic [2].

One solution to this problem, is given a transaction T, (i) to execute T on every replicas of T, (ii) to compute the transitive closure of the precedence constraints linking T to concurrent conflicting transactions, and (iii) if a cycle appears, to abort at least one the transactions involved in this cycle.

Unfortunately as the number of replicas grows, sites may crash, and the network may experience congestion. Consequently to compute (ii) the replicas of T need to agree upon the set of concurrent transactions accessing $item(T)$.

Our solution is to use a TO-multicast protocol per data item.

3.1   Overview

To ease our presentation we consider in the following that a transaction executes initially on a single site. Section 3.9 generalizes our approach to the case where a transaction initially executes on more than one site. We structure our algorithm in five phases:

- In the *initial execution phase*, a transaction T executes at some site i.

- In the *submission phase*, i transmits T to $replicas(T)$.

- In the *certification phase*, a site j aborts T if T has read an outdated value. If T is not aborted, j computes all the precedence constraints linking T to transactions previously received at site j.

- In the *closure phase*, j completes its knowledge about precedence constraints linking T to others transactions.

- Once T is closed at site j, the *commitment phase* takes place. j decides locally whether to commit or abort T. This decision is deterministic, and identical on every site replicating a data item written by T.

3.2   Initial execution phase

A site i executes a transaction T coming from a client according to the two-phases locking rule [2], *but* without applying write operations[2]. When site T reaches a commit statement, it is not committed, instead i releases T's read locks, converts T's write locks into intention to write locks, computes T's update values, and then proceeds to the submission phase.

3.3   Submission phase

In this phase i R-multicasts T to $replicas(T)$. When a site j receives T, j marks all T's operations as pending using variable *pending*. Then if it exists an operation $o \in pending$, such that $j = WLeader(replicas(o))$, j TO-multicasts o to $replicas(o)$.[3]

---

[2]If T writes a data item x then reads it, we suppose some internals to ensure that T sees a consistent value.

[3]If instead of this procedure, i TO-multicasts all the operations, then the system blocks if i crashes. We use a weak leader and a reliable multicast to preserve liveness.

3.4 Certification phase

When a site i TO-delivers an operation o *for the first time*[4],i removes o from *pending*, i certifies o.

To certify o, i considers any preceding write operations that conflicts with o. We say that a conflicting operation o' *precedes o at site i*, $o' \rightarrow_i o$, if i TO-delivers o' then i TO-delivers o:

$$o' \rightarrow_i o \triangleq \left\{ \begin{array}{l} \text{TO-deliver}_i(o') \prec \text{TO-deliver}_i(o) \\ conflict(o', o) \end{array} \right.$$

Where given two events e and e', $e \prec e'$ is the relation e *happens-before* e', and TO-deliver$_i(o')$ is the event: "site i TO-delivers operation o'".

If o is a read, we check that o did not read an outdated value. It happens when o executes concurrently to a conflicting write operation o' that is now committed. Let *committed$_i$* be the set of transactions committed at site i, the read operation o aborts, if it exists an operation o' such that $o' \rightarrow_i o \wedge trans(o') \parallel trans(o) \wedge trans(o') \in committed_i$, where $trans(o') \parallel trans(o)$ means that the transactions $trans(o')$ and $trans(o)$ were executed concurrently during the initial execution phase.

If now o is a write, i gives an IW lock to o: function *forceWriteLock(o)*. If an operation o' holds a conflicting IW lock, o and o' share the lock (see Table 1); otherwise it means that $trans(o')$ is still executing at site i, and function *forceWriteLock(o)* aborts it.[5]

3.5 Precedence graph

Our algorithm decides to commit or abort transactions, according to a *precedence graph*. A precedence graph G is a directed graph where each node is a transaction T, and each directed edge $T \rightarrow T'$, models a precedence constraint between an operation of T, and a *write* operation of T':

$$T \rightarrow T' \triangleq \exists (o, o') \in T \times T', \exists i \in \Pi, o' \rightarrow_i o$$

A precedence graph contains also for each vertex T a flag indicating whether T is aborted or not: $isAborted(T, G)$, and the subset of T's operations: $op(T, G)$, which contribute to the relations linking T to others transactions in G.

Given a precedence graph G, we note $G.\mathcal{V}$ its vertices set, and $G.\mathcal{E}$ its edges set. Let G and G' be two precedence graphs, the union between G and G', $G \cup G'$, is such that:

- $(G \cup G').\mathcal{V} = G.\mathcal{V} \cup G'.\mathcal{V}$,

- $(G \cup G').\mathcal{E} = G.\mathcal{E} \cup G'.\mathcal{E}$,

- $\forall T \in (G \cup G').\mathcal{V}, isAborted(T, (G \cup G')) = isAborted(T, G) \vee isAborted(T, G')$.

