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. 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.

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 [23] . In this article we extend the group communication approach to large-scale systems.

Highlights of our protocol:

- Replicas do not reexecute 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.

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 datum 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).
  - 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 r be a run of Π such that a non-empty subset c of g is correct in r. 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 datum 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 by 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

[1] Our taxonomy comes from [5].

		lock held		
		R	*W*	*IW*
lock requested	*R*	1	0	0
	W	0	0	0
	IW	0	0	1

Table 1. Lock conflict table

(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 $replicas(o) \triangleq replicas(item(o))$;

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

$$conflict(o,o') \triangleq \begin{cases} item(o) = item(o') \\ isWrite(o) \vee isWrite(o') \end{cases}$$

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: $item(T) \triangleq \bigcup_{o \in T} item(o)$.
- and $replicas(T) \triangleq replicas(item(T))$.

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]

3.4 Certification phase

When a site i TO-delivers an operation o *for the first time*[4],i removes o from *pending*, and if o is a read, i certifies o. To certify o, i considers any preceding write operations

[2] If T writes a datum 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.

[4] Recall that the leader is eventual, consequently i may receive o more than one time.

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:

$$o' \rightarrow_i o \stackrel{\triangle}{=} \begin{cases} \text{TO-deliver}_i(o') \prec \text{TO-deliver}_i(o) \\ \mathit{conflict}(o', o) \end{cases}$$

Where given two events e and e', we note $e \prec e'$ the relation e *happens-before* e', and $\text{TO-deliver}_i(o')$ the event: "site i TO-delivers operation o'". If such a conflicting operation o' exists, i sets T's abort flag to 1 (see hereafter).

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

$$T \rightarrow T' \stackrel{\triangle}{=} \exists (o, o') \in T \times T', \exists i \in \Pi, o' \rightarrow_i o$$

A precedence graph contains also for each vertex T a abort flag indicating whether T is aborted or not: $\mathit{isAborted}(T, G)$, and the subset of T's operations: $\mathit{op}(T, G)$, 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:

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

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

- $G.\mathcal{V} \subseteq G'.\mathcal{V} \wedge G.\mathcal{E} \subseteq G'.\mathcal{E}$,
- $\forall T \in G.\mathcal{V}$, $\mathit{isAborted}(T, G) \Rightarrow \mathit{isAborted}(T, G')$,
- $\forall T \in G.\mathcal{V}$, $\mathit{op}(T, G) \subseteq \mathit{op}(T, G')$.

Let G be a precedence graph, $\mathit{in}(T, G)$ (respectively $\mathit{out}(T, G)$) 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: $\mathit{pred}(T, G)$, is the precedence graph representing the transitive closure of the dual of the relation $G.\mathcal{E}$ on $\{T\}$.

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

```
1: variable G' := (∅,∅)                                   ▷ a directed graph
2:
3: for all C ⊆ cycles(G) do
4:     if ∀T ∈ C, ¬isAborted(T,G) then
5:         G' := G' ∪ C
6: if T ∈ breakCycles(G') then
7:     return false
8: else
9:     return true
```

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 $G \not\subseteq G_i$, for every transaction T in G_i, such that $pred(T,G) \not\subseteq pred(T,G_i)$, 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:

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

[5] This operation prevents local deadlocks.

Our closure phase ensures that during every run r, 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: $wo(T) = \emptyset$, 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 our technical report [22].

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 operation o accessing a datum it does not replicate, i sends o to some site $j \in replicas(o)$. Upon reception j executes o. At the end of execution, j sends back to i the transitive closure starting from o's read-from dependency.

Once i has executed locally or remotely all the read operations, i checks if the resulting read-from dependancies graph contains a cycle 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, the update values, and R-multicasts it with the read-from dependancies graph to $replicas(T)$. The dependencies are merged to precedence graphs when sites TO-deliver T's operations. The rest of the algorithm remains the same.

3.10 Performance analysis

Precedence constraints in a cycle are *not* causally related. Moreover our algorithm handles cycles of size 2 without executing lines 31 to 35. Consequently it is unlikely that closing transactions in a cycle requires additional steps, and we do not suppose it hereafter.

We consider Paxos [14] as a solution to Uniform Total Order Multicast . Algorithm 2 achieves a latency degree[6] of 4: 1 for Uniform Reliable Multicast, and 3 for Uniform Total Order Multicast. Let o be the number of operations per transaction, and d the replication degree, the message complexity of Algorithm 2 is $4od + (od)^2$: od for Uniform Reliable Multicast, o Uniform Total Order Multicasts, each costing $3d$ messages, and od replicas execute line 29, each site sending od messages.

[6] Maximum length of the causal path to commit a transaction in the best run.

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** $isRead(o) \wedge \exists o', o' \rightarrow_i o$ **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:

Algorithms totally ordering transactions [17,11,19] achieve at least a latency degree of 3, and a message complexity of $3n$. Totally ordering transactions requires to contact $n/2$ sites, whereas our approach needs only to contact *od* sites; it reduces latency. Moreover in large scale systems we expect $od << \sqrt{3n}$, and consequently our algorithm achieves better message complexity than total order based solutions.

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 [23]. 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-bizantine 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].[7] We believe that the closure generalizes to a wider context, where a constraint is a temporal logic formula over sequences of concurrent operations.

[7] 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.

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. JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. Partial database replication using epidemic communication. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 485, Washington, DC, USA, 2002. IEEE Computer Society.
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.
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. 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. P. Sutra and M. Shapiro. Fault-tolerant partial replication in large-scale database systems, http://hal.inria.fr/inria-00232662/fr/. Technical report.
23. 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.