City Research Online

City, University of London Institutional Repository

Citation: Gashi, I., Popov, P. T., Stankovic, V. & Strigini, L. (2004). On designing dependable services with diverse off-the-shelf SQL servers. Lecture Notes in Computer Science, 3069, pp. 191-214. doi: 10.1007/978-3-540-25939-8_9

This is the unspecified version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/523/

Link to published version: https://doi.org/10.1007/978-3-540-25939-8_9

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers

Ilir Gashi, Peter Popov, Vladimir Stankovic, Lorenzo Strigini

Centre for Software Reliability,
City University,
Northampton Square
London EC1V 0HB
United Kingdom
http://www.csr.city.ac.uk
{I.Gashi, V.Stankovic}@city.ac.uk,
{Ptp, Strigini}@csr.city.ac.uk

Abstract. The most important non-functional requirements for an SQL server are performance and dependability. This paper argues, based on empirical results from our on-going research with diverse SQL servers, in favour of diverse redundancy as a way of improving both. We show evidence that current data replication solutions are insufficient to protect against the range of faults documented for database servers; outline possible fault-tolerant architectures using diverse servers; discuss the design problems involved; and offer evidence of the potential for performance improvement through diverse redundancy.

1 Introduction

‘Do not put all eggs in the same basket’, ‘Two heads are better than one’ summarise the intuitive human belief about the value of redundancy and diversity as a means of reducing the risk of failure. We are more likely to trust the results of our complex calculation if a colleague has arrived independently at the same result. In this regard, Charles Babbage was probably the first person to advocate using two computers - although by computer he meant a person [1].

In many cases, e.g. in team games, people with diverse, complementary abilities signify a way of improving the overall team performance. Every football team in the world would benefit from having an exceptional player such as Ronaldo[1]. A good team is one in which there is a balance of defenders, midfielders and attackers because the game consists of defending, play making and, of course, scoring. Therefore, a team of 11 Ronaldos has little chance of making a good team.

High performance of computing systems is often as important as the correctness of the results produced. When a system performs various tasks, optimising the performance with respect to only one of them is insufficient; good response time must be achieved on different tasks, similarly to how a good team provides a balanced performance in defence, midfield and attack. When both performance and

[1] At the time of writing the Brazilian footballer Ronaldo is recognised as one of the best forwards in the world.

dependability are taken into account, there is often a trade-off between the two. The balance chosen will depend on the priorities set for the system. In some cases, improving performance has a higher priority for users than improving dependability. For instance, a timely, only approximately correct response is sometimes more desirable than one that is absolutely correct but late.

The value of redundancy and diversity as a means of tolerating faults in computing systems has long been recognised. Replication of hardware is often seen as an adequate mechanism for tolerating 'random' hardware faults. If hardware is very complex, however, e.g. VLSI chips, and hence design faults are likely, then diverse redundancy is used as a protection against hardware design faults [2]. For software faults as well, non-diverse replication will fail to detect, or recover from, all those failures that do not produce obvious symptoms like crashes, or that occur in identical ways on all the copies of a replicated system, and at each retry of the same operations. For these kinds of failures, diverse redundancy (often referred to as 'design diversity') is required. The assumptions about the failure modes of the system to be protected dictate the choice between diverse and non-diverse replication.

Diverse redundancy has been known for almost 30 years [3] and is a thoroughly studied subject [4]. Many implementations of the idea exist, for instance recovery blocks [3], N-version programming [5] and self-checking modular redundancy [6].

Over the years, diverse redundancy has found its way to various industrial applications [7]. Its adoption, however, has been much more limited than the adoption of non-diverse replication. The main reason has been the cost of developing several versions of software to the same specification. Also, system integration with diverse versions poses additional design problems, compared to non-diverse replication [8], [4], [9].

The first obstacle – the cost of bespoke development of the versions - has been to a large extent eliminated in many areas due to the success of standard products in various industries and the resulting growth in the market for off-the-shelf components. For many categories of applications software from different vendors, compliant with a particular standard specification, has become an affordable commodity and can be acquired off-the-shelf.[2] Deploying several diverse off-the-shelf components (or complete software solutions) in a fault-tolerant configuration is now an affordable option for system integrators who need to improve service dependability.

In this paper we take a concrete example of a type of system for which replication can be (and indeed has been) used – SQL servers[3]. We investigate whether design diversity is useful in this domain from the perspectives of dependability and performance.