- $\forall T \in (G \cup G').\mathcal{V}, op(T, (G \cup G')) = op(T, G) \cup op(T, G')$.

---

[4]Recall that the leader is eventual, consequently i may receive o more than one time.
[5]This operation prevents local deadlocks.

Algorithm 1 $decide(T, G)$, code for site i

1: **variable** $G' := (\emptyset, \emptyset)$ ▷ a directed graph
2:
3: **for all** $C \subseteq cycles(G)$ **do**
4:     **if** $\forall T \in C, \neg isAborted(T, G)$ **then**
5:         $G' := G' \cup C$
6: **if** $T \in breakCycles(G')$ **then**
7:     **return** *false*
8: **else**
9:     **return** *true*

We say that G is a subset of G', noted $G \subseteq G'$, if:

. $G.\mathcal{V} \subseteq G'.\mathcal{V} \wedge G.\mathcal{E} \subseteq G'.\mathcal{E}$,

. $\forall T \in G.\mathcal{V},\ isAborted(T, G) \Rightarrow isAborted(T, G')$,

. $\forall T \in G.\mathcal{V},\ op(T, G) \subseteq op(T, G')$.

Let G be a precedence graph, $in(T, G)$ (respectively $out(T, G)$) is the restriction of $G.\mathcal{V}$ to the subset of vertices formed by T and its incoming (resp. outgoing) neighbors. The *predecessors* of T in G: $pred(T, G)$, is the precedence graph representing the transitive closure of the dual of the relation $G.\mathcal{E}$ on $\{T\}$.

3.6   Deciding

Each site i stores its own precedence graph G_i, and decides locally to commit or abort a transaction according to it. More precisely i decides according to the graph $pred(T, G_i)$. For any cycle C in the set of cycles in $pred(T, G_i)$: $cycles(pred(T, G_i))$. i must abort at least one transaction in C. This decision is deterministic, and i tries to minimize the number of transactions aborted.

Formally speaking i solves the minimum feedback vertex set problem over the union of all cycles in $pred(T, G_i)$ containing only non-aborted transactions The minimum feedback vertex set problem is an NP-complete optimization problem, and the literature about this problem is vast [6]. We consequently postulate the existence of an heuristic: $breakCycles()$. $breakCycles()$ takes as input a directed graph G, and returns a vertex set S such that $G \setminus S$ is acyclic.

Now considering a transaction $T \in G_i$ such that $G = pred(T, G_i)$, Algorithm 1 returns *false* if i aborts T, or *true* otherwise.

3.7   Closure phase

In our model sites replicate data partially, and consequently maintain an incomplete view of the precedence constraints linking transactions in the system. Consequently they need to complete their view by exchanging parts of their graphs. This is our closure phase:

- When i TO-delivers an operation $o \in T$, i adds T to its precedence graph, and adds o to $op(T, G_i)$. Then i sends $pred(T, G_i)$ to $replicas(out(T, G_i))$ (line 29).

- When i receives a precedence graph G, if $G \nsubseteq G_i$, for every transaction T in G_i, such that $pred(T, G) \nsubseteq pred(T, G_i)$, i sends $pred(T, G \cup G_i)$ to $replicas(out(T, G_i))$. Then i merges G to G_i (lines 31 to 35).

Once i knows all the precedence constraints linking T to others transactions, we say that T is *closed* at site i. Formally T is closed at site i when the following fixed-point equation is true at site i:

$$closed(T, G_i) = \begin{cases} op(T, G) = T \\ \forall T' \in in(T, G_i).\mathcal{V}, closed(T', G_i) \end{cases}$$

Our closure phase ensures that during every run ρ, for every correct site i, and every transaction T which is eventually in G_i, T is eventually closed at site i.

3.8 Commitment phase

If T is a read-only transaction: $wo(T) = \varnothing$, i commits T as soon as T is executed (line 9).

If T is an update, i waits that T is closed and holds all its IW locks: function $holdIWLocks()$ (line 35). Once these two conditions hold, i computes $decide(T, pred(T, G_i))$. If this call returns *true*, i commits T: for each write operation $o \in wo(T)$, with $i \in replicas(o)$, i considers any write operation o' such that $T \rightarrow trans(o') \in G_i \wedge conflict(o, o')$. If $trans(o')$ is already committed at site i, i does nothing; otherwise i applies o to its database.

