Performance Evaluation of a Metaprotocol for Database Replication Adaptability

M. I. Ruiz-Fuertes and F. D. Muñoz-Escoí
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Camino de Vera, s/n. 46022 Valencia, Spain
Email: {miruifue,fmunyoz}@iti.upv.es

***Abstract*—Common solutions to database replication use a single replication protocol. This approach lacks flexibility for changing scenarios or when dealing with heterogeneous client application requirements. Our proposal is a metaprotocol that supports several replication protocols which may follow different replication techniques or provide different isolation levels. With our metaprotocol, replication protocols can either work concurrently with the same data or be sequenced for adapting to dynamic environments. Experimental results demonstrate its low overhead and measure the influence of protocol concurrency on system performance.**

I. INTRODUCTION

Many replication protocols have been designed and studied, proving that different protocols provide different features (consistency guarantees, scalability, etc.) and obtain different performance results depending on the environment characteristics (workloads, network latencies, access patterns, etc.). Protocols performance was compared in recent studies such as [1], which presented the ROWAA approach as the most suitable for the general case, and [2], based on total order broadcast. For a particular scenario with certain workload, access pattern and isolation requirements, a specific protocol can be chosen as the most suitable. Thus, common solutions analyze the general case of their environment and choose their protocol accordingly. But this initially chosen protocol remains in the system while the environment evolves, degrading its performance or even being incapable of meeting new client requirements. This way, when dealing with dynamic scenarios or when concurrent client applications have different requirements (e.g. different isolation levels), a more adaptable solution is needed.

A sign that this adaptability is demanded by real applications is represented by Microsoft SQL Server. It supports two concurrency controls simultaneously: one optimistic, based on *row versioning* [3]; and one pessimistic, based on locks. Thus, it concurrently supports different isolation levels –mainly, snapshot isolation and serializable, respectively.

However, no academic result had yet raised this support to middleware level for database replication systems by enabling several replication protocols to work concurrently. To meet this adaptability requirement, we have designed a metaprotocol that supports the concurrency of a set of consistency protocols. With our metaprotocol, each concurrent application can take advantage of the protocol that better suits its needs: e.g., an application with long transactions will prefer pessimistic replication to ensure their commitment (as the longer the transaction, the higher the probability of abortion in optimistic techniques due to conflicts with concurrent transactions). Additionally, each protocol can be replaced if the application access pattern is drastically modified or if the overall system performance changes due to specific load variations or network or infrastructure migrations. This way, adaptability is provided at two levels: a) *outward adaptability*, which makes the system able to deal with different concurrent requirements, and b) *forward adaptability*, which allows the system to adapt to dynamic characteristics in the environment or in the clients themselves.

The metaprotocol was first presented in [4] with its complete pseudocode. This paper continues that work, providing an experimental evaluation of the metaprotocol overhead and the potential performance penalty of every possible combination of protocols when working concurrently.

The rest of this paper is structured as follows. Section II summarizes the main aspects of the metaprotocol. Section III presents the experimental results. Section IV discusses some related work and, finally, Sect. V concludes the paper.

II. METAPROTOCOL

The metaprotocol function is to support multiple replication protocols concurrently at each replica, properly managing the dependencies between them. It also allows a working protocol to be exchanged when it is detected that another one would fit better (a common situation in dynamic environments). This protocol exchange is seamlessly performed: already started transactions end their execution using the protocol they started with, while new ones use the new protocol. Thus, processing does not need to be halted.

A. Supported Protocols

Currently, three replication protocols have been tested with our metaprotocol. As some common characteristics are needed in order to make concurrency feasible or, at least, practical, all protocols follow the same replica consistency

This work has been partially supported by FEDER and the Spanish MEC under grants TIN2006-14738-C02-01 and BES-2007-17362, and IMPIVA and EU FEDER under grant IMIDIC/2007/68.

model (sequential) [5]. Thus, each targeted protocol is a representative of three protocol families based on FIFO total order broadcast [2]: active, certification-based and weak voting replication. All these families are update-everywhere [6] (to send its request, a client chooses one server, which is known as the delegate server), so they are decentralized replication protocols. On the other hand, each replication protocol may provide different transaction isolation levels, which can be exploited by different client applications.

