AKARA: A Flexible Clustering Protocol for Demanding Transactional Workloads

A. Correia Jr. J. Pereira R. Oliveira

Computer Science and Technology Center (CCTC)
University of Minho

Abstract

Shared-nothing clusters are a well known and cost-effective approach to database server scalability, in particular, with highly intensive read-only workloads typical of many 3-tier web-based applications. The common reliance on a centralized component and a simplistic propagation strategy employed by mainstream solutions however conduct to poor scalability with traditional on-line transaction processing (OLTP), where the update ratio is high. Such approaches also pose an additional obstacle to high availability while introducing a single point of failure.

More recently, database replication protocols based on group communication have been shown to overcome such limitations, expanding the applicability of shared-nothing clusters to more demanding transactional workloads. These take simultaneous advantage of total order multicast and transactional semantics to improve on mainstream solutions. However, none has already been widely deployed in a general purpose database management system.

In this paper, we argue that a major hurdle for their acceptance is that these proposals have disappointing performance with specific subsets of real-world workloads. Such limitations are deep-rooted and working around them requires in-depth understanding of protocols and changes to applications. We address this issue with a novel protocol that combines multiple transaction execution mechanisms and replication techniques and then show how it avoids the identified pitfalls. Experimental results are obtained with a workload based on the industry standard TPC-C benchmark.

1 Introduction

Database replication based on group communication [8, 14, 13, 19, 9, 12] supplies the foundation for affordable and scalable clusters that eschew a shared storage infrastructure. By enabling the use of commodity machines and a variety of database solutions, it helps reduce costs associated with building fault-tolerant architectures and easies the scale-out factor [15, 3]. The result also improves on reliability when compared to most mainstream solutions, as these often reduce to lazy update or rely on centralized components.

Generically, the approach is eager and builds on the classical replicated state machine [16]: The exact same sequence of update operations is applied to the same initial state, thus producing a consistent replicated output and final state. The problem is then to ensure deterministic processing without overly restricting concurrent execution, which would dramatically reduce throughput.

This is achieved by executing the bulk of the transaction at a single replica and then propagating raw updates, in a passive replication, which has the additional advantage of avoiding re-execution. A single total order broadcast for each transaction suffices for coordination, thus being able to achieve close to linear scalability even with write-intensive loads [7]. In contrast, eager replication based on distributed locking and atomic commit protocols, require much finer grained coordination and fall prey of deadlocks [6].

Protocols differ mainly in whether transactions are executed optimistically [13, 8] or conservatively [14]. In the former, a transaction is executed by a receiving replica without a priori coordination with other replicas. Just before committing, replicas coordinate and check for conflicts between concurrent transactions. Transactions that would locally commit may abort due to conflicts with remote concurrent transactions. In the conservative approach, all replicas first agree on the execution order for (potentially) conflicting transactions assuring that when a transaction executes there is no concurrent conflicting transaction being executed remotely and therefore its success depends entirely on the local database engine. Clearly, two transactions conflict if both access the same conflict class (e.g. table) and one of them update it.

Despite the promising benchmark results, the practicality of such protocols is limited as all have disappointing performance with specific subsets of demanding real-world workloads. Namely, the optimistic approach may become impractical as long-running transactions may experiment unacceptable abort rates. This makes it very hard to commit such transactions in a heavily loaded server, even when resubmission is possible, thus compromising liveness. This issue does not arise with conservative protocols. However, achieving good performance may require a careful application-specific definition of conflict classes, if possible at all, without changes to application semantics [7]. This is particularly troublesome as a labeling mistake can lead to inconsistency.

Furtheremore, simple statements that update large number of items result in heavy network traffic in both approaches, while saving little in avoiding re-execution. The same is true for DDL statements (e.g. create index, alter table), in which extracting and applying updates may require intimate knowledge of database internals and also yield large updates. In these situations active replication is desirable.

The challenge is therefore to combine the ease of use of the optimistic approach, with the fairness of the conservative approach and the straightforward implementation of the active replication. This paper thus shows how it is possible to overcome this challenge and is organized as follows. Section 2 surveys the current protocols focusing on dynamic aspects, namely, on queuing that happens in different parts of the system and on the amount of concurrency that can be achieved.

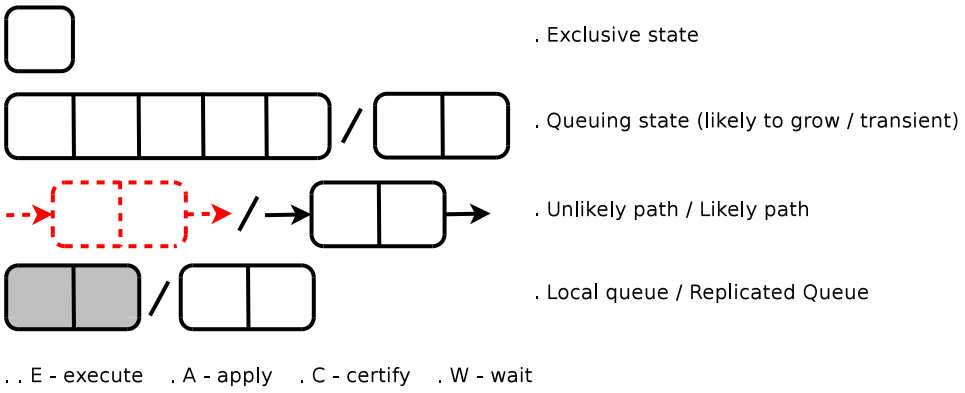


Figure 1: Notation.

Section 3 proposes the novel AKARA database replication protocol based on group communication. As happens with conservative protocols, AKARA is fair and takes advantage of a judicious definition of conflict classes to maximize concurrency. However, to attain the performance level of optimistic protocols, AKARA exploits the tentative execution of potentially conflicting transactions as allowed by the underlying system, i.e. by any database management system. Furthermore, as in active replication protocols, AKARA allows any deterministic statement to be actively replicated. Section 4 evaluates AKARA using the workload from TPC-C and show that it provides, even with a generic application independent (i.e. strictly syntactic) definition of conflict classes, peak performance comparable with a purely optimistic protocol while at the same time enforcing fairness. Section 5 discusses open issues and Section 6 concludes the paper.