Algorithm 2 describes our algorithm. This protocol provides serializability for partially replicated database systems: any run of this protocol is equivalent to a run on a single site [2]. The proof of correctness appears in Appendix.

3.9 Initial execution on more than one site

When initial execution phase does not take place on a single site we compute the read-from dependencies. More precisely when a site i receives a read o accessing a data item it does not replicate, i sends o to some replica $j \in replicas(o)$. Upon reception j executes o. At the end of execution j sends back to i the transitive closure containing read-from dependencies and starting from T.

Once i has executed locally or remotely all the read operations in T, i checks if the resulting graph contains cycles in which T is involved. If this is the case, T will be aborted, and instead of submitting it, i re-executes at least one of T's read operations Otherwise i computes the write set and the update values, and sends T with its read-from dependencies by Uniform Reliable Multicast. The dependencies are merged to precedence graph when a site receives an operation by Total Order Multicast. The rest of the algorithm remains the same.

Algorithm 2 code for site i

1: **variables** $G_i := (\emptyset, \emptyset)$; $pending := \emptyset$

2:

3: **loop**   ▷ Initial execution
4:     let T be a new transaction
5:     $initialExecution(T)$
6:     **if** $wo(T) \neq \emptyset$ **then**
7:         R-multicast(T) to $replicas(T)$
8:     **else**
9:         $commit(T)$

10:

11: **when** R-deliver(T)   ▷ Submission
12:     **for all** $o \in T : i \in replicas(o)$ **do**
13:         $pending := pending \cup \{o\}$

14:

15: **when** $\exists o \in pending \wedge i = WLeader(replicas(o))$
16:     TO-multicast(o) to $replicas(o)$

17:

18: **when** TO-deliver(o) **for the first time**   ▷ Certification
19:     $pending := pending \setminus \{o\}$
20:     let $T = trans(o)$
21:     $G_i.\mathcal{V} := G_i.\mathcal{V} \cup \{T\}$
22:     $op(T, G_i) := op(T, G_i) \cup \{o\}$
23:     **if** $isRead(o) \wedge (\exists o', o' \rightarrow_i o \wedge trans(o') \parallel trans(o) \wedge trans(o') \in committed_i)$ **then**
24:         $setAborted(T, G_i)$
25:     **else if** $isWrite(o)$ **then**
26:         $forceWriteLock(o)$
27:         **for all** $o' : o' \rightarrow_i o$ **do**
28:             $G_i.\mathcal{E} := G_i.\mathcal{E} \cup \{(trans(o'), T)\}$
29:     send($pred(T, G_i)$) to $replicas(out(T, G_i))$

30:

31: **when** receive(T, G)   ▷ Closure
32:     **for all** $T \in G_i$ **do**
33:         **if** $pred(T, G) \not\subseteq pred(T, G_i)$ **then**
34:             send($pred(T, G_i \cup G)$) to $replicas(out(T, G_i))$
35:     $G_i := G_i \cup G$

36:

37: **when** $\exists T \in G_i, \begin{cases} i \in replicas(wo(T)) \\ closed(T, G_i) \\ holdIWLocks(T) \end{cases}$ ▷ Commitment
38:     **if** $\neg isAborted(T, G_i) \wedge decide(T, pred(T, G_i))$ **then**
39:         $commit(T)$
40:     **else**
41:         $abort(T)$

42:

3.10   Performance analysis

We consider Paxos [14] as a solution to Uniform Total Order Multicast. Since precedence constraints in a cycle are *not* causally related, Algorithm 2 achieves a message delay of 5: 2 for Uniform Reliable Multicast, and 3 for Uniform Total Order Multicast. It reduces to 4, if in each replica group the leader of Paxos is also the weak leader of g.

Let o be the number of operations per transaction, and d be the replication degree, the message complexity of Algorithm 2 is $5od + (od)^2$: $2od$ for Uniform Reliable Multicast, o Uniform Total Order Multicasts, each costing $2d$ messages, and od replicas execute line 29, each site sending od messages. Again, if in each replica group, the leader of Paxos is also the weak leader of g, the message complexity of our protocol reduces to $4od + (od)^2$

4   Concluding remarks

4.1   Related work

Gray et al. [7] prove that scale traditional eager and lazy replications does not scale: the deadlock rate increase as the cube of the number of sites, and the reconciliation rate increases as the square. Wiesmann and Schiper confirm practically this result [22]. Fritzke et al. [10] propose a replication scheme where sites TO-multicast each operations and execute them upon reception. However they do not prevent global deadlocks with a priority rule; it increases abort rate. Preventive replication [16] considers that a bound on processor speed, and network delay is known. Such assumptions do not hold in a large-scale system. The epidemic algorithm of Holiday et al [9] aborts concurrent conflicting transactions and their protocol is not live in spite of one fault. In all of these replication schemes, each replica execute all the operations accessing the data items it replicates. Alonso proves analytically that it reduces the scale-up of the system [1].