[2] The difference between commercial-off-the-shelf (COTS) and just off-the-shelf (e.g. freeware or open-source software) is not important for our discussion despite the possible huge difference in cost. Even if the user is to pay thousands for a COTS product, e.g. a commercial SQL server, this is a tiny fraction of the development cost of the product.

[3] Although many prefer relational Databases Management System (RDBMS), we instead use the term SQL server to emphasise that Structured Query Language (SQL) will be used by the clients to interact with the RDBMS.

Many vendors offer support for fault-tolerance in the form of server ‘fail-over’, i.e. solutions with replicated servers, which cope with crashes of individual servers by redistributing the load to the remaining available servers. Despite the relatively long history of database replication [10], effort on standardisation in the area has only started recently [11]. Fail-over delivers some improvement over non-replicated servers although limited effectiveness has been observed in some cases [12]. Fail-over can be used as a recovery strategy irrespective of the type of failure (not necessarily “fail-stop” [13]). However its known implementations assume crash failures, as they depend on detecting a crash for triggering recovery.

The rest of the paper is organised as follows. In Section 2 we summarise the results of a study on fault diversity of four SQL servers [14] which run against the common assumptions that SQL servers fail-stop and failures can be tolerated simply by rollback and retry. In Section 3, we study the architectural implications of moving from non-diverse replication with several replicas of the same SQL server to using diverse SQL servers, and discuss the main design problems that this implies. We also demonstrate the potential for diversity to deliver performance advantages and compensate for the overhead created by replication, and in Section 4 we present preliminary empirical results suggesting that these improvements can indeed be realised with at least two existing servers. This appears to be a new dimension of the usefulness of design diversity, not recognised before. In Section 5 we review some recent results on data replication. In Section 6 we discuss some general implications of our results. Finally, in Section 7 some conclusions are presented together with several open questions worth addressing in the future.

2 A Study of Faults in Four SQL Servers

Whether SQL servers require diversity to achieve fault tolerance depends on how likely they are to fail in ways that would not be tolerated by non-diverse replication. There is little published evidence about this. First, we must consider *detection*: some failures (e.g. crashes) are easily detected even in a non-diverse setting. A study using fault injection [15] found that 2% of the bugs of Postgres95 server violated the fail-stop property (i.e., they were not detected before corrupting the state of the database) even when using the transaction mechanism of Postgres95. 2% is a high percentage for applications with high reliability requirements. The other question is about *recovery*. Jim Gray [16] observed that many software-caused failures were tolerated by non-diverse replication. They were caused by apparently non-deterministic bugs (“*Heisenbugs*”), which only cause failures under circumstances that are difficult to reproduce. These failures are not replicated when the same input sequence is repeated after a rollback, or applied to two copies of the same software. However, a recent study of fault reports about three open-source applications (including MySQL) [17] found that only a small fraction of faults (5-14%) were triggered by transient conditions (probable Heisenbugs).

We have recently addressed these issues via a study on *fault diversity* in SQL servers. We collected 181 reports of known bugs reported for two open-source SQL servers

(PostgreSQL 7.0 and Interbase 6.0[4]) and two commercial SQL servers (Microsoft SQL 7.0 and Oracle 8.0.5). The results of the study are described in detail in [14]. Here we concentrate on the aspects relevant to our discussion.

2.1 SQL Servers Cannot Be Assumed to 'Fail-Stop'

Table 1 summarises the results of the study. The bugs are classified according to the characteristics of the failures they cause, as different failure types require different recovery mechanisms:
Engine Crash failures: crashes or halts of the core engine.
Incorrect Result failures: not engine crashes, but incorrect outputs: the outputs do not conform to the server's specification or to the SQL standard.
Performance failures: the output is correct, but observed to carry an unacceptable time penalty for the particular input.
Other failures.

Table 1. A summary of the study with reported bugs for 4 SQL servers. The first 6 rows represent the observations after running the bug scripts. Each shaded column represents the results of running bug scripts on the server for which the bugs were reported, while the non-shaded columns represent the results of running the scripts on the other three servers. The last 6 rows represent a classification of the observed failures.