2 Background

In this section we make a brief overview of major approaches to database replication using group communication. We do this survey however with a novel twist. We focus on dynamic aspects, namely, (i) on queuing that happens in different parts of the system and (ii) on the amount of concurrency that can be achieved. Then, we contrast the original assumptions underlying such protocols with our experience with actual implementations and using the TPC-C workload [7]. This is extremely relevant, as previous protocols have been proposed as exploiting optimistic assumptions on system dynamics that we are not able to confirm in our realistic setting. The conclusion sets the motivation for proposing the AKARA protocol in Section 3.

Figure 1 introduces the notation used to depict protocol state-machines. Given the emphasis on dynamic aspects, we use different symbols for states that represent queuing and states in which at most a single non-conflicting transaction can be at any given time. We show also which queues are likely to grow without bound when

the system is congested. When alternative paths exist, due to optimistic execution, we show which is considered to be the more likely to be executed. We make a distinction between local and replicated queues and identify relevant actions: execute, apply, certify and wait.

Since all protocols involve an atomic broadcast step, we use a consistent naming for queues in different protocols. Q0 is before the abcast, Q1 is between abcast and delivery, and Q2 is after delivery. Protocols with an optimistic assumption use messages in Q1 which has messages with tentative order, i.e. messages that were optimistically delivered. In contrast, messages in Q2 have a final order.

2.1 Non-Disjoint conflict classes and Optimistic multicast(NODO)

In NODO, data is a priori partitioned in conflict classes, not necessarily disjoint. Each transaction has an associated set of conflict classes (the data partitions it accesses) which are assumed to be known in advance. This approach requires to know the entire transaction before its execution precluding the processing of interactive transactions.

When a transaction is submitted (Q0), its id and conflict classes are atomically multicast to all replicas obtaining a total order position (Q2). Each replica has a queue associated with each conflict class and, once delivered, a transaction is classified according to its conflict classes and enqueued in all corresponding queues. As soon as a transaction reaches the head of all of its conflict class queues it is executed. Transactions are executed by the replica to which they are submitted.

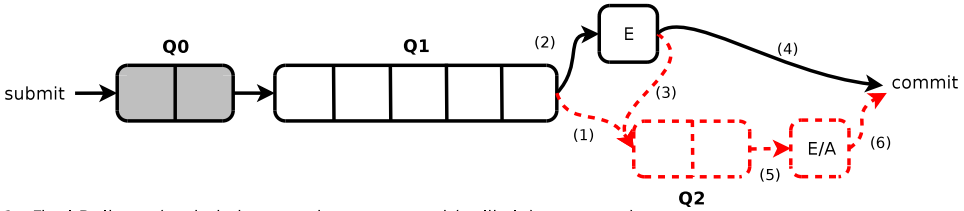
Clearly, the conflict classes have a direct impact on performance. The fewer the number of transactions with overlapping conflict classes, the better the interleave among transactions. Conflict classes are usually defined at the table level but can have a finer grain at the expense of a non-trivial validation process to guarantee that a transaction does not access conflict classes that were not previously specified.

When the commit request is received, the outcome of the transaction is reliably multicast to all replicas along with the replica's changes (write-set) and a reply is sent to the client. Each replica applies the remote transaction's updates with the parallelism allowed by the initially established total order of the transaction.

The protocol ensures 1-copy serializability [2] as long as transactions are classified taking into account read/write conflicts. To achieve 1-copy snapshot isolation [9] transactions must be classified take into account just write/write conflicts.

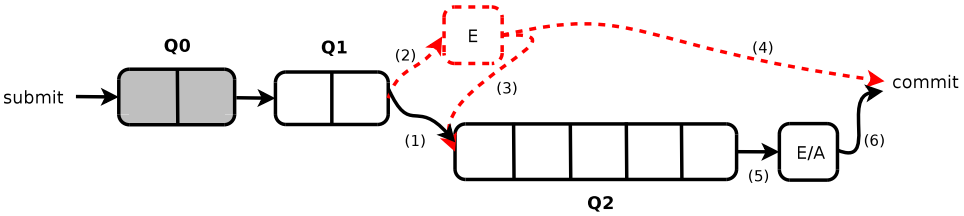
A transaction is scheduled optimistically if there is no conflicting transaction already ordered (Q2). This tentative execution may be done at the expense of an abort if a concurrent transaction is later on ordered before it.

Figure 2(a) shows the states that a transaction goes through upon being submitted by a client. According to the designers' assumptions, the time spent in the queue waiting for total order (Q1) is significant enough compared to time taken to actually execute such that it is worthwhile to optimistically execute transactions (transition 2 instead of transition 1). This makes it possible that when a transaction



1 - Final Delivery (optimistic execution not started (unlikely) or remote)
2 - Submit transaction to optimistic exectuion
3 - Final Delivery (missed order and then rollback (unlikely))
4 - Final Delivery (correct order)
5 - Submit transaction to execution in order
6 - Execution finished in order

(a) Assumption of the designers.



1 - Final Delivery (optimistic execution not started or remote)
2 - Submit transaction to optimistic exectuion (unlikely)
3 - Final Delivery (missed order and rollback (unlikely, but it doesn't matter))
4 - Final Delivery (correct order)
5 - Submit transaction in order
6 - Execution finished in order

(b) Our assumption.

Figure 2: States, transitions, and queues in NODO.

is ordered, it is immediately committed (transition 4). Assuming that optimistic ordering is correct, a rollback (transition 3) is unlikely.

We have however reasons to believe that this assumption is invalid. The first hints for these are actually in the original description of the proposal [14]. First, the end-to-end transaction execution latency measured is larger than 50 ms, of which most surely only less than 10% can be attributed to the latency of group communication. In fact, the protocol used for experiments, is very well known for its extremely good performance [1]. If this is true, then queuing will happen in queue Q2 and not in queue Q1. Thus if Q2 is never empty, then no transaction in queue Q1 is eligible for optimistic execution. This is confirmed by the abort rate being always extremely low, even with a large share of conflicting update transactions [14], a hint that transition 2 is never taken.