In *active* replication, the delegate broadcasts the client request to all replicas in total order. Later, server replicas, including the delegate, execute and commit the transaction in the order it was delivered. Due to the sequential execution of transactions, no abortion arises in this replication model, which can provide any isolation level supported by the local database management system (DBMS).

In *certification-based* replication, transactions are first locally executed in their delegate server and their writesets (set of written objects) and, in some isolation levels, also their readsets (set of read objects), are broadcast to all replicas. After delivery, a deterministic certification phase, based on conflicts with concurrent transactions, starts in all replicas to determine if such transaction can commit or not.

In *weak voting* replication, transactions are also locally executed and then their writesets are broadcast. But in this case, only the delegate (since readsets are never broadcast in this kind of protocols) is in the position to validate a transaction –again, based on conflicts with concurrent transactions–, broadcasting later its decision to all replicas.

The transaction metadata needed when all protocols are working is compound by the writeset and a timestamp of the transaction begin. With this information, certification-based and weak voting techniques can provide snapshot isolation. Additionally, if the underlying DBMS features a serializable concurrency control, weak voting techniques provide 1-copy-serializability at no extra cost. On the contrary, certification-based techniques need to broadcast readsets to provide the same isolation level. As readset management is costly, certification-based replication is normally used only for snapshot isolation. Thus, we do not consider readsets.

Each of the targeted protocols presents advantages and disadvantages. Active replication is pessimistic and forces all replicas to completely execute every transaction, which increases the system load but ensures that no transaction ever aborts. This is very useful for long transactions that otherwise will have a high probability of being aborted due to conflicts with concurrent transactions. The other two techniques are both optimistic. Weak voting techniques may provide 1-copy-serializability without the need of working with readsets, but they require an extra broadcast that forces non-delegate replicas to wait. Certification-based replication achieves fast certification of transactions, but it is not practical for isolation levels stronger than snapshot, due to readset management. The metaprotocol adaptability allows to switch to the most appropriate protocol at any moment to exploit all the advantages, while trying to overcome the disadvantages.

B. Metaprotocol Outline

Two shared lists are maintained by the metaprotocol in each replica: the *log* list, with the history of all the system transactions (can be purged as suggested in [2]); and the *tocommit* list, with the transactions pending to commit in the underlying database. These lists, initially empty, contain transactions from all the protocols working at the moment. Each transaction has an associated type, which represents its current status and, thus, is modified during the transaction lifetime. Possible types are the following: a) *resolved*, a committable (or already committed) transaction with writeset info available; b) *c-pending*, a transaction with writeset info available but not yet committable (e.g. a weak voting transaction waiting for its voting message); and c) *w-pending*, a transaction with no writeset info available (i.e. an active transaction not yet committed).

A brief outline of the major steps of the metaprotocol at one replica, R_k, is depicted in Fig. 1A. Roughly speaking, protocols send messages in step I, which are processed in step II depending on the message type. In the following steps (Fig. 1B), the metaprotocol delegates transaction management on the installed protocols, which have access to all shared variables. These include the *log* and *tocommit* lists and two integer counters: `L-TOI` and `N-TOI`. Based on their delivery order, transactions are assigned a unique `TOI`, or total order index. Variable `N-TOI`, initialized to 1, stores the index for the next transaction to be delivered. Variable `L-TOI`, initialized to 0, stores the index of the last committed transaction in the replica. These counters provide logical timestamps for transactions.

Active transactions (Fig. 1Ba) do not need a validation phase: all commit in the order established by the broadcast. Thus, when an active message arrives, the transaction is added to both lists and marked as w-pending. As its writeset will be only known after commitment, it prevents subsequent certifications from being completed, as writesets are needed to determine if transactions present write conflicts. Transaction dependencies will be explained in detail later.

Messages from the certification-based protocol (Fig. 1Bc) contain a writeset which has to be certified. This certification is based on two integers representing the transaction start and end, respectively: *bot*, begin of transaction, set before broadcasting to the current value of `L-TOI` –its validity as logical timestamp of transaction start is ensured by a conflict detection mechanism [7]–, and *toi*, the total order index set at reception. With a negative result, the transaction is aborted. Otherwise, it is added to both lists. Certification can obtain a *pending* result if there are concurrent w-pending transactions or conflicting transactions whose certification/validation phase is incomplete, i.e. c-pending transactions. This creates a dependency between the trans-

