Evaluation of a Metaprotocol for Database Replication Adaptability

M. I. Ruiz-Fuertes, F. D. Muñoz-Escoí

Instituto Tecnológico de Informática, Valencia, Spain

{miruifue, fmunyoz}@iti.upv.es

Technical Report TR-ITI-ITE-08/18

Evaluation of a Metaprotocol for Database Replication Adaptability

M. I. Ruiz-Fuertes, F. D. Muñoz-Escoí

Instituto Tecnológico de Informática, Valencia, Spain

Technical Report TR-ITI-ITE-08/18

e-mail: {miruifue, fmunyoz}@iti.upv.es

Nov, 2008

Abstract

Common solutions to database replication use a single replication protocol providing certain performance and isolation characteristics. The main drawback of this approach is its lack of flexibility for changing scenarios –i.e. workloads, network latencies, access patterns...– 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 changing environments. Experimental results are provided that demonstrate the low overhead introduced by the metaprotocol when it is compared with stand-alone versions of the same replication protocols. Its low cost makes this metaprotocol a perfect choice when adaptability is needed. In this line, the use of a load monitor would enable the selection of the most appropriate protocol for each transaction, according to the current environment characteristics. Finally, we also studied the influence that protocol concurrency may have on system performance.

1 Introduction

Many replication protocols have been designed and studied, proving that different protocols provide different features –such as consistency guarantees, scalability. . . – and obtain different performance results depending on the environment characteristics. Recent studies such as [7] and [20] compare protocols performance. [7] presents the ROWAA approach as the most suitable for the general case and [20] is based on total order broadcast techniques. Thus, for a specific scenario with certain workload, access pattern and isolation requirements, a specific protocol can be chosen as the most suitable. Following this idea, 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 need different isolation requirements, a more adaptable solution is demanded. It is worth noting that we are not specifically addressing mobile systems when talking about dynamic environments, but any replicated system that experiments changes in its main characteristics (e.g. networks with long peaks of load due to massive client requests at some fixed times –like activities related to signing in and out at start and end of the working day–, or periodical administrative procedures that change the standard access patterns).

A sign that this adaptability is demanded by real applications is represented by Microsoft SQL Server, which, since version 2005, supports two concurrency controls simultaneously: an optimistic one, based on *row versioning* [18]; and a pessimistic one, based on locks. This way, it can concurrently support different isolation levels –mainly, snapshot isolation and serializable, respectively–.

However, no academic result had yet achieved to raise 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 is able to take advantage of the protocol that better suits its needs. Not only the best protocol for the general case can be selected for an application but also it can be replaced for another one 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: adaptability to different concurrent requirements and adaptability to dynamically changing environmental characteristics.

This paper continues the work started in [14], where the complete pseudocode of the metaprotocol was first presented. While that first work was focused on presenting the metaprotocol, this one is aimed at measuring with experimental tests the overhead introduced and the differences in performance when comparing the metaprotocol with stand-alone versions of the protocols supported by it. As we will demonstrate later, our metaprotocol introduces a very low overhead, thus providing a good performance, comparable with that of the stand-alone versions. Another important aspect to measure is the potential performance penalty of every possible combination of protocols when working concurrently. Our tests prove that some combinations have an excellent performance while others clearly degrade as load increases. Moreover, our tests were repeated for 2 and 4 replicas, in order to see the effect that the number of system nodes has on performance. Once again, some protocol combinations presented a good scalability while others greatly increased their response time. This can be easily explained considering the behavior of the targeted protocols, as we will see in this work.

The rest of this paper is structured as follows. Section 2 summarizes the main functional aspects of the metaprotocol. Section 3 presents and explains the experimental results obtained. Later, Section 4 discusses some related work and, finally, Section 5 presents the paper conclusions.

2 Metaprotocol

The metaprotocol function is to provide the proper support to allow one or more replication protocols to being able to execute concurrently in one replica. This way, when client applications requirements can be fulfilled with just one replication protocol, the metaprotocol works with only one protocol, but allows to change the current working protocol when it is detected that another one would fit better –possibly common in changing environments–. Moreover, this protocol exchange is performed without stopping processing, working both protocols concurrently during the meantime: already started transactions end their execution using the protocol they started with, while new ones use the new protocol.

On the other hand, if client applications have different requirements that are best met by different replication protocols, the metaprotocol supports their concurrent execution, properly managing the dependencies that arise between them. However, it is important to note that concurrency can lead to inefficient systems due to the natural differences in the behavior of the protocols.

2.1 Supported Protocols

Currently, three replication protocols have been tested with our metaprotocol. As some common characteristics are needed in order to make concurrency feasible, each of them is a representative of three protocol families based on total order broadcast [20]: active, certification-based and weak voting replication. All these families are update-everywhere [21], so they are decentralized replication protocols.

In *active* replication, the delegate server forwards the client request to all replicas using a total order broadcast. Later, server replicas –including the delegate one– execute and commit the transaction in the order it has been delivered.

In *certification-based* replication, transactions are first locally executed in their delegate server and then broadcast to all server replicas. After delivery, a deterministic certification phase starts in all replicas to determine if such transaction can be committed or not.

In *weak voting* replication, transactions are also locally executed and then broadcast. But in this case, only the delegate is in the position to validate a transaction, broadcasting later its decision to all replicas.