The appropriate scenario for the NODO protocol is thus depicted in Figure 2(b): The optimistic path is seldom used and the protocol boils down to a coarse-grained distributed locking approach, which has a very large impact in scalability. Notice that if there are k (disjoint) conflict classes, there can be at most k transactions executing in the whole system. Again, this seems to be confirmed by the original presentation of the protocol (Figure 6 of [14]): With an update intensive load and $k = 16$ distinct conflict classes, the scale factor for 15 nodes is just five-fold ($5\times$). Although this is attributed to saturating system resources, one cannot know for sure as it is not measured. One should expect that if k is smaller, this result is even worse.

Our experiments using the TPC-C workload confirm these hints. Figure 7 show the NODO protocol saturating when there are still plenty of system resources available. Although our implementation does not have the optimistic functionality, queue Q2 is always large and thus the optimistic path would not be used anyway.

2.2 Active replication

Active replication is a technique to build fault-tolerant systems in which transactions are deterministically processed at all replicas and as such requires that each transaction's statement be processed by the same order at them. This might be ensured by means of a centralized or a distributed scheduler.

Sequoia 4.x [4], which was built after the C-JDBC [3], for instance, uses a centralized scheduler at the expense of introducing a single point of failure. Usually, any distributed scheduler would circumvent this resilience problem but would require a distributed deadlock detection mechanism. To avoid the distributed deadlocks, one might annotate transactions with conflict-classes and request distributed locks through an atomic multicast before starting executing a transaction. In contrast with NODO, however, a reliable message to propagate changes would not be needed as transactions would be actively executed. In both approaches, the consistency criteria would be similar to those provided by NODO.

The case against the active replication is shown in the NODO paper: unbearable contention with high write ratio. This technique additionally has the draw-

back of requiring a parser to remove non-deterministic information (e.g. random() or date()), thereby leading to re-implement several features already provided by a database management system.

The active replication pays off when the overhead between transferring raw updates in a passive replication is higher then re-executing statements. And of course, it makes it easy to execute DDL statements.

2.3 Database State Machine(DBSM) and Postgres-R(PGR)

In both protocols, transactions are immediately executed by the replicas to which they are submitted without any a priori coordination. Locally, transactions are synchronized according to the specific concurrency control mechanism of the database engine.

Upon receiving a commit request, a successful transaction is not readily committed. Instead, its changes (write-set) and read data (read-set) are gathered and a termination protocol initiated. The goal of the termination protocol is to decide the order and the outcome of the transaction such that a global correctness criterion is satisfied (e.g 1-copy serializability [2] or 1-copy snapshot isolation [9]). This is achieved by establishing a total order position for the transaction and certifying it against concurrently executed transactions. The certification of a transaction is done by evaluating the intersection of its read-set and write-set (or just write-set in case of the snapshot isolation) with the write-set of concurrent, previously ordered transactions. The fate of a transaction is therefore determined by the termination protocol and a transaction that would locally commit may end up aborted.

These protocols differ on the termination procedure. Considering 1-copy serializability, both protocols use the transaction's read-set in the certification procedure. In the PGR, the transaction's read-set is not propagated and thus only the replica executing the transaction is able to certify it. In the DBSM, conversely, the transaction's read-set is propagated allowing each replica to autonomously certify the transaction.

In detail, upon the reception of the commit request for a transaction t, in PGR the executing replica atomically multicasts t's id and t's write-set. As soon as all transactions ordered before t are processed, the executing replica certifies t and reliably multicasts the outcome to all replicas. The certification procedure consists in checking t's read-set and write-set against the write-sets of all transactions ordered before t. The executing replica then commits or aborts t locally and replies to the client. Upon the reception of t's commit outcome each replica applies t's changes through the execution of a high priority transaction consisting of updates, inserts and deletes according to t's previously multicast write-set. The high priority of the transaction means that it must be assured of acquiring all required write locks, possibly aborting any locally executing transactions. In other words, if t does not end up aborted by a high priority transaction, it is transparently and indirectly certified what we entitle an in-core certification.

The termination protocol in the DBSM is significantly different and works as

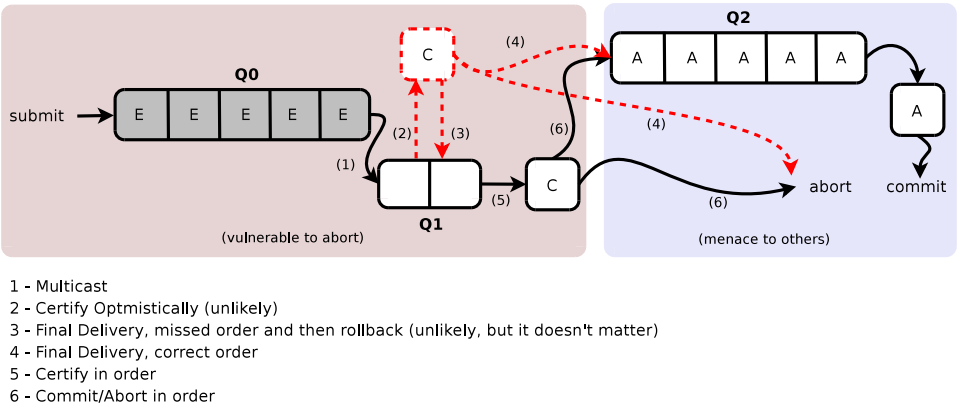


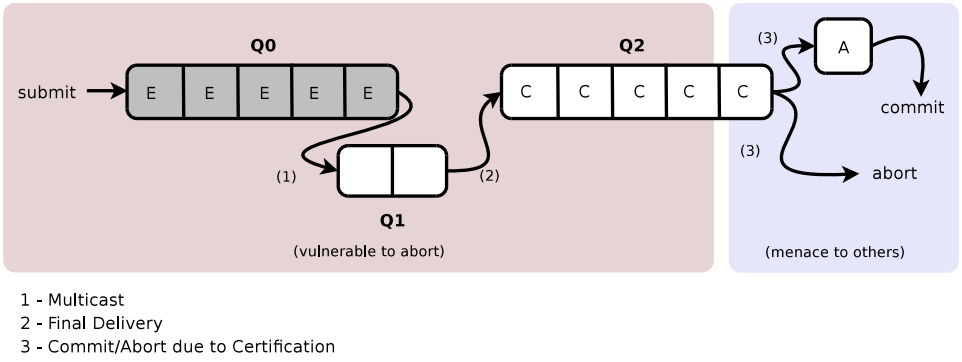
Figure 3: States, transitions, and queues in DBSM (our assumption).