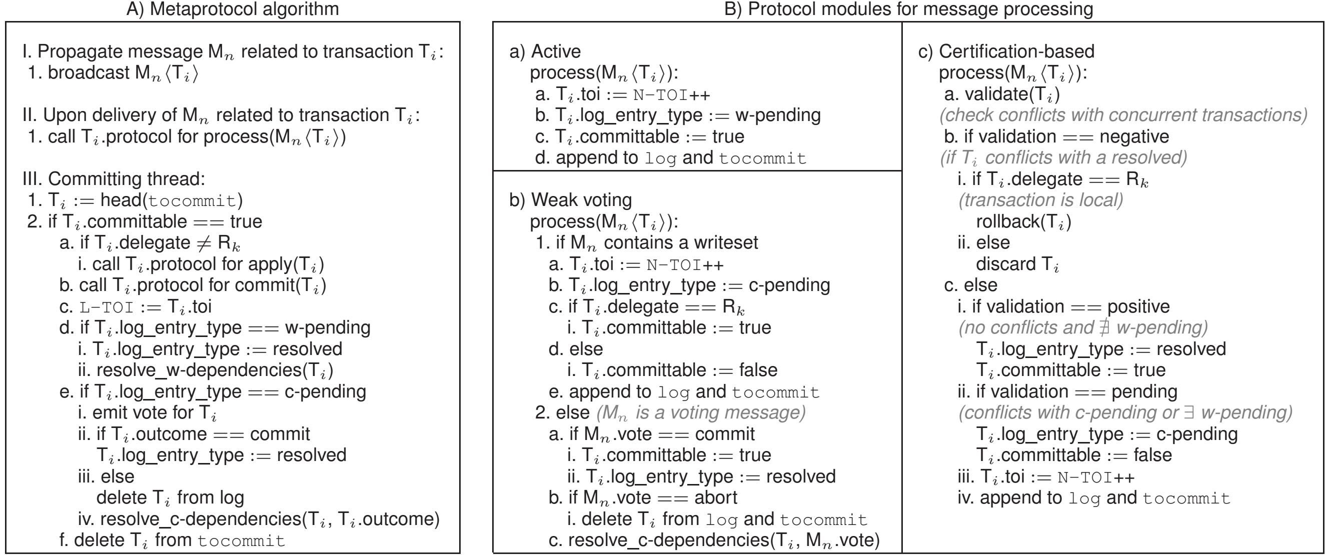

Figure 1. Metaprotocol and protocol modules at replica R_k

action being certified and each of the previously delivered transactions that create the indecision.

When the delivered message contains a writeset from the weak voting protocol (Fig. 1Bb), the transaction is added to both lists and marked as c-pending as its outcome is unknown until commit time in the delegate node or the arrival of the voting message in the rest of nodes. In the pseudocode, the validation is based on the local concurrency control: waiting to commit turn and trying to commit the transaction in the delegate. If the commitment succeeds, a positive vote is broadcast (reliably but without total order) to all replicas. Otherwise, a negative vote is sent.

The reception of a voting message for a weak voting transaction changes the transaction status and resolves the dependencies between this transaction and subsequent ones.

The third major step of the pseudocode consists in committing the first transaction in the *tocommit* list, provided that it is committable. This is performed sequentially by the metaprotocol, one transaction at a time, following the list order (provided by the total order broadcast). When applying remote writesets, conflicts with local transactions may arise. At this point, our conflict detection mechanism [7] aborts those local transactions allowing the correct remote writeset application. After commitment, active transactions obtain their writeset and possible dependencies are resolved. In the case of weak voting transactions, the outcome is used to resolve dependencies and to emit the vote.

As seen in the pseudocode, the common processing is carried out by the metaprotocol (maintenance of data structures, sending and reception of messages, scheduling of transactions...), which calls the corresponding protocol when protocol-specific processes have to be done (treatment of messages, commitment of transactions...). On the other hand, communication with client applications remains in the protocols, thus preserving previous client-protocol interfaces. This way, the system presents high modularity and protocols remain very simple and easy to maintain (required adaptations to work within the metaprotocol only consist in simplifications), while they are still able to introduce some optimizations in their specific methods (e.g. a pre-certification process prior to broadcast, which may save useless network communication and subsequent processing).

C. Dependencies Between Protocols

Concurrency can lead to inefficient systems due to natural differences in the behavior of the protocols. Several dependencies may arise when certifying transactions in certification-based protocols, or when validating weak voting transactions in the delegate node, if such validation is performed in a phase similar to the certification.