TYPE	DESCRIPTION
A	transaction of an active protocol
C	writeset of a transaction executed in a certification-based protocol
WV-1	weak voting transaction writeset
WV-2	voting message of a weak voting protocol

Table 1: Message types

TYPE	DESCRIPTION
resolved	a committable (or already committed) transaction with writeset info available
c-pending	a transaction with writeset info available but not yet committable (e.g. a weak voting transaction waiting for its voting message)
w-pending	a transaction with no writeset info available (e.g. an active transaction not yet committed)

Table 2: Log entry types

The transaction metadata needed when all protocols are working is compound by the writeset –the set of written objects– and a timestamp representing the begin of transaction. Moreover, readsets are also needed if the certification-based technique is used for providing 1-copy-serializability. Therefore, due to the high cost associated to manage readsets, it would be preferable to use the certification-based technique only for providing snapshot isolation. This way, when talking about certification-based transactions, only the writesets will be considered from now on.

2.2 Correctness Arguments

All three targeted protocol families are based on a total order broadcast that determines the order on which transactions are committed and, if applicable, the result of the certification/validation phase. In [12], a replication scheme based on atomic –total order– broadcast is presented and proved correct. The correctness proof first proves that every two processes produce the same multiversion serialization graph and, then, it proves that every graph is acyclic. Using the Multiversion Graph theorem [1], the proof concludes that the proposed replication protocol guarantees one-copy serializability. This reasoning can be used to sketch an intuitive correctness proof for these three protocol families and even for the metaprotocol itself, as it follows the same basic schema of that used in the original protocols: (a) transactions are locally executed in one process without interaction among other processes; (b) when the transaction requests its commit, its writeset is propagated to the other processes, so that the transaction can be certified/validated and, if possible, committed. As each process receives the same input –committing transactions– in the same order –established by the atomic broadcast–, and each process follows a deterministic algorithm to commit or abort transactions, the resulting history of applied transactions will be the same in all processes, thus achieving one-copy equivalence.

2.3 Metaprotocol Outline

In order to support the previously presented protocol families, two shared lists are needed: the *log* list with the history of all the system transactions and the *tocommit* list, with the transactions pending to commit in the underlying database. These lists are maintained by the metaprotocol in each replica and contain transactions from all the protocols working at the moment.

A brief outline of the major steps of the metaprotocol is depicted in Figure 1 (for a complete pseudocode listing, see [14]). Tables 1 and 2 contain messages and log entry types, respectively. Roughly speaking, protocols send messages in step I, which are processed in step II depending on the message type:

- For an active message, the transaction is added to both lists and its entry in the log is marked as w-pending. This means that its writeset is unknown –it will be obtained after commitment–, which prevents subsequent certifications from being completed –writesets are needed to determine if transactions present write conflicts–. Active transactions do not need a validation phase: all are committed in the order established by the broadcast.

```
Initialization:
 1. tocommit := ∅; log := ∅
 2. L-TOI := 0; N-TOI := 1
 (TOI = Total Order Index
 N-TOI = TOI for the next transaction to be delivered.
 L-TOI = TOI of the last committed transaction.)

I. $P_q$ asks to send message $M_n$={t,c} where t is
   message type and c is message content, including
   related transaction $T_i$:
   (see message types in table 1)
 1. if t == C
      a. $T_i$.bot := L-TOI
 2. broadcast $M_n$

II. Upon delivery of $M_n$ related to transaction $T_i$:
 1. if $M_n$.type == A
      a. $T_i$.toi := N-TOI++
      b. $T_i$.log_entry_type := w-pending
      c. $T_i$.committable := true
      d. append to log and tocommit
 2. if $M_n$.type == C
      a. call $T_i$.protocol for validate($T_i$)
      (check conflicts with concurrent transactions)
      b. if validation == negative
      (if $T_i$ conflicts with a resolved)
        i. if $T_i$.replica == $R_k$
        (transaction is local)
           call $T_i$.protocol for rollback($T_i$)
        ii. else
           discard $T_i$
      c. else
        i. if validation == positive
        (if $T_i$ has no conflicts and $\nexists$ w-pending)
           $T_i$.log_entry_type := resolved
           $T_i$.committable := true
        ii. if validation == pending
        (if $T_i$ conflicts with a c-pending or $\exists$ w-pending)
           $T_i$.committable := false
        iii. $T_i$.toi := N-TOI++
        iv. append to log and tocommit
 3. if $M_n$.type == WV-1
      a. $T_i$.toi := N-TOI++
      b. $T_i$.log_entry_type := c-pending
      c. if $T_i$.replica == $R_k$
        i. $T_i$.committable := true
      d. else
        i. $T_i$.committable := false
      e. append to log and tocommit
 4. if $M_n$.type == WV-2
      a. if $M_n$.vote == commit
        i. $T_i$.committable := true
        ii. $T_i$.log_entry_type := resolved
      b. if $M_n$.vote == abort
        i. delete $T_i$ from log and tocommit
      c. resolve_c-dependencies($T_i$, $M_n$.vote)

III. Committing thread:
 1. $T_i$ := head(tocommit)
 2. if $T_i$.committable == true
      a. if $T_i$.replica $\neq$ $R_k$
        i. call $T_i$.protocol for apply($T_i$)
      b. call $T_i$.protocol for commit($T_i$)
      c. L-TOI := $T_i$.toi
      d. if $T_i$.log_entry_type == w-pending
        i. $T_i$-log_entry_type := resolved
        ii. resolve_w-dependencies($T_i$)
      e. if $T_i$.log_entry_type == c-pending
        i. emit vote for $T_i$
        ii. if $T_i$.outcome == commit
           $T_i$.log_entry_type := resolved
        iii. else
           delete $T_i$ from log
        iv. resolve_c-dependencies($T_i$, $T_i$.outcome)
      f. delete $T_i$ from tocommit
```