			Interbase	PostgreSQL	Oracle	MSSQL	PostgreSQL	Interbase	Oracle	MSSQL	Oracle	Interbase	MSSQL	PostgreSQL	MSSQL	Interbase	Oracle	PostgreSQL
Total Scripts			55	55	55	55	57	57	57	57	18	18	18	18	51	51	51	51
Script cannot be run (Functionality Missing)			n/a	23	20	16	n/a	32	27	24	n/a	13	13	12	n/a	36	32	31
Further Work			n/a	5	4	6	n/a	2	0	0	n/a	1	1	2	n/a	3	7	2
Total scripts run			55	27	31	33	57	23	30	33	18	4	4	4	51	12	12	18
No failure observed			8	26	31	31	5	23	30	31	4	4	4	3	12	11	12	12
Failure observed			47	**1**	0	**2**	52	0	0	**2**	14	0	0	**1**	39	**1**	0	**6**
Types of failures	**Poor Performance**		3	0	0	0	0	0	0	0	1	0	0	0	6	0	0	0
	Engine Crash		7	0	0	0	11	0	0	0	3	0	0	0	5	0	0	0
	Incorrect Result	**Self-evident**	4	0	0	**1**	14	0	0	**1**	3	0	0	0	10	0	0	**6**
		Non -self-evident	23	**1**	0	**1**	20	0	0	**1**	7	0	0	**1**	17	**1**	0	0
	Other	**Self-evident**	2	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0
		Non -self-evident	8	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0

[4] Made available as an open-source product under this name by Borland Inc. in 2000. The company reverted to closed development for subsequent releases. The product continues to be maintained as an open source development under a different name - "Firebird".

We also classified the failures according to their detectability by a client of the database servers:
Self-Evident failures: engine crash failures, cases in which the server signals an internal failure as an exception (error message) and performance failures.
Non-Self-Evident failures: incorrect result failures, without server exceptions within an accepted time delay.
[14] shows that the fraction of reported faults causing crash failures varies across servers from 13% (MS SQL) to 21% (Oracle and PostgreSQL). These are small percentages, despite crashes being *easy to detect* and thus likely to get reported [14]. More than 50% of the faults cause failures with incorrect but seemingly legal results, i.e. a client application will not normally detect them. In other words, an assumption that either a server will process a query correctly or the problem will be detected is *flatly wrong*. Any replication scheme that tolerates server crashes only does not provide any guarantee against these failures – the incorrect results may be simply replicated. Although our results do not show how likely non-self-evident *failures* are - the percentages above are based on *fault* counts - the evidence in [14] seems overwhelming against assuming (until actual failure counts are available) that 'fail-stop' failures are the main concern to be resolved by replication.

2.2 Potential of Design Diversity for Detecting/Diagnosing Failures

Table 2 gives another view on the reported bugs of the 4 SQL servers: what would happen if 1-out-of-2 fault-tolerant SQL servers were built using these 4 SQL servers.

Table 2. Potential of diverse pairs of servers for tolerating the effects of the reported bugs in our sample. *IB* stands for Interbase, *PG* for PostgreSQL, *OR* for Oracle and *MS* for MS SQL

Pairs of servers	Number of bug scripts run	Failure Observed (in at least one server)	One out of two servers failing		Both servers failing		
			Self-evident	Non - Self-evident	Non – Detectable	Detectable	
						Self-evident	Non – Self-evident
IB + PG	62	43	17	25	1	0	0
IB + OR	62	29	8	21	0	0	0
IB + MS	69	35	11	21	2	1	0
PG + OR	64	30	13	16	0	0	1
PG + MS	76	46	18	21	1	6	0
OR + MS	71	14	7	7	0	0	0

What we want to find out is how many of the coincident failures are *detectable* in the 2-version systems. We define:
Detectable failures: Self-Evident failures or those where servers return different incorrect results (the comparison algorithm must be written to allow for possible differences in the representation of correct results). All failures affecting only one out of two (or up to n-1 out of n) versions are detectable.
Non-Detectable failures: the two (or more) servers return identical incorrect results.
Replication with identical servers would only detect the self-evident failures: crash failures, failures reported by the server itself and poor performance failures. For all

four servers, less than 50% of faults cause such failures. Instead, with diverse pairs of servers many of the failures are detectable. All the possible two-version fault-tolerant configurations detect the failures caused by at least 94% of the faults.

3 Architecture of a Fault-Tolerant Diverse SQL Server

3.1 General Scheme

Studying replication protocols is not the focus of this paper. Data replication is a well-understood subject [10]. A recent study compared various replication protocols in terms of their performance and the feasibility of their implementation [18]. One of the oldest replication protocols, 'Read once write all available (ROWAA)' [10] comes out as the best protocol for a very wide range of scenarios. In ROWAA, read operations are on just one copy of the database (e.g. the one that is physically nearest to the client) while write operations must be replicated on all nodes. An important performance optimisation for the updates is executing the update statements only once and propagating the updates to the other nodes [10]. This may lead to a very significant improvement; with up to a fivefold reduction in execution time of the update statements [19], [20]. However, these schemes would not tolerate non-self-evident failures that cause incorrect updates or return incorrect results by select queries. For the former, incorrect updates would be propagated to the other replicas and for the latter, incorrect results would be returned to the client. This deficiency can be overcome by building a fault-tolerant server node ("FT-node") from two or more diverse SQL servers, wrapped together with a "middleware" layer to appear to each client as a single SQL server and to each of the SQL servers as a set of clients, as shown in Fig. 1.