These dependencies, a natural and inevitable consequence of the concurrency between different replication techniques, introduce additional waiting times. Thus, it is important to understand them and to carefully study their implications, as they may have a notable impact on system performance.

A transaction, in order to be successfully certified or validated, must know the writesets of all concurrent and previously delivered transactions that will eventually commit. However, this may not be immediately known, as there is some pending information in the entries of the log list. First, the writesets of w-pending transactions are unknown until their commit time. Second, the final termination of c-pending transactions is not yet known (e.g. weak voting transactions waiting to their vote). This pending information prevents the certification/validation of a transaction T_k from finishing in two ways: a) T_k cannot check for conflicts with

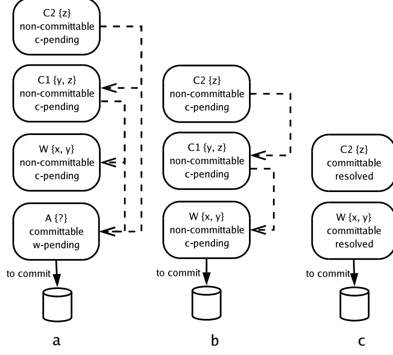

Figure 2. Dependencies between transactions

a w-pending transaction T_w, as T_w's writeset is not yet available (here we say that T_k has a w-dependency with T_w); and b) although a c-pending transaction T_c's writeset is known and thus T_k can check for conflicts, the final outcome for T_c is yet unknown due to a pending vote or another dependency (here we say that T_k has a c-dependency with T_c). Notice that a conflict should only cause the abortion of T_k if conflicting transaction T_c's final outcome is a commit.

Note also that a transaction T_i may present several dependencies, i.e. depend on several previous transactions, and it will not be considered as resolved until all its dependencies are resolved. At that moment, the dependencies caused by T_i on following transactions will be resolved in cascade.

Let us consider an example situation to review the steps of the metaprotocol. Suppose that all three protocols are executing, so transactions from all of them are delivered at replica R_k. Suppose also that, at a given moment, the *tocommit* list is empty. At this moment, an active transaction A is delivered. A is directly appended to the lists, marked as committable and w-pending. The commitment of A begins. Then, a weak voting transaction W, writing objects x and y, is delivered. Suppose R_k is not its delegate replica. Thus, W is added to the lists, marked as c-pending and non-committable until its vote arrives. A new transaction is delivered: C1, a certification-based transaction that writes objects y and z. C1 obtains a *pending* result in its certification, as there is a concurrent w-pending transaction (A) and C1 presents write conflicts with W, which is c-pending (thus, C1 has two dependencies). This pending certification forces C1 to be marked as c-pending and, thus, non-committable until both dependencies are resolved. Later, another certification-based transaction C2 is delivered. C2, which writes object z, is also marked as c-pending and non-committable because of the conflict with C1 and the existence of A. The current stage corresponds to Fig. 2a (where dashed arrows represent dependencies). At this moment, A finally ends its commit operation and its writeset is collected: it wrote objects u and v. Now it is time to resolve the w-dependencies of C1 and C2. As A does not conflict with them, both w-dependencies are just removed. Figure 2b represents the current situation. Now, the voting message for W is delivered with a commit vote. So W is now committable and some c-dependencies can be resolved. As W presented conflicts with C1 and W is going to commit, C1 must abort. Due to the termination of C1, more c-dependencies are resolved on cascade, thus removing all the dependencies presented by C2, that becomes committable. This stage is depicted in Fig. 2c. Any committable transaction at the head position of the *tocommit* list is eventually committed.

III. Experimental Results

Naturally, there is a trade-off between the high level of adaptability provided by the metaprotocol and system performance. Tests were conducted to measure two aspects. a) The overhead introduced by the metaprotocol management. To this end, stand-alone versions of the active, weak voting and certification-based replication protocols were implemented. Later, we run each protocol in both the stand-alone manner and as the only available protocol within the metaprotocol. Differences in performance will give a measure of the metaprotocol overhead. b) The penalty in performance due to protocol concurrency. All possible combinations of protocols were tested in concurrency within the metaprotocol. Measures were taken separately for each protocol, thus showing, e.g., the variations on completion time for certification-based transactions when executed in a pure certification-based system or when another protocol is also working and some dependencies arise.