Figure 1: Metaprotocol algorithm at replica R_k

- For messages containing a writeset from the certification-based protocol, a certification is required. This certification is based on two integers representing the transaction start and end, respectively: *bot*, set before broadcasting –its validity as logical timestamp of transaction start is ensured by a conflict detection mechanism [11]–, and *toi*, set at reception. With a negative result, the transaction is aborted. Otherwise, it is added to both lists. As said before, certification can obtain a *pending* result if there exist concurrent w-pending transactions or conflicting transactions whose certification/validation phase is incomplete –c-pending transactions–. This situation creates what we call a dependency between the transaction being certified and each of the previously delivered transactions that create the indecision.

- For a message containing a writeset from the weak voting protocol, 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, we assume that the validation is performed in the simplest way: waiting to commit turn and trying to commit the transaction in the delegate. If the commitment succeeds, no conflicting transaction has been committed before and a positive vote is broadcast (reliably but without total order) to all replicas. Otherwise, a negative vote is sent. Possible optimizations include the delegate validating transactions in a phase similar to the certification of the certification-based replication. In fact, the current implementation uses this optimization.

- A voting message for a weak voting transaction changes the status of the transaction and, if needed, starts a process to resolve the dependencies established between this transaction and subsequent ones.

The third major step consists in committing the first transaction on the *tocommit* list, provided that it is marked as committable. This step is performed sequentially, one transaction at a time, following the list order, which, in turn, is provided by the total order broadcast. When applying remote transactions, conflicts with local transactions may arise. At this point, our conflict detection mechanism [11] eliminates those local conflicting 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 major processing is carried out by the metaprotocol –control of data structures, reception and treatment of messages, scheduling of transactions…–, which calls the corresponding protocol when protocol-specific processes have to be done –the commitment of a transaction or the certification phase in certification-based protocols–. On the other hand, communication with client applications remains in the protocols, thus preserving previous client-protocol interfaces. This way, the system presents a high modularity and protocols remain very simple and easy to maintain, while they are still able to introduce some optimizations in their specific methods –e.g. a pre-certification process prior to transaction broadcast, which may save useless network communications and subsequent processing–.

2.4 Dependencies Between Protocols

As already pointed out, several dependencies may arise during certification of transactions managed by certification-based protocols, or during the validation of a weak voting transaction on the delegate node if such validation is performed in a phase similar to the certification –an optimization previously mentioned–.

These dependencies are a natural and inevitable consequence of the concurrency between different replication techniques. But, as they introduce additional waiting times to the original protocols, they may have a notable impact on system performance. For this reason, it is important to understand them and to carefully study their implications, as we will do in Section 3.

A transaction, in order to be 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 (see Table 2). First, the writesets of w-pending transactions are unknown until their commit time. Second, c-pending transactions final termination is not yet known (e.g. weak voting transactions waiting to their vote or transactions waiting for their certification/validation phase to complete).

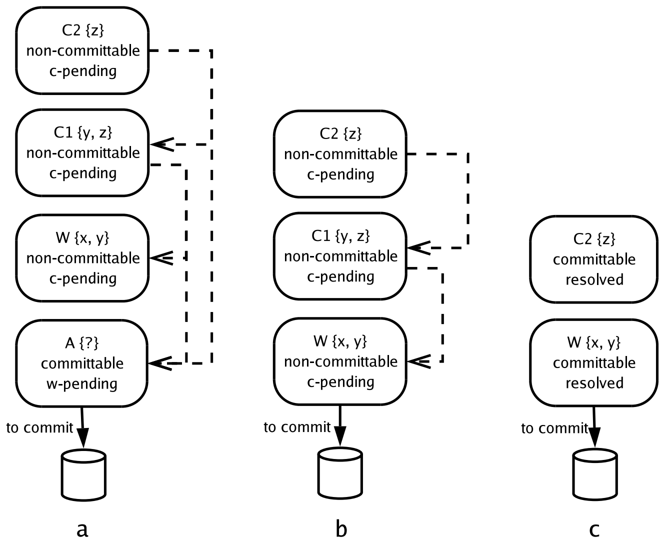


Figure 2: Dependencies between transactions

This pending information prevents the certification/validation of a transaction T_k from finishing in different ways: (a) T_k cannot check for conflicts with 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 be considered as resolved when all its dependencies are resolved. At that moment, the dependencies caused by T_i on following transactions will be resolved in cascade.

2.5 An Easy Example

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 object 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 Figure 2a. At this moment, A finally ends its commit operation and its writeset is collected: it wrote objects p and q. 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 its 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, its c-dependencies are resolved on cascade, thus removing all the dependencies presented by C2, that becomes committable. This stage is depicted in Figure 2c. Any committable transaction is eventually committed when it arrives to the head position of the *tocommit* list.

