Improving DBMS Performance through Diverse Redundancy

Vladimir Stankovic, Peter Popov
Centre for Software Reliability,
City University,
Northampton Square,
London EC1V 0HB,
United Kingdom
V.Stankovic@city.ac.uk, Ptp@csr.city.ac.uk

Abstract

Database replication is widely used to improve both fault tolerance and DBMS performance. Non-diverse database replication has a significant limitation - it is effective against crash failures only. Diverse redundancy is an effective mechanism of tolerating a wider range of failures, including many non-crash failures. However it has not been adopted in practice because many see DBMS performance as the main concern.

In this paper we show experimental evidence that diverse redundancy (diverse replication) can bring benefits in terms of DBMS performance, too. We report on experimental results with an optimistic architecture built with two diverse DBMSs under a load derived from TPC-C benchmark, which show that a diverse pair performs faster not only than non-diverse pairs but also than the individual copies of the DBMSs used. This result is important because it shows potential for DBMS performance better than anything achievable with the available off-the-shelf servers.

1. Introduction

The most important non-functional requirements for a Database Management System (DBMS) are performance and dependability, which often require mutually exclusive mechanisms. Thus, a trade-off between the two is sought, which would be optimal for a specific system.

Data replication has proved to be a viable method of enhancing both dependability and performance of DBMSs. Performance is improved by balancing the load between the deployed replicas, while fail-over mechanisms are normally used to re-distribute the load of a failed replica among the remaining operational ones. Crashes are commonly believed to be the main type of failure of DBMSs. Providing that only crashes occur, using several identical replicas provides appropriate protection. Under this assumption the replication scheme ROWAA (read once write all available) is adequate [1]. Unfortunately, this common belief is hard to justify. In a recent study, we presented overwhelming evidence against crash failures being the main concern [2]. Using the log of known bugs reported for four major DBMSs we observed for all four servers that more than 50% of the known bugs lead to non-crash failures, which will not be tolerated by a non-diverse replication. Only by deploying diverse redundancy, i.e. deploying diverse replicas, would we deliver an adequate protection against the non-crash failures of the DBMSs.

A possible architecture for a fault-tolerant server employing (diverse) redundancy is depicted in

Figure 1. The middleware propagates the statements generated by the client applications to both (all, in case of more than 2) diverse replicas for execution. The results from the replicas are collected by the middleware and in the case of a positive adjudication the middleware reports a result back to the client application(s). Clearly, this architecture differs from the ROWAA scheme. In the new architecture all statements (including the reads from the database) are executed multiple times by several diverse replicas, while in the ROWAA scheme all active replicas execute only the writes to the databases.

While dependability gains from deploying diverse redundancy are beyond doubt, it is far from obvious what the implications of this architecture would be for system performance. From the known applications of design diversity in other areas, it is well known that fault-tolerant mechanisms (failure detection, fault-containment, state recovery, etc.) have their performance cost. Is diverse redundancy then

necessarily a bad thing in terms of system performance?

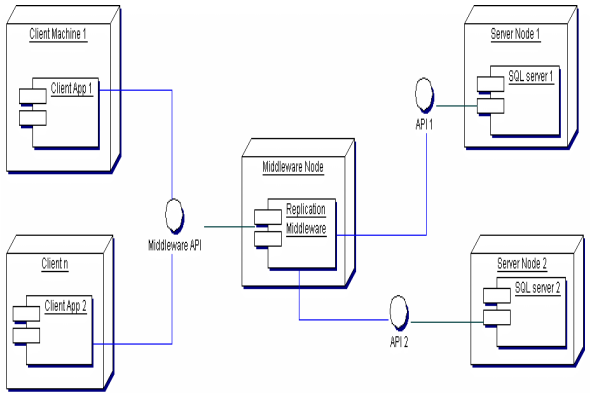

Figure 1. Fault-tolerant server node (FT-node) with two (possibly more) diverse DBMSs (SQL server 1 and SQL server 2). The middleware "hides" the servers from the clients (1 to n) for which the data storage appears as a single DBMS.