System Model: We assume a partially synchronous distributed system where each node holds a replica of the database. For local transaction management each node has a local DBMS on top of which a database replication middleware system is deployed. This middleware uses a group communication service (abbr., GCS) that provides a total order multicast. Failures are not considered in this paper, but will be analyzed and managed in future work.

Test Description: To accomplish the analysis, we use Spread as GCS and PostgreSQL as underlying DBMS providing snapshot isolation by means of a multiversion concurrency control. Transactions access a database with a single table (the smaller the database, the greater the probability of conflicts and, thus, of dependencies) of 10,000 rows and two columns. The first column is the primary key; the second, an integer field subject to updates made by transactions.

Both the metaprotocol and the stand-alone protocols were tested in our replication middleware MADIS [8] with 2 and 4 nodes. Each node has an AMD Athlon(tm) 64 Processor at 2.0 GHz with 2 GB of RAM running Linux Fedora Core 5 with PostgreSQL 8.1.4 and Sun Java 1.5.0, and interconnected by a 1 Gbit/s Ethernet. Transactions are initiated at a fixed pace in each replica in order to obtain system input rates of 20, 40, 60 and 80 TPS (transactions per second). Each transaction writes 20 rows (a fixed number of items,

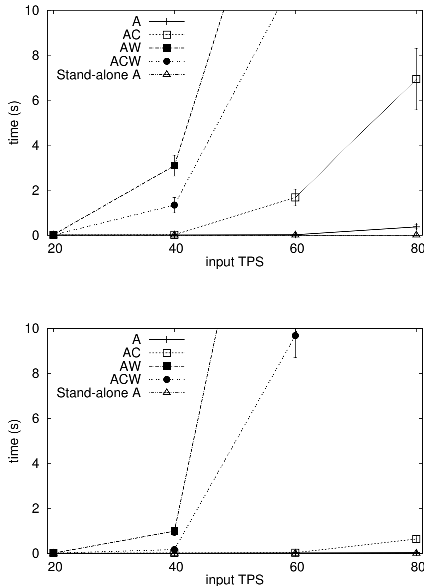

Figure 3. Active replication. Transaction length with (a) 2, (b) 4 replicas.

as the protocol tasks do not depend on transaction length). This workload was designed to meet the purpose of stressing the system gradually with higher and higher input rates of write-only transactions causing more and more conflicts and, therefore, dependencies. We discard read operations, which are only locally executed and have no conflicts. As a result, we subject the system to a worst-case environment.

Scenarios: Active (A), certification-based (C) and weak voting (W) techniques were tested in a stand-alone manner and within the metaprotocol. In the latter case, combinations of 1, 2 and 3 protocols were used (a total of 5 scenarios for each technique, e.g. for active replication: Stand-alone A, A, AC, AW and ACW). Each protocol managed a proportional part of the issued transactions, which were analyzed separately to compute length and abortion rate. The plotted results are means with their 95% confidence interval.

Results: Protocols do not show important differences in response time when executed in a stand-alone manner or as the only protocol within the metaprotocol. Indeed, performance is virtually the same at low input rates or when using a system of 4 replicas. Only when 2 replicas must support a high input rate, differences appear. Thus, the main factor to consider is the penalty due to concurrency between different techniques. This concurrency is appropriate when multiple client applications access the same database in different ways. The active technique is pessimistic while the other two are optimistic. This different approach limits the performance: optimistic techniques are forced to wait to the processing of pessimistic transactions, thus reducing the advantages of their optimism. Indeed, as soon as there is one active transaction in the *tocommit* queue, all non-active subsequent transactions in the queue establish a dependency with it. This dependency lasts until the active transaction is committed. Moreover, weak voting replication is handicapped by the second broadcast needed to emit the vote: non-delegate replicas must wait for the delegate to validate the transaction and for the vote to arrive. All these drawbacks join when mixing active and weak voting replication, leading to poorer performance in AW and ACW scenarios.

Figure 3 corresponds to the active technique (recall that it never aborts transactions). Response times for active transactions increase when adding the optimistic certification-based technique (AC). A more important degradation occurs when combining active and weak voting techniques, specially with heavy workloads (i.e., with 60 and 80 TPS).