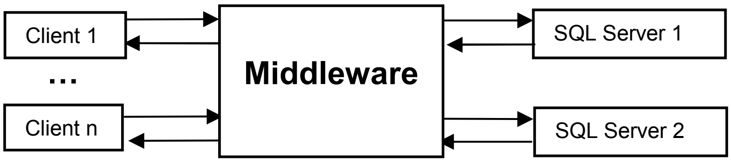

Fig. 1. Fault-tolerant server node (FT-node) with two or more diverse SQL servers (in this case two: *SQL Server 1 and SQL Server 2*). The *middleware* "hides" the servers from the clients (*1 to n*) for which the data storage appears as a single SQL server

Some design considerations about this architecture follow.

The middleware must ensure connectivity with the clients and the multiple servers. The connectivity between the clients and the middleware can implement a "standard" API, e.g. JDBC/ODBC, or some proprietary API. The middleware communicates with the servers using any one of the connectivity solutions available for the chosen servers (with server independent API, e.g. JDBC/ODBC, or the server proprietary API).

The rest of Section 3 deals with other design issues in this fault-tolerant design:

- synchronisation between the servers to guarantee data consistency between them;
- support for fault-tolerance for realistic modes of failure via mechanisms for:
  - error detection;
  - error containment;
  - state recovery
- "replica determinism": dealing with aspects of server behaviour which would cause inconsistencies between database replicas even with identical sequences of queries;
- translation of the SQL queries coming from the client to be "understood" by diverse SQL servers which use different "dialects" of the SQL syntax;
- "data diversity": the potential for improving fault tolerance through expressing (sequences of) client queries in alternative, logically equivalent ways;
- performance effects of diversity, which depending on the details of the chosen fault-tolerance scheme may be negative or positive.

3.2 Fault Tolerance Strategies

This basic architecture can be used for various forms of fault-tolerance, with different trade-offs between degree of replication, fault tolerance and performance [21].
We can discuss separately various aspects of fault tolerance:

- *Failure detection and containment.* Self-evident server failures are detected as in a non-diverse server, via server error messages (i.e. via the existing error detection mechanisms inside the servers), and time-outs for crash and performance failures. Diversity gives the additional capability of detecting non-self-evident failures by comparing the outputs of the different servers. In a FT-node with 3 or more diverse versions, majority voting can be used to choose a result and thus mask the failure to the clients, and identify the failed version which may need a recovery action to correct its state. With a 2-diverse FT-node, if the two servers give different results, the middleware cannot decide which server is in error: it needs to invoke some form of manual or automated recovery. The middleware will present the failure to the client as a delay in response (due to the time needed for recovery), or as a self-evident failure (crash - a "fail-silent" FT-node; or an error message - a "self-checking" FT-node). The voting/comparison algorithm will need to allow for "cosmetic" differences between equivalent correct results, like padding characters in character strings or different numbers of digits in the representations of floating point numbers.
- *Error recovery.* As just described, diversity allows for more refined diagnosis (identification of the failed server). This improves availability: the middleware can selectively direct recovery actions at the server diagnosed as having failed, while letting the other server(s) continue to provide the service. State recovery of the database can be obtained in the following ways:
  - via standard backward error recovery, which will be effective if the failures are due to Heisenbugs. To command backward error recovery, the middleware may use the standard database transaction mechanisms: aborting the failed transaction and replaying its queries may produce a correct execution. Alternatively or additionally, checkpointing [22] can be used. At regular intervals, the states of the servers are saved (by database "backup" commands: e.g., in PostgreSQL the pg_dump command). After a failure, the

    database is restored to the state before the last checkpoint and the sequence of (all or just update) queries since then is replayed to it;
  - additionally, diversity offers ways of recovering from Bohrbug-caused failures, by essentially copying the database state of a correct server into the failed one (similarly to [23]). Since the formats of the database files differ between the servers, the middleware would need to query the correct server[s] for their database contents and command the failed server to write them into the corresponding records in its database, similar to what is proposed in [11]. This would be expensive, perhaps to be completed off-line, but a designer can use multi-level recovery, in which the first step is to correct only those records that have been found erroneous on read queries.