The overall performance of the system shown in Figure 1 will depend on the performance of the diverse replicas deployed and on the performance characteristics of the middleware itself. For instance, the middleware can use different adjudication mechanisms. A few reasonable alternatives are listed below:

- *Slowest response*. The middleware collects the results of the *individual statements* (a multitude of which constitute a whole transaction) executed by diverse replicas. Once a sufficient number of responses are collected, they are adjudicated and only if identical responses from all the replicas are observed a successful completion of the statement is reported back to the client application.
- *Fastest response.* Alternatively, the middleware may buffer the statements coming from a client application and make them available to the diverse replicas as soon as the statements are placed in the respective buffers. Each diverse replica collects the next available statement from its respective buffer, executes it, marks it as being completed and makes the response from the statement available to the middleware. As soon as the middleware receives the first response to a statement from a replica, it is immediately passed on to the client application, thus letting the client application proceed with the other statements within the transaction. The fastest response comes from either of the DBMSs, depending on the SQL statement (**Figure** 2)**.** Responses from the diverse replicas to the same statement are adjudicated later, when a sufficient number of responses are collected, but before the end of the transaction. Buffering the statements in the middleware allows the diverse replicas to work at a maximum speed within transactions, as shown in **Figure** 2 (DBMS1 would start execution of the next SQL statement even though the DBMS2 has not finished the previous one as indicated with the dashed rectangle). The transactions are committed (or aborted) based on the outcome of adjudicating the results of the statements. Commit is only applied if all the replicas execute all the statements successfully and all the statement responses are positively adjudicated. Otherwise, the transaction is aborted.
- *Optimistic response*. This is similar to the fastest response except: i) *no adjudication* of the responses from the diverse replicas is applied; ii) a *skip* feature is implemented in the middleware as follows. Before a replica, X, executes a *read* (i.e. SELECT) *statement* it checks if a response to this statement has already been received from another replica, Y≠X. If so then X does not execute the statement (i.e. skips it)[1]. The modifying SQL statements (DELETE, INSERT and UPDATE) are executed on all servers, i.e. they cannot be skipped. Clearly, this regime of operation does not offer the same level of protection as the previous ones. It may, however, be adequate in many cases, which we discuss later (see the Discussion section).

We have already [3] on systematic differences between the times it takes diverse DBMSs to execute the same statement. This may be due, for example, to the respective execution plans being different, the concurrency control mechanisms being implemented differently, etc. When the *slowest response* regime is used such differences will lead to the fault-tolerant node (FT-node) being slower than the respective DBMSs it consists of. When the *optimistic regime* is used, however, the systematic difference might lead to improved performance. If the mix of statements within a transaction is such that both servers 'skip' statements,

1 The functionality of looking up the next statement and the 'skip' feature is, of course, implemented in the middleware, which relays to the DBMSs the statements for execution. If a read statement is to be skipped, then the middleware simply does not pass it to the respective DBMS for execution.

then the transaction will take the FT-node shorter than it would take each of the DBMSs it uses. When the 'skip' feature is not used the best that the FT-node can do is process SQL statements as fast as the faster of the two servers can, thus diversity cannot bring any performance gains.

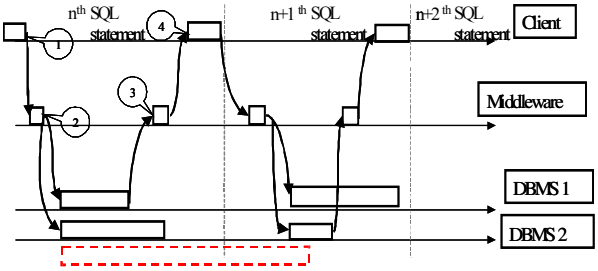

Figure 2. Timing diagram of a client communicating with two, possibly diverse, database servers and the middleware running in fastest response or optimistic regime. The meanings of the callouts are: 1 – the client sends an SQL statement to the middleware, 2 – the middleware translates the request to the dialects of the servers and places the resulting SQL statements, or sequences of SQL statements, in the respective server buffers; 3 – the fastest response is received by the middleware; 4 - the middleware sends the response to the client. The dashed rectangle indicates that DBMS2 will not be ready to start (n+1)th SQL statement at the same time with DBMS1

This paper, therefore, is focused on the empirical investigation of whether the potential performance gains with the optimistic regime of operation of the FT-node can be achieved under a realistic load, such as the one defined by the TPC-C performance benchmark [4]. Whichever regime under the FT-node operates, data consistency between the diverse replicas must be guaranteed, which is typically defined as *1-copy serialisability* between the transaction histories of the replicas [1].

Should a level of replication be required that is higher than the number of diverse replicas used in a single FT-node, then the FT-node can be combined with any database replication scheme, which is considered adequate for a particular set of requirements. These can be schemes for *eager* database replication, e.g. based on group communication primitives [5], or even *lazy* replication [6]. In either case the FT-node will replace a replica of a particular DBMS used by the particular database replication scheme.