follows. Upon the reception of the commit request for a transaction t, the executing replica atomically multicasts t's id, the version of the database on which t was executed, and t's read-set and write-set. As soon as t is ordered, each replica is able to certify t on its own. For the certification procedure, t's read-set and write-set are checked against the write-sets of all transactions committed since t's database version. If they do not intersect, t commits, otherwise t aborts. If t commits then its changes are applied through the execution of a high priority transaction consisting of updates, inserts and deletes according to t's previously multicast write-set. Again, the high priority of the transaction means that it must be assured of acquiring all required write locks, possibly aborting any locally executing transactions. The executing replica replies to the client at the end of t.

In both protocols, transactions are queued while executing, as would happen in a non-replicated database, using whatever native mechanism is used to enforce ACID properties. This is queue Q0 in Figures 3 and 4.

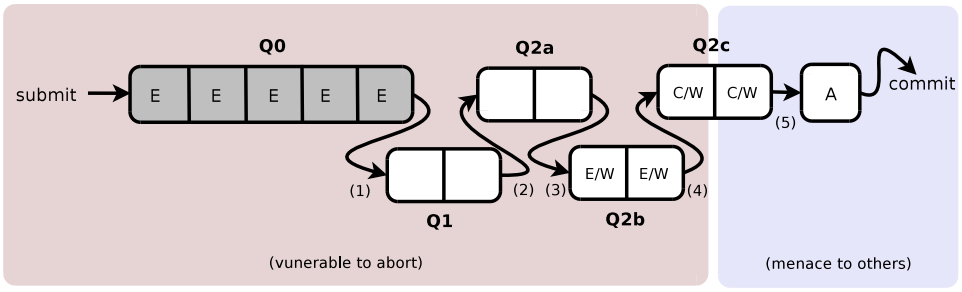
The most noteworthy feature of both protocols is that ever since a transaction starts until it is certified, it is vulnerable to being aborted by a concurrent transaction that gets to commit and write a conflicting item. On the other hand, from the instant that a transaction is certified until it finally commits on every node, it is a menace to other transactions which will be aborted if they touch a conflicting item. Latency in any processing stage is thus bound to increase the abort rate. A side-effect of this is that the resulting system, when loaded, is extremely unfair to long running transactions.

In the DBSM, the initial assumption was that the only added latency introduced by replication was in the atomic multicast step, similarly to NODO (Q1) in Figure 2(a). PGR [8] does not use optimistic delivery. However, this is only an issue in WANs. In clusters, latency comes from exhausting resources within each replica as queues build up in Q0 and Q2. It is thus no surprise that any contention whatsoever makes the abort rate shoot up.



1 - Multicast
2 - Final Delivery
3 - Commit/Abort due to Certification

Figure 4: States, transitions, and queues in PGR.



1 - Pre-classification and multicast
2 - Final Delivery
3 - Scheduler: scheduled to run optimistically or wait if remote or active
4 - Optmistic execution finished or simply next in line
5 - Next in line

Figure 5: States, transitions, and queues in AKARA.

3 The AKARA Protocol

3.1 Intuition

The goal of AKARA is three-fold: maximize resource usage by scheduling sufficient concurrent executions (avoiding the pitfall of NODO) while at the same time keeping queuing outside the danger zones thus ensuring fairness (avoiding the pitfalls of DBSM) and overcome a profound limitation of both NODO and DBSM by allowing seamless active execution.

Figure 5 depicts the major states, transitions, and queues of this protocol. Let us assume that conflict classes are tables and, for simplicity, that all transactions access at least a common table. In Section 3.2, we relax this assumption and also consider the case that transactions have no conflict classes in common.

Upon submission, transactions are classified according to a set of conflict classes and totally ordered by means of an atomic multicast primitive. This global order

allows to prevent conflicting transactions to run concurrently. Once ordered, a transaction is queued into Q2a waiting to be scheduled. Progression in Q2a depends on an admission control policy. When a transaction reaches the top of Q2a it is transferred to Q2b and then executed. Transactions executed while in Q2b are said to be run optimistically as they may end up aborting due to conflicts with concurrent transactions in Q2b or Q2c. After execution, and having reached the top of Q2b, a transaction is transferred to Q2c. When a transaction reaches the top of Q2c it may be ready to commit or not (because it had to abort due to conflicts). If it is ready to commit, its changes are propagated to all other replicas and the transaction commits. Otherwise, the transaction is re-executed conservatively by imposing its priority on any locally running transaction.

AKARA maximizes resource usage through the concurrent execution of potentially conflicting transactions by means of an admission control mechanism. It is worth noticing however that an admission policy that only allows to execute non-conflicting transactions according to their conflict classes makes AKARA to fall down as a simple conservative protocol as NODO. The key is therefore to judiciously schedule the execution of each transaction in order to exploit idleness thus reducing contention introduced by a conservative execution while at the same time avoiding re-execution. We assume here a policy that just allows to optimistically execute n transactions in parallel. The analysis of more sophisticated policies is not target in this paper as this simple policy suffices to show the effectiveness of our novel protocol.

Such optimistic executions however may lead to local deadlocks. Consider two conflicting transactions t and $t\prime$ that are ordered $< t, t\prime >$ and scheduled to run concurrently (both are in Q2b). If $t\prime$ grabs a lock first on a conflicting data item, it prevents t from running. However $t\prime$ cannot leave Q2b before t without infringing the global commit order. Two extreme solutions for this problem are:

- Roll back right after execution, reapplying updates later on if no conflicts arise. This has a serious drawback as it imposes a severe overhead even when conflicts are unlikely or even nonexistent. And, when there are conflicts, it always implies a re-execution.