Results from certification-based techniques are presented in Fig. 4. Again, ACW is the worst scenario, where longer transaction times also raise the abortion rate. A degradation is also observable in the CW scenario due to the handicap suffered by weak voting transactions. Other scenarios do not involve important difficulties for certification-based transactions. When 4 replicas are used, some behavioral trends observed in a 2-replica system are maximized. Thus, configurations that performed well for 2 replicas have better performance, as each node is less loaded.

The weak voting technique is an excellent option when readsets must be considered for validation, as it avoids the collection and transmission of readsets. Unfortunately, as already explained, system perfomance is reduced (Fig. 5).

Figure 6 show the output TPS, i.e. the amount of committed transactions per second in the whole system. It is clearly seen how some combinations have an excellent performance while others degrade as load or system size increase.

Final remarks: The trade-off between adaptability and performance must be carefully analyzed in each system. Nevertheless, the CW combination has shown an excellent performance. This can be combined with the flexibility already offered at DBMS level by Microsoft SQL Server, which concurrently supports snapshot and serializable isolation levels. Thus, when the middleware is deployed on top of such DBMS, our metaprotocol can provide a straightforward support for both isolation levels in replicated environments. To our knowledge, no other solution offers at a middleware layer such degree of transparency and functionality.

IV. Related Work

No academic result, apart from our previous work in [4], presents a valid solution for a database replication system capable of supporting concurrent replication protocols.

A different approach was developed in [9], where a single replication protocol supports multiple isolation levels. Unfortunately, the resulting protocol is complex and it lacks the modularity and maintainability of our metaprotocol.

The idea of a single replication protocol providing a flexible behavior was also studied in [10], with the AKARA

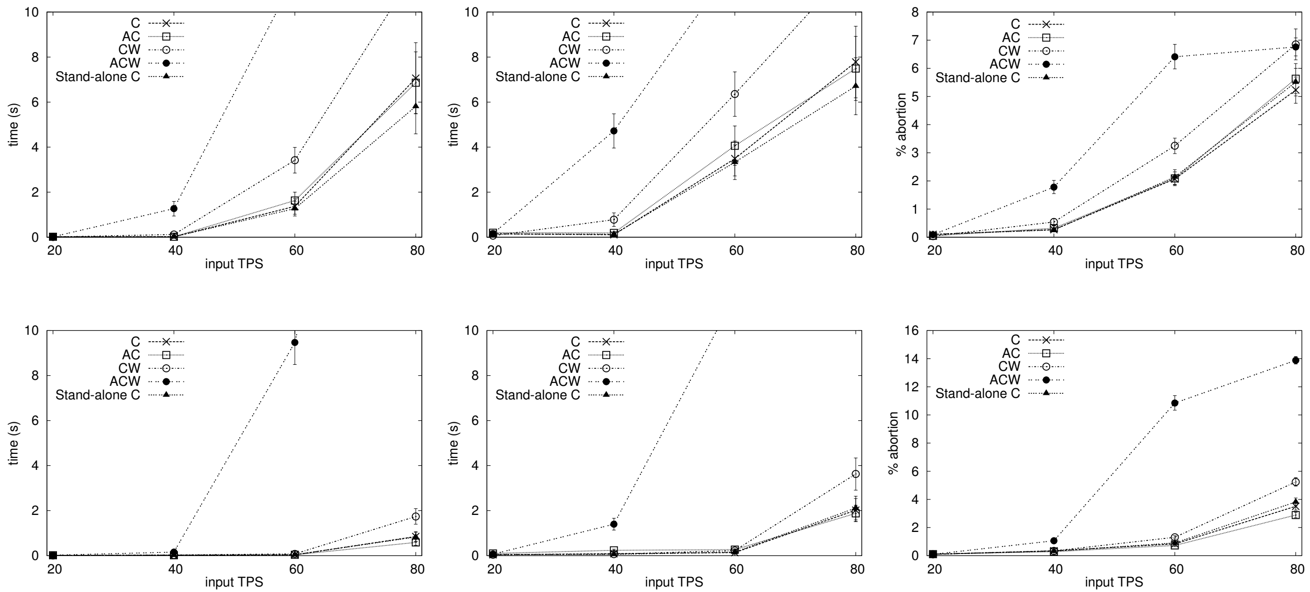

Figure 4. Certification-based replication. With 2 replica nodes: (a) length of committed transactions, (b) length of aborted transactions, (c) abortion rate. With 4 replica nodes: (d) length of committed transactions, (e) length of aborted transactions, (f) abortion rate.