This paper is structured as follows. In section 2 we describe the experimental setup. In section 3 we enumerate possible configurations of the FT-node. In section 4 we show consistency of the experimental results. In section 5 we present the performance comparison of different FT-node configurations. In section 6 we compare performance of the diverse pair and a non-diverse solution. In section 7 we discuss possible performance gains when diverse DBMSs are used and in section 8 we present conclusions made and describe provisions for future work.

2. Experimental Setup

In the empirical study we used our own implementation of the industry-standard benchmark for online transaction processing - TPC-C [4], to evaluate the potential for performance improvement. TPC-C defines five types of transactions: *New-Order (NO), Payment (P)*, *Order-Status (OS), Delivery (D)* and *Stock-Level (SL)* and sets the probability of execution of each. The minimum probability of execution for each transaction type is as follows: NO – 43%, P – 43%, OS – 4%, D – 4% and SL – 4%. The benchmark provides a mechanism for performance comparison of the DBMSs from different vendors, with different hardware configurations and operating systems. The specified measure of throughput is the number of *NO* transactions completed per minute (under the specified mix of transaction types). Our measurements were more detailed than those required by the standard. We recorded the response times of the individual SQL statements and transactions executed by the DBMSs used in the FT-node. The test harness consisted of three machines:

- a client machine, which executes a JAVA implementation of the TPC-C standard (it uses JDBC to access the DBMSs);
- two server machines, on which two diverse open-source DBMSs are run, namely InterBase 6.0 and PostgreSQL 7.4.0 (referred to as IB and PG, respectively, in the rest of the paper).

The two DBMSs ran on Linux RedHat 6.0 (Hedwig) operating system, while the client machine ran under Windows 2000 Professional (sp4) operating system. The hardware specifications are as follows:

- *client machine*: 1.5 GHz Intel Pentium 4 processor, 640 MB RAMBUS RAM and 20GB HDD (Maxtor DiamondM)

- *server machines*: 1.5 GHz Intel Pentium 4 processor, 384 MB RAMBUS RAM and 20GB HDD (Seagate U Series).

The implementation of the TPC-C application did not necessitate the use of any proprietary features from either IB or PG. The SQL statements were implemented using the common subset of the language. Nevertheless we have developed preliminary versions of our own SQL translator tool. In addition one could make use of commercial products for porting between different DBMSs such as *Fyracle* [7], Oracle-mode Firebird or its PostgreSQL counterpart *EnterpriseDB* [8].

3. FT-node Configurations

We run a set of experiments with the following server configurations:

- 1IB1PG, an FT-node with a copy of IB and PG. 1IB,
- a single replica of IB;
- 1PG, a single replica of PG;
- 2IB, an FT-node with two replicas of IB, and
- 2PG, an FT-node with two replicas of PG.

Each experiment comprises the *same sequence* of 10,000 transactions and was repeated five times, for reasons detailed below. The server machines were restarted and databases restored between the repetitions.

All the measurements were associated with a single TPC-C client under different *server loads* as follows:

- no additional clients;
- 10 additional clients, and
- 50 additional clients.

Whenever additional clients were deployed they executed a mix of read-only transactions (RO mix) instead of the mix of transactions recommended by the TPC-C[2]. The RO mix consists of the two read-only transactions: *Order-Status* and *Stock-Level* of almost equal proportion. Thus, only one TPC-C compliant client modifies the database. The *readers* and *writers* do not conflict in the two DBMSs, since both IB and PG implement a type of MVCC (Multi-Version Concurrency Control). Hence data consistency between the replicas is guaranteed (experimentally confirmed by successfully running a comparison between the databases at the end of the experiments).

The overhead that the test harness introduces (mainly due to using JAVA multi-threading for communication of the clients with the middleware and of the middleware with the different DBMSs) is the same irrespective whether a single or two replicas are used in the experiment. It has been measured to be negligible compared with the time taken by the respective DBMSs to process the 10,000 transactions.

4. Confidence in the Results

Each experimental setup (with a fixed configuration and load) was repeated *five times* so that we could detect significant variation between the observed results due to factors beyond our control (e.g. fragmentation of files on the servers).

Figure **3** shows the mean transaction times for all transactions together and per transaction type in a 10,000-transaction run, grouped by experiment repetitions when only a single TPC-C compliant client is deployed. There is no significant variation between the results across the repetitions. This is true for both a particular transaction type and all transactions together.