The DataBase State Machine approach [17] applies update values only but in a fully replicated environment. Its extensions [19, 21] to partial replication require a total order over transactions.

Committing transactions using a distributed serialization graph is a well-known technique [20]. Recently Haller et al. have proposed to apply it [8] to large-scale systems, but their solution does not handle replication, nor faults.

4.2   Conclusion

We present an algorithm for replicating database systems in a large-scale system. Our solution is live and safe in presence of non-byzantine faults. Our key idea is to order conflicting transaction per data item, then to break cycles between transactions. Compared to previous existing solutions, ours either achieves lower latency and message cost, or does not unnecessarily abort concurrent conflicting transactions.

The closure of constraints graphs is a classical idea in distributed systems. We may find it in the very first algorithm about State Machine Replication [13], or in a well-known

algorithm to solve Total Order Multicast [5].[6]We believe that the closure generalizes to a wider context, where a constraint is a temporal logic formula over sequences of concurrent operations.

References

[1] G. Alonso. Partial database replication and group communication primitives in 2nd European Research Seminar on Advances in Distributed Systems, 1997.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. .

[3] J-M. Busca, F. Picconi, and P. Sens. Pastis: A highly-scalable multi-user peer-to-peer file system. In *Euro-Par*, 2005.

[4] L. Camargos, F. Pedone, and M. Wieloch. Sprint: a middleware for high-performance transaction processing. *SIGOPS Oper. Syst. Rev.*, 2007.

[5] X. Defago, A. Schiper, and P. Urban. Totally ordered broadcast and multicast algorithms: a comprehensive survey, 2000.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[7] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, 1996.

[8] K. Haller, H. Schuldt, and C. Türker. Decentralized coordination of transactional processes in peer-to-peer environments. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, 2005.

[9] J. Holliday, D. Agrawal, and A. Abbadi. Partial database replication using epidemic communication, 2002.

[10] U. Fritzke Jr. and P. Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 284. IEEE Computer Society, 2001.

[11] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *The VLDB Journal*, pages 134–143, 2000.

[12] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[14] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[15] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[16] Esther Pacitti, Cédric Coulon, Patrick Valduriez, and M. Tamer Özsu. Preventive replication in a database cluster. *Distrib. Parallel Databases*, 18(3):223–251, 2005.

[17] F Pedone, R Guerraoui, and A Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, July 2003.

---

[6]In [13] Lamport closes the $\ll$ relation for every request to the critical section. In [5] the total order multicast protocol attributed to Skeen, closes the order over natural numbers to TO-multicast a message.

[18] Michel Raynal. Eventual leader service in unreliable asynchronous systems: Why? how? In *NCA*, pages 11–24. IEEE Computer Society, 2007.

[19] N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. In *10th International Conference on Principles of Distributed Systems (OPODIS'2006)*, 2006.

[20] C-S Shih and J. A. Stankovic. Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems. Technical report, 1990.

[21] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine, 2001.

[22] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4), 2005.

.1   Additionnal notations

We note $\mathbb{D}$ the universal set of data item, $\mathbb{T}$ the universal set of transactions, and $\mathbb{G}$ the universal set of precedence graphs constructed upon $\mathbb{D}$.

Let ρ be a run of Algorithm 2, given a site i we note $event_i$ when the event $event$ happens at site i during ρ ; moreover if $value$ is the result of this event we note it: $event_i = value$.

Let ρ be a run of Algorithm 2, we note:

- $faulty(\rho)$ the set of sites that crashes during ρ,

- $correct(\rho)$ the set $\Pi \setminus faulty(\rho)$.

- $committed(\rho)$ the transactions committed during ρ, i.e. $\{T \in \mathbb{T}, \exists i \in \Pi, T \in committed_i\}$,

- and $aborted(\rho)$ the transactions aborted during ρ, i.e. $\{T \in \mathbb{T}, \exists i \in \Pi, T \in aborted_i\}$.

Given a site i and a time t, we note $G_{i,t}$ the value of G_i at time t.

.2   Proof of correctness

Since the serializability theory is over a finite set of transactions, we suppose hereafter that during ρ a finite subset of $\mathbb{T}$ is sent to the system.

Let ρ be a run of Algorithm 2, we now proove a series of propositions leading to the fact that ρ is serializable.