3 Experimental Results

Experimental tests were performed in our replication middleware MADIS [6, 11] in order to measure both the overhead introduced by the metaprotocol management and the performance penalty of the concurrency. To this end, stand-alone versions of the active, weak voting and certification-based replication protocols were implemented and compared against the metaprotocol when supporting only the corresponding protocol. On the other hand, all possible combinations of protocols were also tested in concurrency.

System Model. We assume a partially synchronous distributed system where each node holds a replica of a given database. For local transaction management, each node has a local DBMS that provides the requested isolation level. On top of the DBMS, a database replication middleware system is deployed. This middleware uses a group communication service (abbr., GCS), that provides a total order multicast.

Protocol implementations. Both weak voting implementations (the one used in the metaprotocol and the stand-alone version) are implemented with the previously commented optimization of starting a validation phase in the delegate node at delivery time, similar to the certification phase of certification-based protocols.

Versions to be used with the metaprotocol and stand-alone versions obviously differ in their implementations due to the different executing environment –e.g. stand-alone versions need to control the data structures that were controlled by the metaprotocol–. But except for the necessary adaptations for all of them to properly work, no additional differences were introduced to favor any of them.

Test Description. To accomplish the analysis, we use Spread [17] as our GCS and PostgreSQL [13] as the underlying DBMS. Transactions access a database with a single table, with 10,000 rows and two columns. The first column is the primary key. The second column is an integer field that is subject to updates made by transactions. A prior tuning process was performed on PostgreSQL, which showed that a reduced number of 10 database connections for client transactions in each replica was the best for our environment. This result relates with [9] which proves that an even more reduced number of database connections is the best option in some environments.

Both the metaprotocol and the stand-alone protocols have been tested using MADIS with 2 and 4 replica 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. They are interconnected by a 1 Gbit/s Ethernet. In each replica, transactions are initiated at a fixed pace in order to obtain system input rates of 20, 40, 60 and 80 TPS. Remember that the number of database connections was reduced for performance, so transactions are started at such pace but then they must wait to get a free connection. This waiting time is included in the transaction length. When a transaction obtains a connection, it accesses a fixed number of 20 rows for writing –these accesses may cause conflicts between transactions, which will be detected during the protocol validation–.

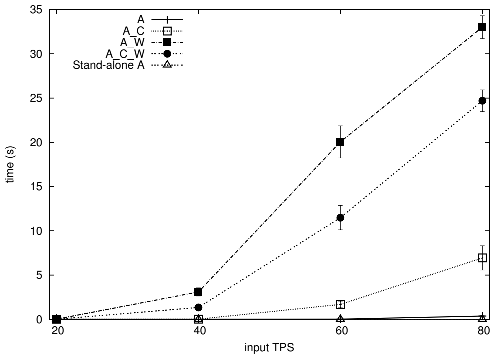
In order to study the influence of the number of replicas in the system, tests were performed twice: with 2 and 4 replicas. As the system load was the same for both scenarios –20, 40, 60 and 80 TPS–, nodes were supposed to be less loaded in the system with 4 replicas. On the other hand, a greater number of nodes may affect communication issues.

Results Three aspects were considered and measured in our tests: length of committed transactions, length of aborted transactions and abortion rate. These aspects were computed at each node in each test iteration. Each one of these iterations dismisses initial and final transient phases. A total of 2,000 local transactions were considered at each node –which gives a total of 4,000 values for each time measure in systems of 2 replicas, and a total of 8,000 values when 4 replicas are used–. Each result presented here is the mean value obtained after 20 of the previously detailed iterations, and it is shown with its 95% confidence interval. Small confidence intervals prove that the presented results are statistically representative.

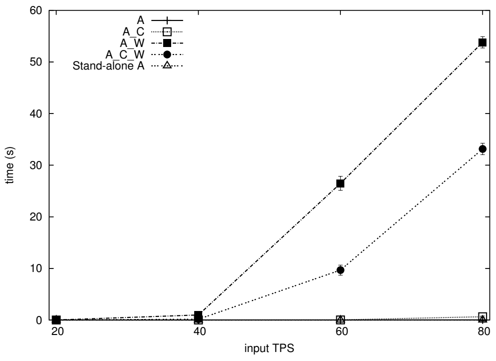
Results are presented separately for each protocol family –notice that active replication only presents results for transaction length, as no transaction in this replication technique is ever aborted–, and for each system size (2 or 4 replicas).

Each replication technique (e.g. active replication) was tested in 5 different scenarios: in a stand-alone manner (Standalone A), as the only protocol inside the metaprotocol (A), mixed with each of the other two techniques inside the metaprotocol (A_C and A_W), and concurrent with the other two inside the metaprotocol (A_C_W). When two protocols were combined, half of the total issued transactions were managed by each of them. Similarly, when all three protocols were concurrent, a third part of the amount of transactions was managed by each protocol. To allow easy comparison, we represent in the same graph the evolution of a certain measure in those 5 scenarios.

Moreover, we include graphs from Figure 8 which show the performance of the system measured in transactions committed per second.