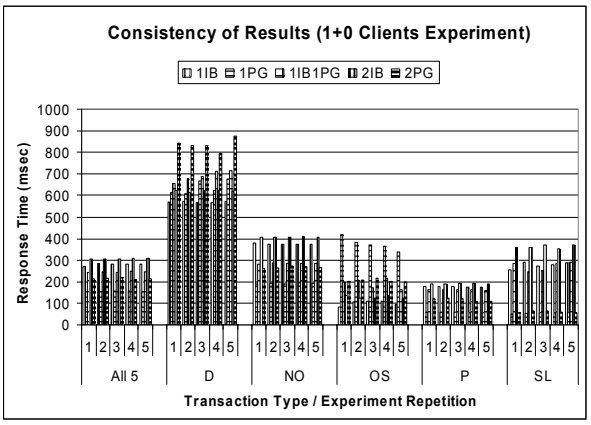

Figure 3. Mean transaction times per transaction type and for all transactions together for 5 repetitions with each of the configurations (1IB, 1PG, 1IB1PG, 2IB, 2PG) with a load generated by a single TPC-C compliant client.

2 We did run multiple concurrent TPC-C clients with our own implementation of 1-copy serialisability between the DBMSs. These experiments, however, did lead to a very large number of non-serialisable transactions, which had to be aborted. Due to non-determinism between the orders in which the servers serve the concurrent clients we could not achieve a repeatable set of experiments to make a fair comparison between the different server configurations. Thus, we chose to restrict the results presented here to experiments with a single TPC-C compliant client while simulating the increased load by deploying an increasing number of read only clients.

A similar picture, consistent across the repetitions, was established for the increased load of 10 and 50 additional clients (Figure **4**). The only configuration with a noticeable variation between the repetitions was 1IB. In particular, the first run is 25% faster than the remaining four in terms of the mean transaction time with all transaction types (represented by the first bar in each of the five groups above the "All 5" category). A noticeable variation also exists between the specific transaction types, for which the percentages vary between 20% and 25%. This variation, however, does not change the ordering between the configurations.

In addition the ordering between the configurations does not change even if we execute a different sequence of transactions. This was experimentally confirmed by executing 10,000 transactions in different order with either a single TPC-C compliant client or with ten additional clients.

Such consistency between the observations, in particular, the fact that the ordering between the configurations remains unchanged across the repeated experiments, is the reason why in the rest of the paper we compare the performances using a single run per configuration.

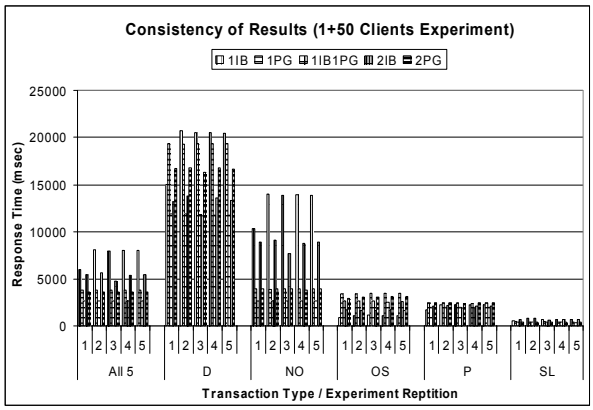

Figure 4. Mean transaction times per transaction type and for all transactions together for 5 repetitions of each experiment type (1IB, 1PG, 1IB1PG, 2IB, 2PG) under increased load with 50 additional read-only clients.

5. Performance Comparison of Different DBMS Configurations

To compare different DBMS configurations we used the following measures of interest:

- mean transaction time (for all five transaction types);
- mean transaction time for a particular type of transaction;
- cumulative transaction time, i.e. experiment duration.

Figure **5** depicts the response time when only a single TPC-C client communicates with the FT-node configurations. 1PG is on average the best configuration under this load, though transactions of type Delivery and Order-Status are faster on 1IB. The ranking changes when the load increases (Figure 6). Now the fastest configuration on average is the diverse pair, albeit not for all transaction types (1IB is the fastest for Order-Status and Payment, while 1PG is the fastest for Stock-Level). The figure indicates that the diverse DBMSs "complement" each other in the sense that when IB is slow to process a transaction then PG is fast (New-Order and Stock-Level) and vice versa (Payment, Order-Status and Delivery). These systematic differences illustrate why the 1IB1PG diverse pair is the best configuration on average. In addition the 'skip' feature enables the diverse pair to augment this advantage by omitting the read (SELECT) SQL statements on the slower DBMS.