- The other solution is to abort a transaction that gets to the top of the queue (that is, reaches its commit order) if a subsequent transaction must finish execution before it. This has however the severe drawback that it prevents many non-conflicting transactions to be executed simultaneously, decreasing the value of the optimistic execution.

If both transactions have the same conflict classes and, of course, are locally executed at the same replica, a better alternative is to allow t' to overtake t in the global commit order. Notice that when a transaction t is totally ordered this ensures that no conflicting transaction will be executed concurrently at any other replica. Therefore, if t's order is swapped with that of a $t\prime$ with the very same conflict classes then it is still guaranteed that both t and $t\prime$ are still executed without the

interference of any remote conflicting transaction. In the experiments conducted in Section 4 with the TPC-C, the likelihood of having two transactions with the very same conflict classes is high as more than 85% of the ocurrences are due to the *NewOrder* and *Payment* transactions.

Finally, the AKARA protocol also allows transactions to be actively executed thus providing a mechanism to easily replicate DDL statements and to reduce network usage. This execution steps are detailed in the next section.

3.2 Algorithm

```
1  Q0, Q2a, Q2b, Q2c: sets;
2  function submit(t)
3      put t into Q0;
4      t.type = compute_type(t);
5      t.cc = compute_classes(t);
6      abcast(t.type,t);
7  end
8  upon deliver(passive,t) to self
9      put t into Q2a;
10     wait (t = next(Q2a, t.cc) ∧
11       scheduled(t));
12     put t into Q2b;
13     execute t;
14 end
15 upon (t is local ∧
16   t is executed ∧
17   t = next(Q2b, t.cc))
18     put t into Q2c;
19     wait (t = next(Q2c, t.cc));
20     if (t is not ready to commit)
       then
21         execute t with priority;
22     rbcast (t.updates, t);
23     commit t;
24     remove t from Q2c;
25 end
26 function next(Q,cc) ≡ t ∈ Q st.
27     t.seq = min ({t'.seq | t' ∈
       Q ∧ t'.cc ∩ cc})
28 end
29 upon deliver(passive,t) to others
30     put t into Q2a;
31     wait (t = next(Q2a, t.cc));
32     put t into Q2b;
33     wait (t = next(Q2b, t.cc));
34     put t into Q2c;
35     wait (t = next(Q2c, t.cc) ∧
36       t.updates were delivered);
37     apply t.updates with
       priority;
38     commit t;
39     remove t from Q2c;
40 end
41 upon deliver(active,t)
42     put t into Q2a;
43     wait (t = next(Q2a, t.cc));
44     put t into Q2b;
45     wait (t = next(Q2b, t.cc));
46     put t into Q2c;
47     wait (t = next(Q2c, t.cc));
48     execute t with priority;
49     commit t;
50     remove t from Q2c;
51 end
52 upon (t, t' are local ∧ t ≠ t' ∧
53   t is ready to commit ∧
54   t' = next(Q2b, t.cc))
55     if (t.cc == t'.cc) then
56         swap(t.seq, t'.seq);
57     else abort t;
58 end
```

Figure 6: AKARA algorithm.

The AKARA algorithm is presented in Figure 6. In that, a transaction is rep-

resented by a data structure containing the following information: seq - a global sequence number which corresponds to the total order established by the atomic multicast; cc - the transaction's estimated set of conflict classes; $type$ - whether the transaction should be passively or actively executed. Although not explicitly used in the algorithm of Figure 6, we assume that this data structure also contains the transaction's write-set.

Each replica maintains different sets: $Q0$, $Q2a$, $Q2b$ and $Q2c$ whose utilization were introduced in Sections 2 and 3 and shall be detailed next.

Once a transaction is submitted, $submit(t)$ is invoked. The transaction t is put into $Q0$, which used to store transactions before any coordination action is carried on. Right after, an external function (line 4) is used to compute the type of t: $passive$ or $active$. Then another external function (line 5) classifies t with respect to its conflict classes. [1] Once t is classified, it is atomically multicast to all replicas (line 6). Upon delivery (lines 8, 29 and 41), t is put into $Q2a$ and $t.seq$ is set. This gives t its commit order, which is total with respect to all its conflicting transactions. It is worth noticing that we omitted $Q1$ here as we do not exploit fast delivered transactions.

Assuming a passive execution (line 8), the initiating replica waits until t can be the next in $Q2a$ to be transferred to $Q2b$ and a scheduler decides to optimistically execute it (line 10). In particular, the function $next(Q, cc)$ (line 26) looks at a queue, in this case $Q2a$, and retrieves information on conflicting transactions. If there is a conflicting transaction ordered before t, i.e. $t \neq next(Q2a, t.cc)$, t waits for its turn. Otherwise, it can be removed from $Q2a$ and proceed.

Once the previous condition is achieved (line 10), t is put into $Q2b$ and its execution is started. From this moment until t can be removed from $Q2c$, it is vulnerable to be aborted by a remote high priority transaction. Therefore it may terminate its execution either upon requesting a commit or due to an abort requested by a conflicting and remote high priority transaction. In the former case, it is marked as ready to commit.

One needs to wait until t is executed and can be removed from $Q2b$ (line 15). However, due to interleaves of concurrent events inside a database, a transaction $t\prime$ ordered before t may be blocked by t thus not being able to make progress and not allowing t to be removed from $Q2b$ and proceed. To overcome this problem, the algorithm (lines 52–58) allows t to overtake $t\prime$ in the global commit order, when both have the same conflict classes and belong to the same replica. Otherwise, it aborts t.

Once the previous condition is achieved (line 15), t is put into $Q2c$. When t can be removed from $Q2c$, its write-set is reliably multicast to all replicas if it is still ready to commit. Otherwise, t is executed as a high priority transaction and right after its write-set is reliably multicast to all replicas. Finally, t is committed at the initiating replica and removed from $Q2c$.

At a remote replica, the execution of a transaction t is straightforward (line 29).