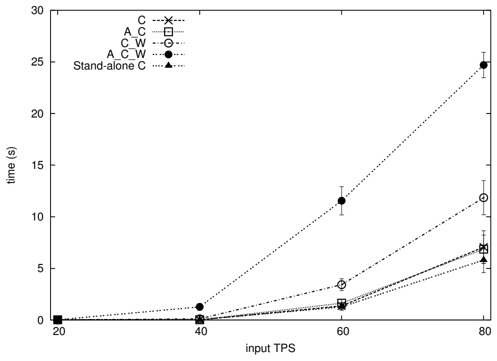
With 2 replica nodes



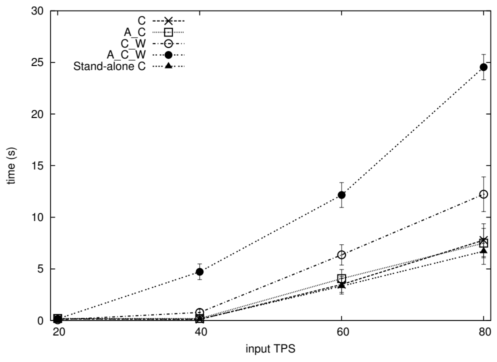
With 4 replica nodes

Figure 3: Active replication – Length of transactions

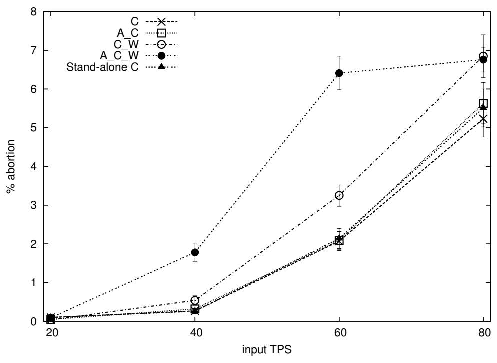
Active Replication Active replication protocols –Figure 3– do not show practical differences between stand-alone and A configurations in either tested system size. When combined with certification (A_C), the response time increases with the system load in the case of 2 replicas, while it remains near zero when 4 replicas are used, proving that this configuration scales correctly. On the other hand, configurations which mix active and weak voting replication not only do not improve their response time when increasing the number of replicas, but they even degrade their performance. This behavioral trend is observed in all presented graphs. The explanation is that active transactions hold dependencies until their commit time, thus preventing many subsequent transactions from being validated until all previous active transactions have committed. When these subsequent transactions are managed by weak voting replication (A_W), this wait may be bearable in delegate nodes, but remember that the rest of nodes must wait until the delivery of the voting message. Let us suppose this time to be a certain amount x of milliseconds. When 2 replicas are used, x milliseconds are wasted per transaction. But when increasing the number of replicas to 4, we are also increasing the number of non-delegate nodes, which then multiplies the wasted time up to 3x for each transaction. In the worst case, this means that all 3 non-delegate replicas have stopped all commitments waiting for the resolution of the transaction in the head of the *tocommit* list. As the global wasted time is greater, the global completed work is less, thus degrading performance. When adding certification to the configuration (A_C_W), resulting times are better, as certification-based transactions are not penalized by voting-waiting times and the dependencies they introduce last less than those of active transactions. Even though, this configuration also degrades when the number of replicas is increased.



Length of committed transactions



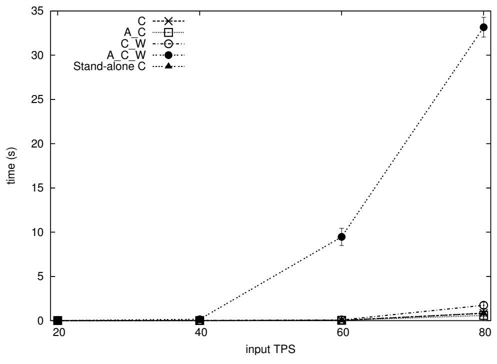
Length of aborted transactions



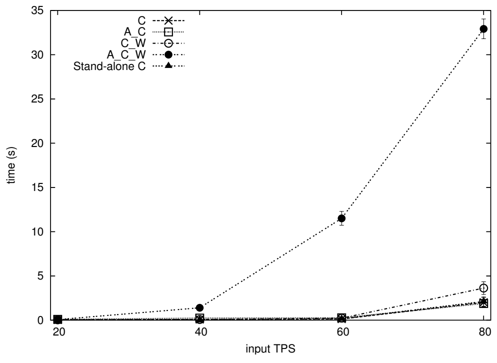
Abortion rate

Figure 4: Certification-based replication – 2 replica nodes

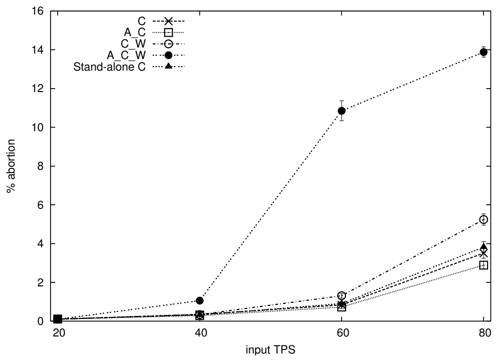
Certification-based Replication Graphs from Figure 4 show the behavior of certification-based protocols when 2 replicas are used. Times for committed and aborted transactions follow similar curves. Stand-alone and C configurations obtain very similar measures; small differences only appear for a 80 TPS load, when the stand-alone version is better. Times for configuration A_C are very close to the previous ones, thus confirming that dependencies caused by active transactions are only detrimental when weak voting transactions are also involved. When combining certification-based and weak voting techniques (C_W), dependencies suffered by weak voting transactions increase response times –but, as these dependencies do not last as much as those caused by active transactions, the performance of C_W is better than that of previous A_W–. Finally, we obtain the worse response time when mixing all three protocols. Regarding the abortion rate, the most interesting curve is that pertaining to the A_C_W configuration. As transactions last longer with this combination, more conflicts appear between them. Moreover, as the load grows, a higher level of concurrency is supported by the system, which involves a greater number of abortions. The last section of the curve presents a lesser slope because the system is already almost saturated at 60 TPS and concurrency slighty grows when introducing 80 TPS –see graphs of Figure 8–.