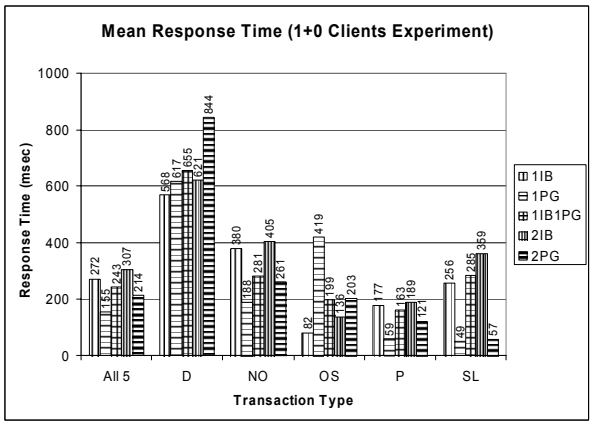

Figure 5. The mean transaction times for each transaction type and for all transactions together under a load generated by a single TPC-C compliant client. The configurations compared under this load are as follows: configurations with a single DBMS (1IB, 1PG), a configuration with a diverse pair of DBMSs (1IB1PG) and configurations with homogenous pairs of DBMSs (2IB, 2PG).

Although a DBMS is fastest on average for a particular transaction type, within the transactions the fastest responses to SQL statements may come from different DBMSs. This fact is utilised in the diverse pair. Hence, it is not surprising that IB executes more SELECT statements in an experiment than PG when the two are employed as a diverse pair (IB executes 70%, while PG executes 51%)[3].

Similar results were obtained under the load with 50 additional clients.

Figure 7 shows how the ordering changes between the configurations as a result of a load increase. An experiment comprising 10,000 transactions under the 'lightest' load (0 additional clients) is fastest with 1PG. Under increased load, however, the diverse pair, 1IB1PG, becomes the fastest configuration. The experiment duration with the diverse pair is shorter than with the individual DBMSs, or with either of the non-diverse (homogenous) DBMS pairs. The diverse pair is 20% faster than the second best configuration (1PG) with 10 additional clients and more than 25% faster than the second best combination (2PG) with 50 additional clients. The benefits of the systematic difference in transaction times between the diverse DBMSs and the efficiency of the 'skip' feature become more clearly pronounced when the load increases.

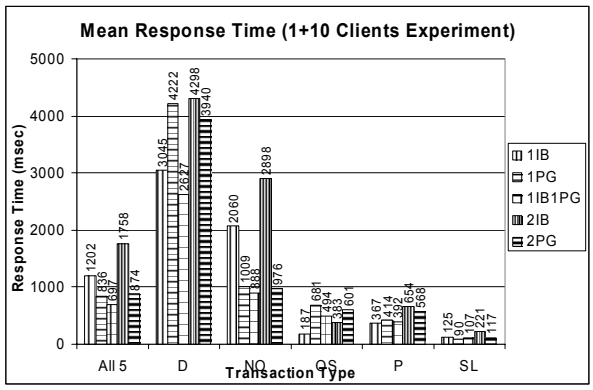

Figure 6. The mean transaction times for single DBMS configurations (1IB, 1PG), diverse DBMSs pair (1IB1PG) and homogenous DBMS pairs (2IB, 2PG) for each transaction type and for all transactions together under an increased load with 10 additional read-only clients.

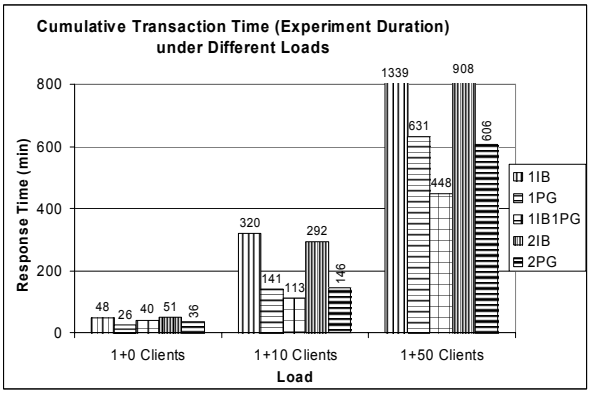

Figure 7. Cumulative transaction time (experiment duration) for the five DBMS configurations under different load (0, 10 and 50 additional read-only clients).