[1] See Section 5 for a brief discussion on these functions.

When t can be removed from $Q2a$, it is immediately moved to $Q2b$, and so forth, until it gets to $Q2c$. When t can proceed from $Q2c$ and its write-set is delivered, it is applied on the replica with a high priority, committed and them removed from $Q2c$.

A transaction t marked as active is executed at all replicas without distinction between a initiating or a remote replica, and its execution is straightforward (line 41). When t can be removed from $Q2a$, it is immediately moved to $Q2b$, and so forth, until it gets to $Q2c$. When t can proceed from $Q2c$, it is executed with a high priority, committed and them removed from $Q2c$. Active transactions are not executed optimistically to avoid different interleaves at different replicas.

4 Evaluation

4.1 Simulation Environment

The simulation environment is based on a centralized simulation model that combines real software components with simulated hardware, software and environment components to model a distributed system. This allows us to setup and run multiple realistic tests with slight variations of configuration parameters that would otherwise be impractical to perform, specially if one considers a large number of clients and replicas [20].

The key components, the replication and the group communication protocols, are real implementations while both the database engine and the network are simulated.

The simulation environment represents a LAN with 9 replicas connected by a network with a bandwidth of 1Gbps and a latency of 120μs. Each replica corresponds to a dual processor AMD Opteron at 2.4GHz with 4GB of memory, running the Linux Fedora Core 3 Distribution with kernel version 2.6.10. For storage we used a fiber-channel attached box with 4, 36GB SCSI disks in a RAID-5 configuration and the Ext3 file system. The database running is a PostgreSQL 7.4.6 with snapshot isolation and the global consistency criterion is 1-copy snapshot isolation [9].

Clients run an implementation that mimics the industry standard on-line transaction processing benchmark TPC-C [21]. TPC-C specifies five transactions: *NewOrder* with 44% of the occurrences; *Payment* with 44%; *OrderStatus* with 4%; *Delivery* with 4%; and *StockLevel* with 4%. The *NewOrder*, *Payment* and *Delivery* are update transactions while the others are read-only.

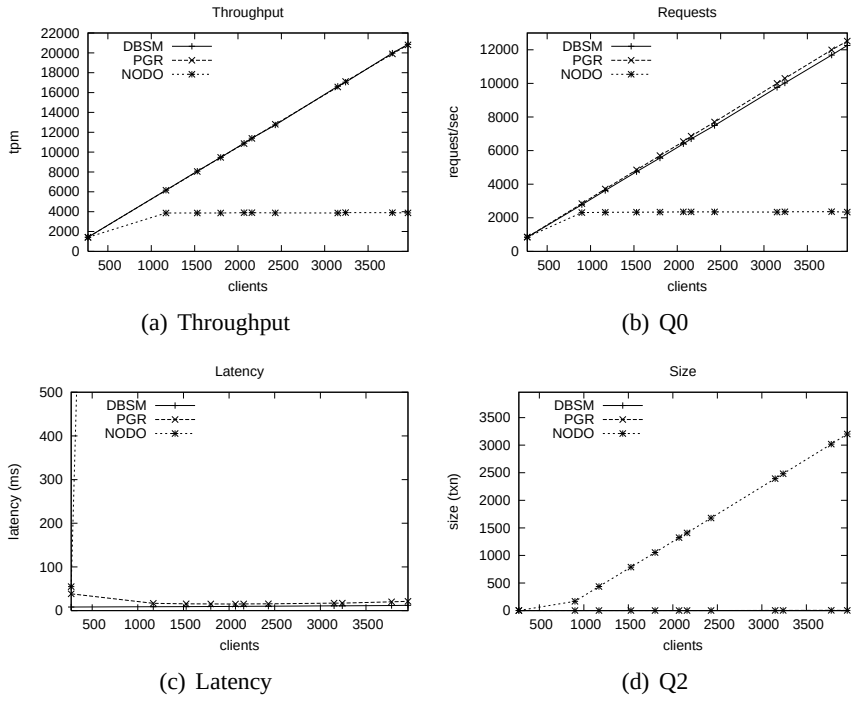
For the experiments in Section 4.2, we added to the benchmark three more transactions that mimic maintenance activities such as adding users, changing indexes in tables or updating taxes over items. Specifically, the first transaction *Light-Tran* creates a constraint on a table if it does not exist or drops it otherwise. The second transaction *Active-Tran* increases the price of products and is actively executed. Conversely, *Passive-Tran* does the same maintenance activity but its changes are passively propagated. These transactions are never executed in the same run,

have a probability of 1% and when are executing the probability of the *NewOrder* is reduced to 43%.

We varied the total number of clients from 270 to 3960 and distributed them evenly among the replicas and each run has 150001 transactions.

4.2 Results

The first set of experiments evaluate the DBSM, NODO and PGR approaches. In the NODO approach, we use the simple definition of a conflict class for each table, what can be easily extracted from the SQL code. Figures 7 and 8 compare the DBSM, PGR and NODO.



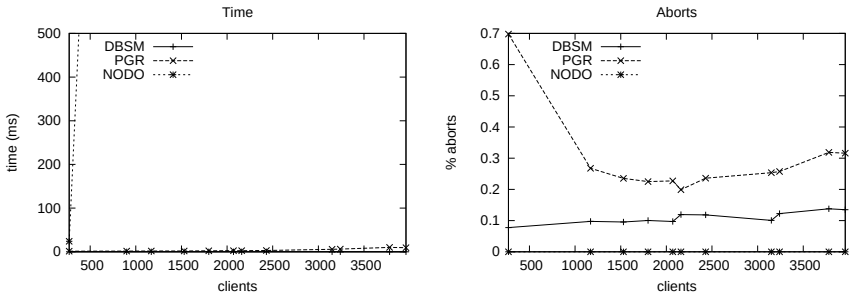
(a) Throughput (b) Q0

(c) Latency (d) Q2

Figure 7: Performance of DBSM, PGR and NODO.