Length of committed transactions



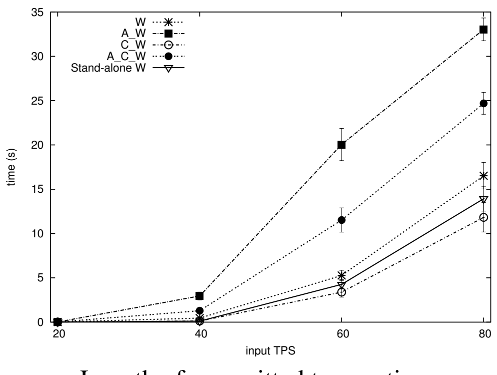
Length of aborted transactions



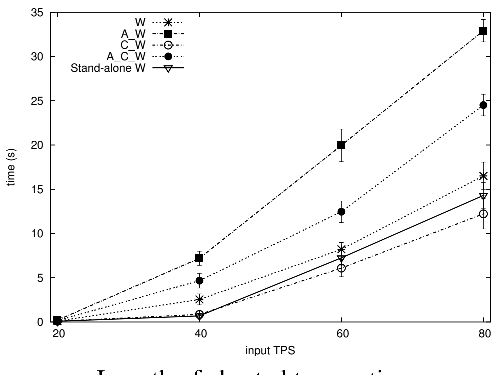
Abortion rate

Figure 5: Certification-based replication – 4 replica nodes

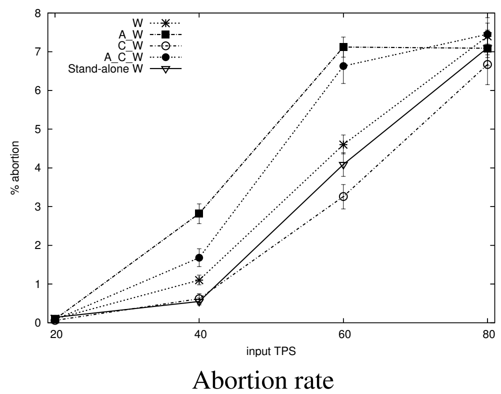
Figure 5 contains graphs corresponding to a system of 4 replicas. The trends observed when analyzing a system of 2 replicas are here maximized. Configurations that performed well for 2 replicas have here an even better performance, as each node is less loaded. On the other hand, bad combinations are worsened by the greater number of replicas. Remember that the global time wasted in the weak voting protocol augments with the number of nodes, thus reducing the overall performance.



Length of committed transactions



Length of aborted transactions



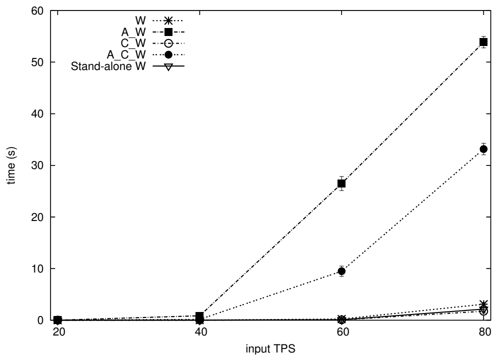
Abortion rate

Figure 6: Weak voting replication – 2 replica nodes

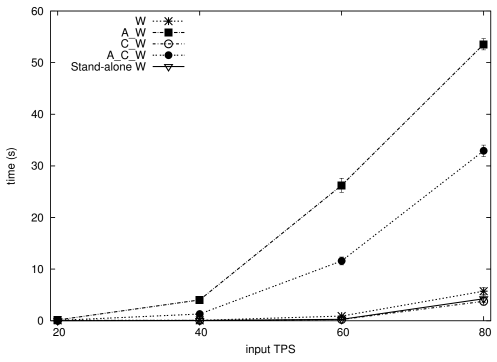
Weak Voting Replication Graphs from Figure 6 correspond to weak voting protocols in a system of 2 replicas. Again, the observed trends for committed and aborted transactions are similar. In this case, the difference in time from stand-alone and W configurations is greater that in other replication techniques. Moreover, it is also the first time that a configuration using the metaprotocol –the C_W combination– is clearly faster than the stand-alone version of the replication protocol. This reinforces the idea of the weak voting technique being penalized by a centralized process –the validation of the delegate– which forces non-delegate nodes to wait for the second broadcast. On the other hand, the weak voting technique is an excellent option when readsets must be also considered for validation, as this technique avoids the collection and transmission of readsets.

As seen in the graphs, mixing certification-based transactions with weak voting ones lightens the system, achieving lower response times. On the other hand, as previously noticed, mixing active transactions with weak voting ones has undesirable consequences, which can be improved adding certification-based transactions to the combination.