6. Comparison of Diverse Pair and a Non-Diverse Solution

In this section we compare the performance of the diverse pair and of a well-known solution for eager data replication [9]. The solution uses non-diverse redundancy. It combines transactional concurrency control and group communication primitives in order to guarantee data consistency (referred to as TCC+GCP in the remainder of the document) among replicas. It provides both fault-tolerance and good performance.

In order to guarantee a 1-copy serialisability, TCC+GCP relies on "totally ordered" [9] delivery of transactions using a reliable multicast protocol. It is based on the ROWAA (Read-Once Write All Available) protocol [1].

Under this replication scheme the clients served by TCC+GCP connect to only one replica, called the *local replica* of the client. For this client the other replicas of the TCC+GCP are remote replicas. A read-only transaction generated by a client is executed by the local replica, only. A *write* transaction (i.e. one that includes write statements) is first executed by the local replica. The outcomes of the write statements are then broadcast (by the respective middleware) to the remote replicas of TCC+GCP in the form of *write sets*. The remote replicas install the *write sets* according to the

[3] There is nothing unusual in the fact that the sum 70% + 50% is greater than 100%. It simply means that there are statements which are executed by both servers. If the fastest server has not completed a statement by the time the slower is ready to start, then both will process the particular statement.

total order of transactions established among the replicas used in TCC+GCP.

Clearly, with TCC+GCP the read-only transactions are load-balanced between the replicas. Ideally, the clients should be fairly divided between the replicas. A fair performance comparison, thus, of TCC+GCP and an FT-node with two diverse DBMSs, would require the following arrangement:

- a single DBMS working in TCC+GCP will be subjected to the write transactions load generated by all clients and half of the load generated by the read-only transactions generated by the clients;
- the FT-node handles the entire load, both from write and read transactions, generated by all the clients.

We ran experiments for loads generated by a single TPC-C client and additional read-only clients: 10 and 50. To make a fair comparison between an FT-node and TCC+GCP, we used the results measured for the FT-node (see above) and run a new set of experiments with 5 and 25 read-only clients respectively with an FT-node. We *simulated* the performance of the TCC+GCP using the measurements obtained with the FT-node under the new loads (with 5 and 25 read-only clients).

We calculated a *lower* and an *upper bound* on the TCC+GCP transaction times as follows. The lower bound is the *actual transaction time* measured in the new experiments with the individual DBMSs and the number of read-only clients equal to 5 and 25, respectively. This lower bound seems unattainable by TCC+GCP, because the installation of the *write sets* (especially the *lock phase*) [10] on the remote replicas, as well as on the local replicas is not accounted for in the lower bound. Installing the write sets is on the *critical path* – it is always done *after* the local replica creates the write sets. The *upper bound is calculated* from the experimental log (in which we record the start and completion times of the individual statements) by doubling the execution time of all *write* SQL statements (DELETE, INSERT and UPDATE) encountered during the experiment. Whether the bound is indeed an upper bound is moot since it is unclear whether the actual overhead due to group communication primitives and the actual installation of the *write sets* by all the replicas is greater or smaller than the time it takes the local replica to execute a write statement. The upper bound may be too pessimistic, if the mentioned overheads are negligible compared with the write statements execution times. On the other hand, however, a simplistic implementation of the write sets would be forwarding them to the remote replicas, which in turn will actually execute them. Under this simplistic scenario, our upper bound will be in fact too optimistic because it does not account for the overhead due to propagating the write sets to the remote replicas. In summary, the realism of the upper bound is questionable and should be scrutinised in the future, ideally by actually implementing TCC+GCP. Despite this problem, however, using the lower and the upper bounds allows us to get preliminary indications of how the performance of FT-node compares with TCC+GCP.

Figure **8** presents the results of a fair comparison between the two replication schemes under a load with 1 modifying and 50 additional read-only clients. Diverse pair performs clearly better than TCC+GCP if IB is used: the transaction times of the FT-node is lower than the lower bound (unattainable by TCC+GCP). The diverse pair is also better (~ 15%) than the upper bound of TCC+GCP with PG. It is, however, worse than the lower bound of TCC+GCP with PG. The diverse pair is ~30% slower than the lower bound. Thus, it remains unclear whether TCC+GCP with PG is faster than the FT-node. Similar results have been observed in all repetitions with this load.

Similar ordering between the FT-node and the TCC+GCP has been observed with a lower load of only 10 additional read-only clients. Again, only the lower bound of the non-diverse replication with PG is faster than the diverse pair. However, the difference between the diverse pair and the upper bound is smaller.