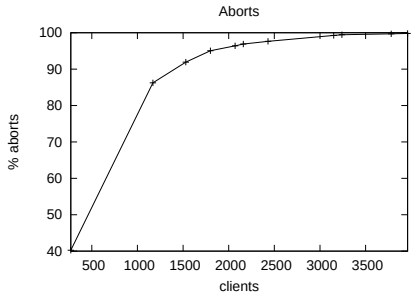
The DBSM and PGR show a throughput higher than 20000 tpm (Figure 7(a)). In fact, both present similar results and the higher the throughput the higher the number of requests per second inside the database (Figure 7(b)). These requests represent access to the storage, CPU, lock manager and to the replication protocol. Clearly, the database is not a bottleneck. In contrast, the throughput presented by NODO is extremely low, around 4000 tpm, and its latency is extremely high (Figure 7(c)). This drawback can be easily explained by the contention observed in $Q2$ (Figure 7(d)).

Unfortunately, with the conservative and optimistic approaches presented above, one may have to choose between latency and fairness. In the NODO, for 3240



(a) Time in Q2

(b) Abort



(c) Delivery's Abort in DBSM

Figure 8: Latency versus Abort (DBSM, PGR and NODO).

clients, 2481 transactions wait in $Q2$ around 40 s to start executing (Figure 8(a)). In contrast, an optimistic transaction waits 1000 times less and the number of transactions waiting to be applied is very low.

The abort rate is below 1% in both optimistic approaches as there is no contention and the likelihood of conflicts is low in such situations (Figure 8(b)). However, to show that the optimistic protocols may not guarantee fairness, we conducted a set of experiments in which one requests an explicit table level locking on behalf of the *Delivery* transaction thus mimicing a hotspot. This is a pretty common situation in pratice, as application developers may explicitly request locks to improve performance or avoid concurrency anomalies. In this case, the abort rate is around 5% and this fact does not have an observable impact on latency and throughput but almost all Delivery Transactions abort, around 99% (Figure 8(c)). In [7], a table level locking is acquired on behalf of the *Delivery* transaction to avoid flooding the network and improve the certification procedure. Although the reason to do so is diferent, the issue is the same.

In all the experiments, the time between an optimistic delivery and a final delivery were always below 1 ms, thus excluding $Q1$ from being an issue.

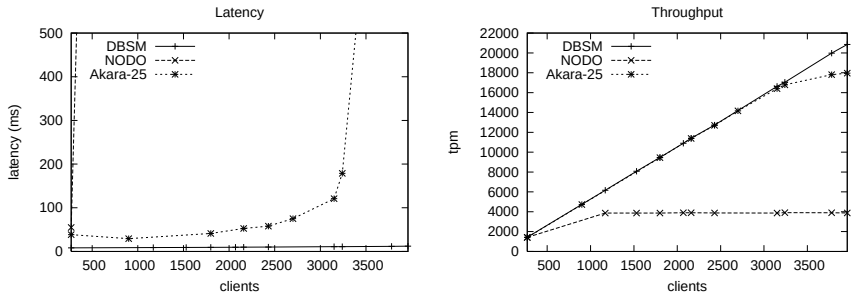
To improve the performance of the conservative approach while at the same time guaranteeing fairness, we used the AKARA protocol. We ran the AKARA protocol varying the number of optimistic transactions that might be concurrently submitted to the database in order to figure out which would be the best value for

	Latency (ms)	Throughput (tpm)	Unsuccess rate (%)
AKARA-25	178	16780	2
AKARA-45	480	16474	5
AKARA-n	37255	3954	89
AKARA-25 with Light-Tran	8151	9950	21
AKARA-25 with Active-Tran	109420	1597	21
AKARA-25 with Passive-Tran	295884	625	22

Table 1: Analysis of AKARA.

our environment. This degree of optimistic execution is indicated by a number after the name of the protocol. For instance, AKARA-25 means that 25 optimistic transactions might be concurrently submitted and AKARA-n means that there is no restriction on this number.

Table 1 shows that indefinitely increasing the number of optimistic transactions that might be concurrently submitted is not worth. Basically for AKARA-n, latency drastically increases and as a consequence throughput decreases. This occurs because the number of transactions that fails the certification procedure increases. For 3240 clients, more than 89% of the transactions fail the certification procedure (i.e. in-core certification procedure like in PGR, see Section 2.3). Furthermore, after failing such transactions are conservatively executed and compete for resources with optimistic transactions that may be executing. Keeping the number of optimistic transactions low however reduces the number of transactions allowed in the database and neither is worth. After varying this number from 5 to 50 in steps of 1, we figured out that the best value for the TPC-C in our environment is 25.



(a) Latency (b) Throughput

Figure 9: DBSM, NODO and AKARA-25.

In what follows, we used the DBSM as the representative of the family of optimistic protocols thus omitting the PGR. Although both protocols present similar performance in a LAN, the PGR is not worth in a WAN due to its extra communication step [7].

Figure 9 depicts the benefits provided by the AKARA-25. In Figure 9(a), we

notice that latency in the NODO is extremely high. In contrast, the AKARA-25 starts degenerating after 3240 clients. For 3240 clients the latency in the DBSM is about 9 ms, and in the AKARA-25, it is about 178 ms. This increase in latency directly affects throughput as shown in Figure 9(b). The NODO presents a steady throughput of 4000 tpm; the AKARA-25, a steady throughput of 18605 tpm after 3960 clients; while the DBSM increases its throughput almost linearly. The DBSM starts degenerating when the database becomes a bottleneck what was not our goal with these experiments.

Table 1 shows the impact on performance when the maintenance activities are handled by our protocol. These maintenance activities represented by the transactions *Active-Tran* and *Light-Tran* are actively executed and integrated in runs with the *AKARA-25*: *AKARA with Active-Tran* and *AKARA with Light-Tran*, respectively. In order to show the benefits of an active execution in such scenario, we provide a run named *AKARA with Passive-Tran* in which the updates performed by the *Active-Tran* are atomically multicast. The run with the *Passive-Tran* presents a latency higher than that with the *Active-Tran* as the former needs to transfer the updates through the network. However, both approaches have a reduced throughput and high latency when compared to the normal AKARA-25 due to contention caused by a large number of updates.