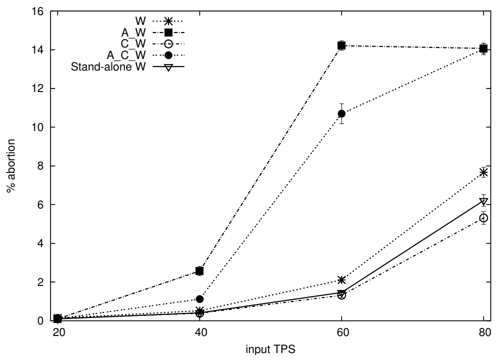
With regard to the abortion rate, curves corresponding to combinations A_W and A_C_W stand out from the rest. The same as before, the longer the transactions, the greater the possibility of conflicts with concurrent transactions.



Length of committed transactions



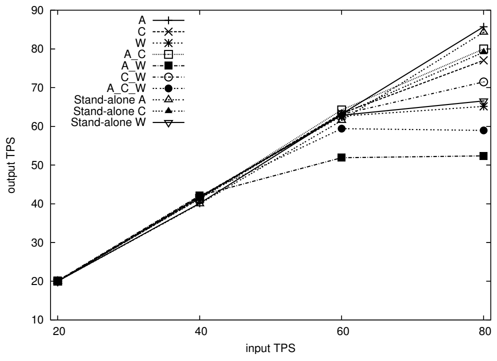
Length of aborted transactions



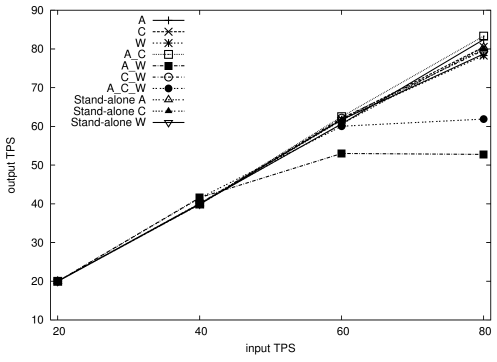
Abortion rate

Figure 7: Weak voting replication – 4 replica nodes

Figure 7 presents the graphs obtained for weak voting protocols with 4 replicas. Again, the behavior observed in 2-replica systems is here maximized and we clearly see how A_W and A_C_W combinations obtain times and abortion rates dramatically higher than those of other configurations.



With 2 replica nodes



With 4 replica nodes

Figure 8: Output TPS

Output TPS Finally, we include graphs from Figure 8, which show the output TPS –the amount of committed transactions per second–. In these graphs we can see how some combinations degrade their throughput as the system load increases. As already seen in previous graphs, all configurations are similar at low loads, but their curves diverge when more and more transactions are initiated per second. Again, configurations A_W and A_C_W show the poorest performance, being the gap with the other curves greater when increasing the system size, as explained before.

Final remarks about experimental results The clear conclusion about the experimental results here presented is that configurations that mix active and weak voting replication should be avoided if high loads need to be supported or large system sizes are used. In these cases, such protocol concurrency reduces system performance, as clearly shown in all the graphs. Nevertheless, if performance is not as important as supporting concurrent client applications of different requirements, these combinations may be very useful.

On the other hand, the C_W combination has shown an excellent performance in a 4-replica system and also a low overhead when using only 2 replicas. This good behavior can be combined with the flexibility already offered at DBMS level by Microsoft SQL Server –which concurrently supports snapshot and serializable isolation levels–, thus allowing our metaprotocol to provide such degree of adaptability at middleware level of replicated environments.

4 Related Work

No academic results, apart from our previous work in [14] which in turn follows the work started in [3], present a valid solution for a database replication system capable of supporting concurrent replication protocols.

A different approach was developed in [16], where a single replication protocol was able to support multiple isolation levels. However, the resulting protocol was complex, and it did not achieve the modularity and maintainability that our metaprotocol provides.

Nevertheless, the general problem of protocol exchange, i.e. dynamic protocol update (abbr., DPU), has been discussed in multiple works, many of which are specifically oriented to group communication systems that achieve adaptability by exchanging the current communication protocol. However, DPU solutions are interested in replacing the working protocol for another one, by means of some sort of synchronization between nodes, and do not consider concurrency of several protocols –except, perhaps, for a short transition phase–.

[2] studies the adaptability provided by RAID-V2, a distributed database system in which three components have built-in adaptability features: the concurrency controller (CC), the replication controller (RC) and the atomicity controller (AC). Each of these elements implements several algorithms and offers the mechanism to convert from one algorithm to another. In that system, protocol replacement is based on a fully replicated relation, the control relation, which contains one row for each site and is updated by special control transactions. An update of a row in this table is interpreted by the corresponding server as a dynamic adaptability request.

[5] also addresses DPU in the context of database replication. Its aim is to provide adaptability to mobile systems, where disconnected nodes can work with weaker consistency levels but switch to stronger levels when connected to each other. This protocol replacement, based on uniform reliable multicast and uniform consensus, is performed by means of a metaprotocol, which enforces three properties also defined in that paper.

In the field of collaborative systems, [22] proposes an approach for supporting adaptable consistency protocols, based on the separation of data and control. This solution, however, is not oriented to transactional systems and relies in a centralized server, which introduces a single point of failure.

In the scope of group communication, [19] presents Ensemble, a network protocol architecture, which provides adaptation by means of replacing the entire protocol stack. A Protocol Switch Protocol (PSP) coordinates the reconfiguration by finalizing the old stack and starting the new one. But, as the whole stack is replaced, communication must be stopped before the adaptation.