$\boxed{\textbf{P1}}$

$$\forall T \in \mathbb{T}, (\exists j \in \Pi, \text{R-deliver}_j(T) \in \rho)$$
$$\Rightarrow (\forall o \in T, \forall i \in replicas(o) \cap correct(\rho), \text{TO-deliver}_i(o) \in \rho)$$

Proof

Let T be a transaction and j a site that R-delivers T during ρ.

F1.1 $\forall i \in replicas(T) \cap correct(\rho), \text{R-deliver}_i(T)$

By the Uniform Agreement property of Uniform Reliable Multicast.

F1.2 $\forall o \in T, \exists k \in correct(\rho) \cap replicas(o), \text{TO-multicast}_k(o) \in \rho$

F1.2.1 $\exists l \in correct(\rho) \cap replicas(o), WLeader_l(replicas(o)) = l \wedge \text{R-deliver}_l(o)$

By fact **F.1.1**, assumption **A1** and the properties of the Eventual Weak Leader Service.

By fact **F1.2.1** eventually a correct site executes line 16 in Algorithm 2.

Fact **F1.2** and the Validity and the Agreement properties of Total Order Multicast conclude our claim.

$\square$

In the following we say that a transaction T is submitted to the system: $T \in submitted(\rho)$, if a site i R-delivers T during ρ.

P2

$\forall T \in submitted(\rho), \forall i \in replicas(T), \forall o \in T,$

$$\exists G \in \mathbb{G}, o \in op(T, G) \wedge \text{receive}_i(G)$$

Proof

    F2.1 $\forall i \in \Pi, \forall t, t', t > t' \Rightarrow G_{i,t} \subseteq G_{i,t'}$

    F2.2 $\forall G \in \mathbb{G}, \forall T \in \mathbb{T}, T \in G \Rightarrow T \in pred(T, G)$

        By definition of $pred(T, G)$.

By proposition **P1**, facts **F2.1** and **F2.2**, and since links are reliable.

$\square$

P3

$\forall T, T' \in submitted(\rho),$

$$T \rightarrow T' \Rightarrow (\exists o, o' \in T \times T', \exists i \in correct(\rho), o \rightarrow_i o')$$

Proof

By definition of $T \rightarrow T'$, let $o, o' \in T \times T'$ and let j be a site such that $o \rightarrow_j o'$. Since $conflict(o, o')$ and an operation applies on a single data item, we note x the unique data item such that $x = item(o) = item(o')$.

    F3.1 $j \in replicas(x)$

        Site j TO-delivers o during ρ and links are reliable.

    F3.2 $\exists i \in replicas(x) \cap correct(\rho), \text{TO-deliver}_i(o) \wedge \text{TO-deliver}_i(o')$

By assumption **A1** $\exists i \in replicas(x) \cap correct(\rho)$, and by the Uniform Agreement property of Total Order Multicast, since i is correct during ρ, i TO-delivers both o and o'.

Fact **F3.2** and the Total Order property of Total Order Multicast concludes our claim.

$\square$

$\boxed{\textbf{P4}}$

$\forall T \in submitted(\rho), \forall i \in \Pi$
$(\exists t, T \in G_{i,t}) \Rightarrow (\exists T_1, \ldots, T_{m \geq 0} \in submitted(\rho), i \in replicas(T_m) \wedge T \rightarrow T_1 \rightarrow \ldots \rightarrow T_m)$

Proof
Since $G_{i,0} = (\emptyset, \emptyset)$, let us consider the first time t_0 at which $T \in G_{i,t}$.
According to Algorithm 2 either:

- i TO-delivers an operation $o \in T$ at t_0, and thus $i \in replicas(T)$. QED

- or i receives a precedence graph G' from a site j such that $T \in G'$. Now since links are reliable, note t_1 the time at which j send G' to i. According to lines 29 and 34, it exists a transactions T' such that $T \in pred(T', G_{j,t_1})$, and a transaction T'' such that $T'' \in out(T', G_{j,t_1})$ and $i \in replicas(T'')$.

  From $T \in pred(T',)$, by definition of the predecessors, we obtain $T \rightarrow \ldots \rightarrow T'$, and from $T'' \in out(T', G_{j,t_1})$ we obtain $T' \rightarrow T''$. Thus $T \rightarrow \ldots \rightarrow T' \rightarrow T''$, with $i \in replicas(T'')$.

$\square$

$\boxed{\textbf{P5}}$

$\forall T \in submitted(\rho), \forall i \in correct(\rho),$

$$(\exists t, T \in G_{i,t}) \Rightarrow (\exists t, op(T, G_{i,t}) = T)$$

Proof
Let T_0 be a transaction submitted during ρ and let i be a site that eventually hold T_0 in G_i.