The run with the *Light-Tran* does not have a large number of updates but its throughput decreases when compared to the AKARA-25 due to failures in the certification procedure. This is caused by the fact that the transaction *Light-Tran* mimics a change on the structure of a table and thus requires an exclusive lock on it.

In a real environment, we expect that maintenance operations occur with a rate lower than 1% and so they should not be a problem as the optimistic execution of other transactions might compensate the temporary decrease in performance.

5 Open Issues

Most benchmarks are modeled as an open or closed system, although, a partly-open system is more accurate for most real scenarios. In particular, the TPC-C is modeled as a closed system [18].

This has a direct impact on the results presented in this paper. Open and partly-open system have a worse degradation in performance due to contention when compared to closed systems: a higher mean response time and reduced throughput. The variability of service demand also has a huge impact on the mean response time. This is particular important when taking into account the *Delivery* transaction which takes around 35 ms to execute, in contrast to others that take no more than 10 ms.

Any additional contention introduced by a replication protocol is troublesome for the overall system performance and should be avoided or circumvented whenever possible. Disregarding this key factor leads to the intensification of weakness in the protocols (e.g. queuing and abort rate) and most likely makes them infeasi-

ble for most real application scenarios. For those reasons, it is extremely important to evaluate the protocols presented here, in particular AKARA, with a partly-open benchmark in order to figure out whether it would behave as expected or not.

Although the current implementation of the AKARA statically specifies the multiprogramming limit (MPL) by establishing the number of optimistic transactions that can be concurrently executed on a replica, this information could be dynamically defined as in [17]. One might use an adaptive mechanism [10] to determine this value taking into account the idleness of the database and the abort rate due to the optimistic execution.

In [11], it is proposed an adaptive mechanism to control the MPL. However, in this case, it basically avoids that latency of the conservative protocol increases drastically by reducing or increasing the number of connections or balancing load among replicas. There is no attempt to reduce the time spent in queues.

Deciding whether a transaction should be passively or actively executed is a task that might be done automatically or manually. In the former case, AKARA might learn from previous executions of a transaction in order to come up with a decision. Usually, the higher the number of changes the more appropriate is the use of an active replication. Furthermore, AKARA might exploit the GORDA API [5] to extract information from a database such as the number of changes made by a transaction and whether there are DDL statements or not. The GORDA API might also be used to help in removing non-deterministic information in statements by withdrawing most of the work from the replication middleware.

Finally, it is worth noticing that having conflict classes based on tables easies the classification procedure regardless if it is done automatically or manually. In particular, if the classification is done manually, it is pretty simple to automatically detect labeling mistakes.

6 Conclusion

The performance of group-based database replication protocols can be challenged by demanding workloads. Namely, conservatively synchronized protocols overly restrict concurrency, and thus throughput, unless a careful application-specific definition of conflict classes is done. On the other hand, optimistically synchronized protocols make it difficult that long lived and prone to conflicts transactions can commit. Finally, both depend on shipping updated data items, which makes it hard to deal with very large updates or DDL statements. Although all these issues can easily be avoided in benchmarks, they are a significant hurdle to adoption in real scenarios.

In this paper we address these issues with the AKARA protocol, which seamlessly combines multiple execution strategies. Experimental evaluation with the TPC-C workload shows that the proposed protocol provides adequate throughput without requiring application-specific tuning of conflict classes. By introducing a small number of transactions with large write sets or DDL statements in the mix to

be actively replicated, one also shows that fairness is ensured and network usage minimized.

References

[1] R. Baldoni, S. Cimmino, C. Marchetti, and A. Termini. Performance Analisys of Java Group Toolkits: a Case Study. In *FIDJI*, 2002.

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

[3] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC:Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference*, 2004.

[4] Continuent. Sequoia 4.x. http://sequoia.continuent.org, 2008.

[5] A. Correia, J. Orlando, L. Rodrigues, N. Carvalho, R. Oliveira, and S. Guedes. Gorda: An Open Architecture for Database Replication. In *IEEE NCA*, 2007.

[6] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *ACM SIGMOD*, 1996.

[7] A. Correia Jr., A. Sousa, L. Soares, J. Pereira, F. Moura, and R. Oliveira. Group-based Replication of On-line Transaction Processing Servers. In *LADC*, 2005.

[8] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB Conference*, 2000.

[9] Y. Lin, B. Kemme, R. Jiménez Peris, and M. Patiño Martíne. Middleware based Data Replication providing Snapshot Isolation. In *ACM SIGMOD*, 2005.

[10] M. Matos, Jr. A. Correia, J. Pereira, and R. Oliveira. Serpentine: adaptive middleware for complex heterogeneous distributed systems. In *ACM SAC*, 2008.

[11] J. Milan-Franco, M. PatioMartnez R. Jimenez-Peri and, and B. Kemme. Adaptive middleware for data replication. In *USENIX International Conference on Middleware*, 2004.

[12] R. Oliveira, J. Pereira, A. Correia Jr, and E. Archibald. Revisiting 1-Copy Equivalence in Clustered Databases. In *ACM SAC*, 2006.

[13] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine Approach. In *Journal of Distributed and Parallel Databases and Technology*, 2003.

[14] R. Jiménez Peris, M. Patiño Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. *IEEE ICDCS*, 2002.

[15] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *USENIX International Conference on Middleware*, 2004.

[16] F. Schneider. Replication management using the state-machine approach. In *Distributed Systems*. ACM Press/Addison-Wesley Publishing Co., 1993.

[17] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wiernam. How to determine a good multi-programming level for external scheduling. In *IEEE ICDE*, 2006.

[18] B. Schroeder, A. Wierman, and Mor Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *NSDI*, 2006.

[19] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial Replication in the Database State Machine. In *IEEE NCA*, 2001.

[20] A. Sousa, J. Pereira, L. Soares, A. Correia Jr., L. Rocha, R. Oliveira, and F. Moura. Testing the Dependability and Performance of GCS-Based Database Replication Protocols. In *IEEE DSN*, 2005.

[21] Transaction Processing Performance Council (TPC). TPC benchmark C Standard Specification Revision 5.0, 2001.