A general solution to the DPU problem was presented in [4], whose system model is a distributed system in which each layer can be adaptive and where algorithms are changed after a three-step process: change detection, agreement and adaptive action. In order to do so, each adaptive component (AC) consists of two types of modules: a component adaptor module (CAM) and several adaptation-aware algorithm modules (AAMs). During the changeover, both AAMs are active and, thus, processing and message exchange do not need to be halted. However, the changeover requires communicattion between the CAM and the AAMs and some activating/deactivating functions in each AAM. This forces to extend the protocol modules. To solve this problem, [15] proposes an architecture in which protocol modules are not even aware that the replacement takes place. This solution, based on services –specifications of distributed protocols– rather than on protocols –implementations of distributed protocols–, is modular and highly flexible.

A solution specific for the total order broadcast primitive is proposed in [8]. In this paper, authors define an Adaptive Group Communication System (AGCS) where an interceptor and a switching protocol carry out the replacements. The changeover is performed by queueing the messages broadcast by the user process until the new total order protocol is working. Once the messages sent through the first protocol have been totally ordered, the second protocol can be activated and the queued messages can be broadcast with it.

[10] also proposes a solution for an adaptive total order algorithm. In this solution, however, messages do not need to be buffered nor stopped during the replacement. On the contrary, messages are sent using both algorithms until a safe point is reached, when the old protocol can be stopped. Additional care must be taken to deliver each of these double-broadcast messages only once.

5 Conclusions

Adaptability is a desirable feature for all systems, especially for those that present very different response times when the environment changes. Moreover, client applications of database replication systems may demand different requirements that can be better served with different replication techniques.

We study here 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 cause concurrency to be penalized. We have shown and explained that certain combinations of protocols should be avoided when performance is a major system goal. Other protocol combinations, however, have shown excellent performance and scalability.

In future works, we will provide new metaprotocol revisions that support other database replication techniques, and extend the current correctness arguments given in Section 2.2, providing a complete correctness proof. We will also study possible optimizations for increasing performance during concurrency. Finally, a load monitor will be developed in order to automatically decide which protocol is the best option for each transaction, depending on relevant environmental characteristics.

Acknowledgements

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.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[2] B. Bhargava, K. Friesen, A. Helal, and J. Riedl. Adaptability Experiments in the RAID Distributed Database System. *Symposium on Reliable Distributed Systems*, pages 76–85, 1990.

[3] F. Castro-Company and F. D. Muñoz-Escoí. An Exchanging Algorithm for Database Replication Protocols. Technical report, Instituto Tecnológico de Informática, Valencia, Spain, 2007.

[4] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting. Constructing Adaptive Software in Distributed Systems. In *Intnl. Conf. on Distributed Computing Systems*, pages 635–643. IEEE Computer Society, 2001.

[5] U. Fritzke Jr., R. P. Valentim, and L. A. F. Gomes. Adaptive Replication Control Based on Consensus. In *SDDDM '08: 2nd workshop on Dependable distributed data management*, pages 1–10, New York, NY, USA, 2008. ACM.

[6] 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. *Euro-Par 2005, Lecture Notes in Computer Science*, 3648:349–359, 2005.

[7] 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*, pages 24–33. IEEE Computer Society, 2001.

[8] E. Miedes de Elías, M. C. Bañuls Polo, and P. Galdámez Saiz. Group Communication Protocol Replacement for High Availability and Adaptiveness. In *Jornadas de Concurrencia y Sistemas Distribuidos*, pages 271–276. Thomson Paraninfo, 2005.

[9] J. M. Milán-Franco, R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Adaptive Middleware for Data Replication. In *Middleware*, volume 3231, pages 175–194. Springer-Verlag, 2004.

[10] J. Mocito and L. Rodrigues. Run-Time Switching Between Total Order Algorithms. *Euro-Par 2006, Lecture Notes in Computer Science*, 4128:582–591, 2006.

[11] 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*, pages 401–410. IEEE Computer Society, 2006.

[12] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. In *Intnl. Euro-Par Conf.*, pages 513–520, Southampton, UK, 1998.

[13] PostgreSQL. Web site. *http://www.postgresql.org*, 2008.

[14] 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 *Intnl. Workshop On Reliability in Decentralized Distributed Systems*, volume 4806, pages 1052–1061. Springer-Verlag, 2007.

[15] 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.

[16] R. Salinas, J. M. Bernabé-Gisbert, and F. D. Muñoz-Escoí. SIRC, a Multiple Isolation Level Protocol for Middleware-based Data Replication. In *Intnl. Symposium on Computer Information Sciences*, Ankara, Turkey, 2007. IEEE-CS Press.

[17] Spread. Web site. *http://www.spread.org/*, 2008.

[18] K. L. Tripp and N. Graves. Sql Server 2005 Row Versioning-Based Transaction Isolation. Technical report, Microsoft, 2006.

[19] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building Adaptive Systems using Ensemble. *Softw. Pract. Exper.*, 28(9):963–979, 1998.

[20] M. Wiesmann and A. Schiper. Comparison of Database Replication Techniques Based on Total Order Broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, 2005.

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

[22] Y. Yang and D. Li. Separating Data and Control: Support for Adaptable Consistency Protocols in Collaborative Systems. In *ACM Conf. on Computer supported cooperative work*, pages 11–20, New York, NY, USA, 2004. ACM.