By proposition **P4** it exists $T_1, \ldots, T_{m \geq 0} \in submitted(\rho)$ such that $i \in replicas T_m$ and $T \rightarrow T_1 \rightarrow \ldots \rightarrow T_m$.

Let $k \in [\![0, m]\!]$, we note $\mathcal{P}(k)$ the following property:

$$\mathcal{P}(k) \triangleq \forall j \in correct(\rho) \cap replicas(T_k), \exists t \in op(T_0, G_{j,t_0}) = T_0$$

Observe that by proposition **P2** $\mathcal{P}(0)$ is true. We now proove that $\mathcal{P}(k)$ is true for all the k by induction:

Let $o, o' \in T_k \times T_{k+1}$, and $j \in correct(\rho)$ such that $o \rightarrow_j o'$.

Let t_0 be the first time at which j TO-delivers o during ρ.

Let t_2 be the first time at which $op(T_k, G_{j,t}) = T_k$ (since $G_{j,0} = (\varnothing, \varnothing)$, and $\mathcal{P}(k)$ is true).

Let t_1 be the first time at which j To-delivers o' during ρ.

Observe that since $o \rightarrow_j o'$, $t_O < t_1$. It follow that we have three cases to consider:

- cases $t_2 < t_0 < t_1$ and $t_0 < t_2 < t_1$

In these cases when j To-delivers o', we have:

$$T_k \rightarrow T_{k+1} \in G_{j,t_1} \wedge op(T_k, G_{j,t_1} = T_k)$$

Thus,

$$T_k \in pred(T_{k+1}, G_{j,t_1}) \wedge op(T_k, pred(T_k, G_{j,t_1})) = T_k$$

and according to Algorithm 2, j sends $pred(T_{k+1}, G_{j,t_1})$ to $replicas(out(T_{k+1},))G_{j,t_1}$.

Now since $replicas(T_{k+1}) \subseteq replicas(out(T_{k+1},))G_{j,t_1}$, given a site $j \in replicas(T_{k+1})$, eventually j receives $pred(T_{k+1}, G_{j,t_1})$, and merges it into its own precedence graph.

- case $t_0 < t_1 < t_2$

We consider two-subcases:

  - At t_2 j delivers an operation of T_k, and this operation is different from o'. Now since $T_k \rightarrow T_{k+1} \in G_{j,t_2}$, $\mathcal{P}(k+1)$ is true.
  - If now j receives a graph G such that $op(T_k, G) = T_k$, by definition of t_2, $G \subseteq G_{j,t_2}$, and more precisely, $pred(T_k, G) \not\subseteq pred(T_k, G_{j,t_2})$.
    It follows that j sends $pred(T_k, G \cup G_{j,t_2})$ to $replicas(out(T_k, G \cup \cup G_{j,t_2}))$. Finally since by definition of t_1, $T_k \rightarrow T_{k+1} \in G_{j,t_2}$, we obtain $T_{k+1} \in out(T_k, G \cup G_{j,t_2})$, from which we conclude that $\mathcal{P}(k+1)$ is true.

To conclude observe that since $i \in replicas(T_m)$ and $\mathcal{P}(m)$ is true, eventually $op(T_0, G_{i,t_0}) = T_0$.

$\square$

$\boxed{\textbf{P6}}$

$\forall T \in submitted(\rho), \forall i \in correct(\rho),$
$$(\exists t, T \in G_{i,t}) \Rightarrow (\exists t, \forall T' \in submitted(\rho), T' {\rightarrow} T \Rightarrow (T', T) \in G_{i,t})$$

Proof

F6.1 $\forall T, T' \in submitted\rho, \forall o, o' \in T \times T', (\exists i \in \Pi, o' {\rightarrow}_{\Rightarrow} o \forall j \in replicas(o), o {\rightarrow}_j o'$

By the Uniform Agreement and Total Order properties of Total Order MBroadcast

F6.2 $\forall T \in submitted(\rho), \forall o \in T, \forall i \in correct(\rho), (\exists t, oop(T, G_{i,t})) \Rightarrow (\forall T' submitted(\rho), T' {\rightarrow} T \Rightarrow \exists t, (T, T') \in G_{i,t})$

Since $o \in op(T, G_{i,t})$ and $G_{i,0} = (\emptyset, \emptyset)$, either:

1. $i \in replicas(T) \wedge \text{TO-deliver}_o(i)$
   First observe that since links are reliable $i \in replicas(o)$.
   Let T' be a a transaction, $o' \in T'$ an operation, and k a site such that $o' {\rightarrow}_k o$.
   By fact **F6.1** since $i, j \in replicas(o)$, $o' {\rightarrow}_i o$.