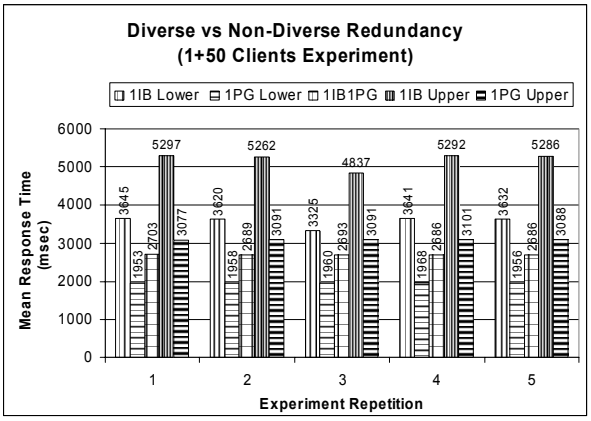

Figure 8. Mean response times for diverse replication (1IB1PG) and calculated lower and upper bound of mean response times for a non-diverse replication schema when either

1IB or 1PG is used. The lower bound is calculated using results from respective individual DBMS (IB or PG) experiment under the load with 1 modifying and 25 read-only clients. The number of read-only clients was halved because the non-diverse schema uses a load balancing approach where reads are executed at only one node. To estimate the upper bound, mean response time of transactions' *write sets* was added to the lower bound.

7. Discussion

Performance implications of using diverse redundancy in the context of database replication are the focus of this paper. Diverse redundancy is the only known realistic protection against *design faults* in complex software products. Once diverse redundancy is deployed there might exist performance implications, which we evaluated empirically.

A standard fault-tolerant architecture (see Figure **1**) would dictate adjudicating all the responses from (a sufficient number of) diverse replicas before a response is returned back to a client application, i.e. adjudication is applied at the level of individual statements. This adjudication, normally implemented by a specialised middleware, can be done as the responses are received (referred to as the *slowest response*) or postponed and completed before the end of the corresponding transaction (the *fastest response* regime). Either way, fault-tolerance will lead to performance penalty and the FT-node cannot be faster than the fastest of the deployed DBMSs.

The schemes adopted for practical database replications provide no protection against design faults. A common assumption is made that node crashes are the main concern, an assumption under which various optimistic regimes of operations are used such as ROWAA. These do not require statement adjudication and as a result the adjudication overhead is simply avoided.

Failures of DBMSs are rare. Most of the time the applications use statements that are handled correctly by the deployed DBMSs. Even if diverse DBMSs are deployed most of the statements will be handled correctly by all the diverse replicas. Thus, most of the time adjudicating the responses of diverse replicas will reveal no discrepancy, making the adjudication overhead a waste of time. The point, of course, is that we will never know which statement will turn out to trigger a fault in the DBMSs and revealing a discrepancy between the replica responses. In some extreme cases, however, *one may know with certainty*, that all the statements used by the application will be processed by the DBMSs correctly; hence one may be prepared to use regimes in which the adjudication is eliminated. One such example is the implementation of the optimistic regime. Its advantage compared with the well-known ROWAA regime of operation lies in the fact that under ROWAA the load is statically distributed between the replicas – in the ideal case a fair load-balancing between the replicas is sought. Instead, when the FT-node operates under the optimistic regime its diverse replicas *naturally get the load that they are better at executing*. As a result the optimistic mode has the potential of performing better than ROWAA. Unfortunately, our experiments did not provide a conclusive answer as to whether this potential can be materialised, but it did not refute it either. Further, more accurate measurements, possibly using proper implementation of the replication schemes based on ROWAA will provide a definitive answer.

It is worth pointing out that the 3 regimes of operation of the FT-node listed above (slowest response, fastest response and optimistic) are not mutually exclusive. In fact, they can be combined to offer *configurable quality of service*, as we argued elsewhere [3]. The clients mainly concerned with high dependability assurance can be served under the slowest response regime. The clients mainly interested in maximising the performance can be served under the optimistic regime of operation. Finally, by deploying learning capabilities, e.g. based on Bayesian inference, [11], the middleware may become capable of switching intelligently between the different regimes of operation: initially a new type of statement (e.g. SQL statement involving a complex and rarely executed JOIN operation) will be treated by the middleware with suspicion and the most conservative, slowest response, regime of adjudication will be applied. As more instances of the same statement are executed successfully (i.e. the adjudication is passed successfully in all the observed instances), then the middleware will switch from slowest response through the fastest response eventually to the optimistic regime of operation. Clearly, adjudication is simply impossible with ROWAA, thus the scope for trading-off intelligently performance for improved dependability assurance is very limited, if not impossible.