protocol. With it, a transaction can be executed in an active or a passive manner, thus taking advantage of the best characteristics of each replication protocol. Unfortunately, AKARA needs the database to be partitioned in conflict classes and classifies each transaction regarding the accessed classes. This allows a straightforward conflict detection but forces to know the entire transaction before its execution, thus precluding interactive transactions. Moreover, conflict classes are usually entire tables, which leads to a coarse-grained conflict detection. On the other hand, our metaprotocol (using the mechanisms described in [7]) provides a row-level conflict detection, which allows the execution of interactive transactions, as no partition in the database is needed. Finally, although AKARA performs better, it has to be noted that it uses the TPC-C benchmark where a new request is only triggered by the completion of the previous one, and read-only transactions are included in the load.

The problem of protocol exchange, or dynamic protocol update (abbr., DPU), has been broadly discussed [11]–[13]. However, DPU solutions provide adaptability by replacing the working protocol and do not consider protocol concurrency (except, perhaps, for a short transition phase).

V. Conclusion

Adaptability is a desirable feature for all systems, especially for those more sensitive to changes in the environment. Moreover, client applications of database replication systems may demand different requirements that can be better served with different replication techniques.

We study the performance of a metaprotocol that supports the concurrent execution of several replication protocols based on atomic broadcast: active, certification-based and weak voting replication. Experimental results demonstrate that our metaprotocol introduces very low overhead when compared with stand-alone versions of the same replication protocols. On the other hand, inherent differences in the protocol behaviors may penalize concurrency. We show and explain that certain combinations of protocols should be avoided if performance is a major system goal. Other protocol combinations, however, showed excellent performance.

As future work, a load monitor will be developed to automatically decide the best protocol for each transaction, depending on relevant environmental characteristics.

References

[1] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso, "How to Select a Replication Protocol According to Scalability, Availability, and Communication Overhead." in *SRDS*. IEEE-CS, 2001, pp. 24–33.

[2] M. Wiesmann and A. Schiper, "Comparison of Database Replication Techniques Based on Total Order Broadcast," *IEEE TKDE*, vol. 17, no. 4, pp. 551–566, 2005.

[3] K. L. Tripp and N. Graves, "SQL Server 2005 Row Versioning-Based Transaction Isolation," Microsoft, Tech. Rep., 2006.

[4] M. I. Ruiz-Fuertes, R. de Juan-Marín, J. Pla-Civera, F. Castro-Company, and F. D. Muñoz-Escoí, "A Metaprotocol Outline for Database Replication Adaptability," in *OTM Workshops (2)*, ser. LNCS, vol. 4806. Springer, 2007, pp. 1052–1061.