2. $\exists G \in \mathbb{G}, \text{receive}_T(G) \wedge o \in op(T, G)$
   According to Algorithm 2 it exists $k_0, \dots, k_m$ sites sucht that:
   - k_0 TO-delivers o during ρ, and execute line 29 sending $pred(T, G_{k_0})$ with $o \in op(T, predecessors T G_{k_0})$ and $k_1 \in replicas(out(T, G_{k_0}))$.
   - k_1 receives $pred(T, G_{k_0})$ during ρ and then execute line 29 or line 34, sending a precedence graph G such that $pred(T, G_{k_0}) \subseteq G$ to a set of replicas containinig k_2.
   - etc ... until i receives it.

   Consequently $pred(T, G_{k_0}) \subseteq G_{i,t}$, and according to our reasonning in item 1, we conclude that fact **F6.2** is true.

Fact **F6.2** and proposition **P5** conclude.

$\square$

We are now able to proove our central theorem: every transaction is eventually closed at a correct site.

$\boxed{\textbf{T1}}$

$\forall T \in submitted(\rho), \forall i \in correct(\rho),$

$$(\exists t, T \in G_{i,t}) \Rightarrow (\exists t, closed(T, G_{i,t}))$$

Proof

We consider that a finite subset of $\mathbb{T}$ are sent to the system, consequently $submitted(\rho)$ is also finite. Let C_T be the graph resulting from the transitive closure of the relation $\rightarrow$ on $\{T\}$. According to proposition **P6**, C_T is eventually in $G_{i,t}$, and thus according to proposition **P5**, T is eventually closed at site i.

$\square$

$\boxed{\textbf{P7}}$

$\forall T \in submitted(\rho), \forall i, j \in \Pi, \forall t, t',$
$(T \in G_{i,t} \wedge T \in G_{j,t'} \wedge closed(T, G_{i,t}) \wedge closed(T, G_{j,t'})) \Rightarrow (pred(T, G_{i,t}) = pred(T, G_{j,t'}))$

Proof

F7.1 $pred(T, G_i).\mathcal{V} = pred(T, G_j).\mathcal{V}$

Let $T' \in pred(T, G_i)$. By definition it exists $T_1, \ldots, T_m$ such that $T' \rightarrow T_1 \rightarrow \ldots \rightarrow T_m \rightarrow T \subseteq G_i$. By an obious induction on m using proposition **P6** we conclude that T' is also in $pred(T, G_j)$.

F7.2 $pred(T, G_i).\mathcal{E} = pred(T, G_j).\mathcal{E}$

Identical to the reasonning proposed for fact **F7.1**.

F7.2 $\forall T' \in pred(T, G_i), op(T', pred(T, G_i)) = op(T', pred(T, G_j))$

By fact **F7.1** and since T is closed at both sites i and j.

F7.4 $\{T' | isAborted(T, G_i)\} = \{T' | isAborted(T, pgraphSite j)\}$

Let $T' \in pred(T, G_i)$ such that $isAborted(T', pred(T, G_i))$. According to Algorithm 2, it exists a site k and a read operation $r \in T'$ such that k TO-delivers r during ρ, and then k set the aborted flag of T' in its precedence graph.

Now let k' be a replica of r, by the Uniform Agreement and the Total Order Property of Total Order Multicast, when k' TO-delivers r, it also set the aborted flag of T' in its precedence graph.

By the conjunction of facts **F7.1** to **F7.4**.

$\square$

We proove now that ρ is serializable [2].

Let $O(x, \rho)$ be the set of write operation over the data item x during ρ, we define the relation $\ll$ as follows:

$$\forall x \in \mathbb{D}, \forall o_1, o_2 \in O(x, \rho), x_1 \ll x_2 \overset{\triangle}{=} \exists i \in replicas(x), o \rightarrow_i o'$$

P8 $\ll$ is a version order for ρ.

Proof

Let $O(x, \rho)$ be the set of write operation over the data item x during ρ ; and let $i \in replicas(x) \cap correct(\rho)$ (assumption **A1**).

According to Algorithm 2 o is executed only if $trans(o)$ is committed during ρ consequently i commits during ρ any transaction T such that $\exists o \in wo(T), O(x, \rho)$. Consequently $\ll$ is total over $O(x, \rho)$, and by the Total Order and Uniform Agreement properties of Total Order Multicast, $\ll$ is an order over $O(x, \rho)$.