8. Conclusions and Future Work

The results presented here show an intriguing possibility to get a *performance gain*, in some cases very substantial, when diverse redundancy is used in the context of database replication. We compared diverse with non-diverse redundancy using an optimistic architecture, FT-node, in which the variation between the execution times of the diverse replicas is turned into a performance advantage. In this setup, diverse redundancy is clearly beneficial compared with non-diverse redundancy.

We also compared non-replicated solutions (a single copy of a DBMS) with an FT-node, in which a diverse pair of DBMSs is deployed. Diverse pair performs significantly faster than the non-replicated solution.

These two results seem very significant since they open up new ways of achieving high performance, especially when the main system concern is achieving as high a performance as possible.

We also looked at how diversity performs against eager replication solutions based on total transaction order (TCC+GCP), which use load balancing for improved performance. The comparison, performed under various simplifying assumptions, is *indecisive* in the general case. Diverse redundancy is not guaranteed to always achieve a known lower bound of performance for those solutions, although we recorded that the diverse pair performed better than TCC+GCP implemented with replicas of Interbase 6.0. This lower bound, however, is unattainable for TCC+GCP too! The performance of diverse redundancy is better than the likely upper bound on the performance of TCC+GCP with replicas of PostgreSQL 7.4. In some cases of simple implementations of TCC+GCP, e.g. the write sets being propagated to the remote replicas in the form of full SQL statements, the upper bound will become a lower bound on the performance of TCC+GCP. For this implementation of TCC+GCP we have a decisive argument in favour of diverse redundancy: it is guaranteed to be faster than TCC+GCP.

In the experiments we used a synthetic load (TPC-C) mainly due to the wide acceptance of the benchmark for performance measurement studies. Although the reported effect is dependant on the mix of SQL statements used we expect similar results in favour of diverse redundancy to be observed for a wide range of real loads. In fact, TPC-C specifies a write intensive mix of statements, not ideal for the optimistic regime of an FT-node. Applications based towards read-only mixes of SQL statements are more suitable for the reported effect to make a bigger impact.

A promising direction for future development is implementation of a configurable middleware, deployable on diverse DBMSs, which would allow the clients to request *quality of service* in line with their specific requirements for performance and dependability assurance.

Acknowledgements

This work has been supported in part by the "Interdisciplinary Research Collaboration in Dependability" (DIRC) project funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC).

Bibliography

1. Bernstain, A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. 1987, Reading, Mass.: Addison-Wesley.
2. Gashi, I., P. Popov, and L. Strigini. *Fault diversity among off-the-shelf SQL database servers*, in *International Conference on Dependable Systems and Networks*. 2004. Florence, Italy: IEEE Computer Society Press.
3. Gashi, I., et al., *On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*, in *Architecting Dependable Systems II*, R. de Lemos, C. Gacek, and A. Romanovsky, Editors. 2004, Springer. p. 191-214.
4. TPC, *TPC Benchmark C, Standard Specification, Version 5.0.* 2002, Transaction Processing Performance Consortium.
5. Patino-Martinez, M., et al. *Scalable Replication in Database Clusters*. In *International Conference on Distributed Computing, DISC'00*. 2000: Springer.
6. Gray, J. and R. Andreas, *Transaction processing: concepts and techniques*. 1993: Morgan Kaufmann.
7. Fyracle, *http://www.janus-software.com/fb_fyracle.html*. 2006.
8. EnterpriseDB, *http://www.enterprisedb.com/products/migration_toolkit.do*. 2006.
9. Jimenez-Peris, R., M. Patino-Martinez, and G. Alonso. *An Algorithm for Non-Intrusive, Parallel Recovery of Replicated Data and its Correctness*. In *21st IEEE Int. Conf. on Reliable Distributed Systems (SRDS 2002)*. 2002. Osaka, Japan.
10. Kemme, B., A. Bartoli, and O. Babaoglu. *Online Reconfiguration in Replicated Databases Based on Group Communication*. In *Int. Conf. on*

Dependable Systems and Networks (DSN 2001). 2001. Goteborg, Sweden: IEEE.

11. Gorbenko, A., et al., *Dependable Composite Web Services with Components Upgraded Online*, in *Architecting Dependable Systems ADS III*, R. de Lemos, C. Gacek, and A. Romanovsky, Editors. in print, Springer. p. 96-128.