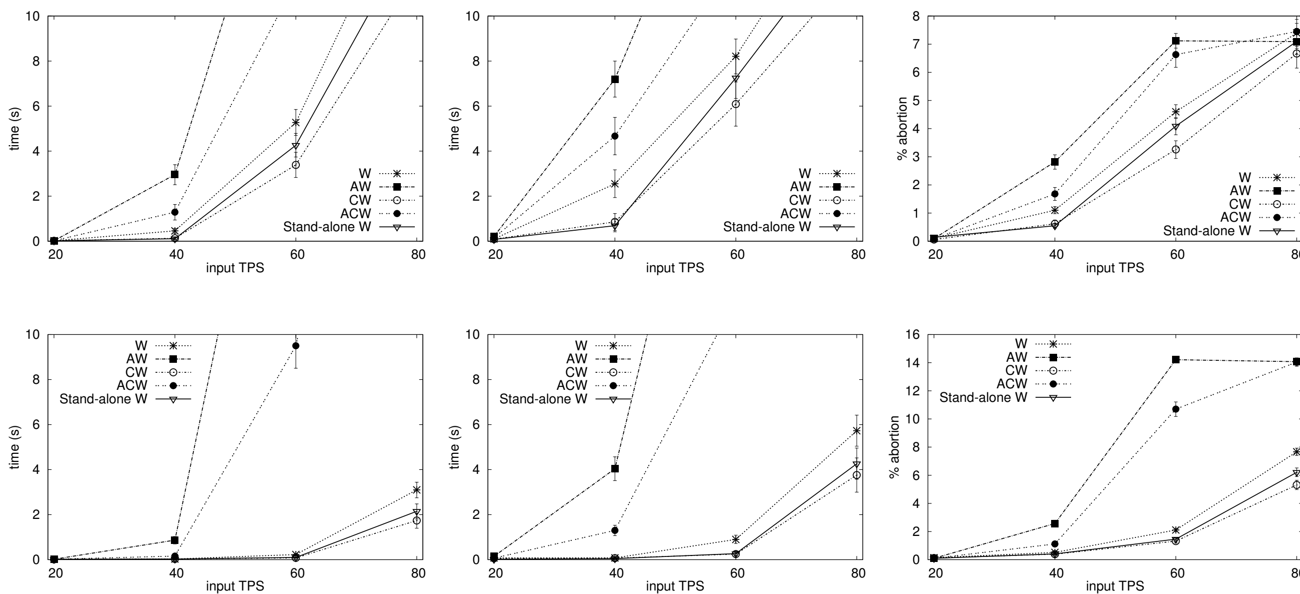

Figure 5. Weak voting replication. With 2 replica nodes: (a) length of committed transactions, (b) length of aborted transactions, (c) abortion rate. With 4 replica nodes: (d) length of committed transactions, (e) length of aborted transactions, (f) abortion rate.

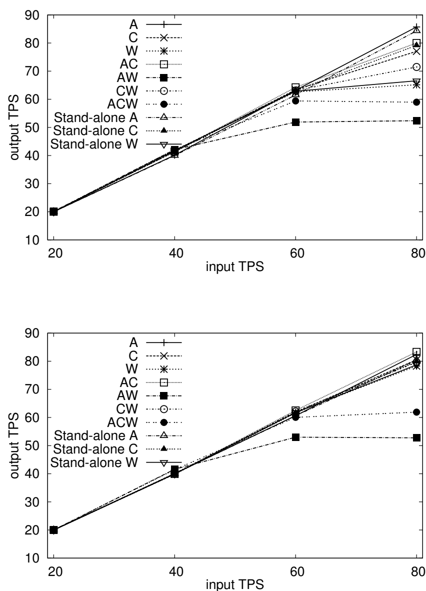

Figure 6. Output TPS with (a) 2, (b) 4 replicas.

[5] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, 1979.

[6] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso, "Database Replication Techniques: A Three Parameter Classification." in *SRDS*, 2000, pp. 206–215.

[7] F. D. Muñoz-Escoí, J. Pla-Civera, M. I. Ruiz-Fuertes, L. Irún-Briz, H. Decker, J. E. Armendáriz-Iñigo, and J. R. González de Mendívil, "Managing Transaction Conflicts in Middleware-Based Database Replication Architectures," in *SRDS*. IEEE-CS, 2006, pp. 401–410.

[8] L. Irún-Briz, H. Decker, R. de Juan-Marín, F. Castro-Company, J. E. Armendáriz, and F. D. Muñoz-Escoí, "MADIS: A Slim Middleware for Database Replication," in *Euro-Par*, ser. LNCS, vol. 3648. Springer, 2005, pp. 349–359.

[9] R. Salinas, J. M. Bernabé-Gisbert, and F. D. Muñoz-Escoí, "SIRC, a Multiple Isolation Level Protocol for Middleware-based Data Replication," in *ISCIS*. IEEE-CS Press, 2007, pp. 1–6.

[10] A. Correia, J. Pereira, and R. Oliveira, "AKARA: A Flexible Clustering Protocol for Demanding Transactional Workloads," in *OTM Conferences (1)*, ser. LNCS, vol. 5331. Springer, 2008, pp. 691–708.

[11] B. Bhargava, A. Helal, K. Friesen, and J. Riedl, "Adaptability Experiments in the RAID Distributed Database System," in *SRDS*, 1990, pp. 76–85.

[12] U. Fritzke Jr., R. P. Valentim, and L. A. F. Gomes, "Adaptive Replication Control Based on Consensus," in *SDDDM: Workshop on Dependable Distributed Data Management*. ACM, 2008, pp. 1–10.

[13] O. Rütti, P. T. Wojciechowski, and A. Schiper, "Structural and Algorithmic Issues of Dynamic Protocol Update," in *IEEE Intnl. Parallel and Distributed Processing Symposium*, 2006.