$\square$

P9

$\forall T, T' \in MVSG(\rho, \ll),$
$$((T, T') \in MVSG(\rho, \ll) \wedge wo(T) \neq \varnothing \wedge wo(T') \neq \varnothing) \Rightarrow T \rightarrow T'$$

Proof

F9.1 If (T, T') is a read-from edge, then $T \rightarrow T'$

Let (T, T') be a read-from relation. By definition it exists a site i, a write $w[x] \in T$, and a read $r[x] \in T'$ such that during ρ at site i w write x then r reads the value written by w.

Let t and t' be respectively the times at which these two events occured; according to Algorithm 2 :

F9.1.1 TO-deliver$_i(o) <_\rho t <_\rho t'$

Then since $T' \in MVSG(\rho, \rightarrow)$, $T' \in submitted(r)$, and by assumption **A1**, it exists $j \in replicas(x) \cap correct(r)$ such that TO-deliver$_j(o')$.

Now, TO-deliver$_i(o) \Rightarrow$ TO-deliver$_j(o)$ by the Uniform Agreement, and the Total Order properties of Total Order Multicast. Consequently using fact **F9.1.1**,

$$\neg(\text{TO-deliver}_i(o') <_\rho \text{TO-deliver}_i(o)) \Rightarrow \text{TO-deliver}_j(o) <_\rho \text{TO-deliver}_j(o')$$

concluding our claim.

F9.2 If (T, T) is a version-order edge, then $T \rightarrow T$

Let T_1, T_2, T_3 be three transactions committed during ρ, and suppose that it exists a version-order edge $(T_1, T_2) \in MVSG(\rho, \rightarrow)$.

According to the definition of a version order it follows either:

1. it exists $w_1 \in wo(T_1)$, $w_2 \in wo(T_2)$, and $r_3 \in ro(T_3)$ such that $r_3[x_3]$, $w_1[x_1]$ and $x_1 \ll x_2$.
   By definition of $x_1 \ll x_2 \Rightarrow T_1 \rightarrow T_2$.

2. it exists $r_1 \in ro(T_1)$, $w_2 \in wo(T_2)$ and $w_3 \in wo(T_3)$ such that $r_1[x_3]$, $w_2[x_2]$ and $x_3 \ll x_2$.
   Let $i \in replicas(x) \cap correct(\rho)$ (by assumption **A1**). Since $T_1, T_2, T_3 \in committed(r) \subseteq submitted(r)$, i TO-delivers r_1, w_1 and w_3 during ρ. Now according to the Total Order property of Total Order Multicast, since $x_3 \ll x_2$, $w_3 \rightarrow_i w_2$.
   Let j be a site on which $r_1[x3]$ happens. Since $w_3 \rightarrow_i w_2$, according to our definition of *commit*() (Section 3.8), $w_2 \prec r_1$.
   Now since $T_1 \in committed(\rho)$, necessarily $r_1 \rightarrow_i w_2$ (otherwise T_1 is aborted: line 24).

By facts **F9.1** and **F9.2**

$\square$

P10 ρ is serializable.

Proof

Consider the sub-graph G_u of $MVSG(r, \ll)$ containing all the transactions T such that $wo(T) \neq \emptyset$, and the edge linking them.

F10.1 G_u is acyclic.

Let $T_1, \ldots, T_m \in G_u$ such that $T_1, \ldots, T_{m \geq 1}$ forms a cycle in G_u, and recall that by definition $T_1, \ldots, T_m \in committed(r)$

According to proposition **P9**, $T_1 \rightarrow \ldots \rightarrow T_m \rightarrow T_1$.

Let i be a replica of T_1, and we note t the time at which i commits T during ρ.

Acooording to Algorithm 2 at time t, $closed(T_1, G_{i,t})$.

Now according to Algorithm 1, and since i commits T_1 at time t,

$$\exists k \in [\![2, m]\!], T_k \in breakCycles(pred(T_1, G_{i,t}))$$

Let $j \in replicas(T_k)$ such that j commit T_k during ρ, and let t" be the time at which this event happens.

Since $T_1 \in pred(T_k, G_{j,t'})$ and $T_k \in pred(T_1, G_{i,t})$, $pred(T_1, G_{i,t}) = pred(T_k, G_{j,t'}, .)$

Consequently since $breakCycles()$ is deterministic, j cannot commit T_k during ρ. Absurd.

F10.2 $MVSG(\rho, \rightarrow)$ is acyclic.

By fact **F10.1** and since read only transactions are executed using two-phases locking.

Fact **F10.2** induces that ρ is serializable.

$\square$