To increase the level of data replication a possibility is to integrate our FT-node scheme with standard forms of replication, like ROWAA, possibly with the optimisation of writes [10]. One could integrate these strategies into our proposed middleware, or for simplicity choose a layered implementation (possibly at a cost in terms of performance) in which our fault-tolerant nodes are used as server nodes in a standard ROWAA protocol. However, a layered architecture using, say, 2-diverse FT-nodes may require more servers for tolerating a given number of server failures.

3.3 Data Consistency between Diverse SQL Servers

Data consistency in database replication is usually defined in terms of 1-copy serialisability between the transaction histories executed on the various nodes [10]. In practical implementations this is affected by:

- the order of delivery of queries to the replicas
- the order in which the servers execute the queries, which in turn is affected by:
  - the execution plans created for the queries
  - the execution of the plans by the execution engines of the servers, which are normally non-deterministic and may differ between the servers, in particular with the concurrency control mechanism implemented.

Normally, consistency relies on “totally ordered” [24] delivery of the queries by reliable multicast protocols. For the optimised schemes of data replication, e.g. ROWAA, only the updates are delivered in total order to all the nodes. Diverse data replication would also rely on the total ordering of messages.

In terms of execution of the queries the difference between non-diverse and diverse replication is in the execution plans, which will be the same for replicas of the same SQL server, but may differ significantly between diverse SQL servers. This may result in significantly different times to process the queries. If many queries are executed concurrently, identical execution plans across replicas do not guarantee the same order of execution, due to for example multithreading. The allocation of CPU time to threads is inherently non-deterministic. In other words, non-determinism must be dealt with in both non-diverse and diverse replication schemes. The phenomenon of inconsistent behaviour between replicas that receive equivalent (from some viewpoint) sequences of requests is not limited to database servers [25] and there are well known architectural solutions for dealing with it [26]. Empirically [27], we repeatedly observed data inconsistency even with replication of the same SQL server.

To achieve data consistency, i.e. a 1-copy serialisable history [10] across replicas, the concurrent execution of modifying transactions needs to be restricted. Two extreme possible scenarios can be exploited to deal with non-determinism in SQL servers, and apply to both non-diverse and diverse SQL servers:

- non-determinism does not affect the combined result of executing concurrent transactions: for instance, the transactions do not "clash". No concurrent transactions attempt modifications of the same data. If this is the case, all possible sub-histories, which may result from various orders of executing the transactions concurrently, are identical and thus 1-copy serialisability across all the replicas (no matter whether diverse or non-diverse) will be guaranteed despite the possibly different orders of execution of the transactions by the different servers;
- non-determinism is eliminated with respect to the modifying transactions by executing them one at a time. Again, 1-copy serialisability is achieved [27]. This regime of serialisability may be limited to within each individual database, thus allowing concurrency between modifying transactions executed on different databases.

Combinations of these two are possible: concurrent transactions are allowed to execute concurrently, but if a "clash" is detected, all transactions involved in the clash are rolled back and then serialised according to some total order [24].

3.4 Differences in Features and SQL "Dialects" between SQL Servers

3.4.1 Missing and Proprietary Features

With two SQL standards (SQL-92 and SQL-99 (SQL 3)) and several different levels of compliance to these, it is not surprising that SQL servers implement many different variants of SQL. Most of the servers with significant user bases guarantee SQL-92 Entry Level of compliance or higher. SQL-92 Entry Level covers the basic types of queries and allows in many cases the developers to write code which requires no modification when ported to a different SQL server. However some very widely used queries are not part of the Entry Level, e.g. the various built-in JOIN operators [28]. Triggers and stored procedures [29] are another example of very useful functionality, used in many business databases, which are not part of SQL-92 (surprisingly they are not yet supported in MySQL, one of the most widely used SQL servers).

In addition vendors may introduce proprietary extensions in their products. For example Microsoft intends to incorporate .NET in "Yukon", their new SQL server [30].

3.4.2 Differences in Dialects for Common Features

In addition to the missing and proprietary features, there are differences even in the dialect of the SQL that is common among servers. For instance the example below shows differences in the syntax for outer joins between the SQL dialects of three servers which we used in experiments with diverse SQL servers [27] (Oracle uses a non-standard syntax for outer joins):

ORACLE 8.0.5

```
select items.number
   from items, orders
   where items.number = orders.item_number (+)
   group by items.number
   having items.number < 20000
   order by items.number desc
```

MS SQL 7.0 and INTERBASE 6.0

```
select items.number
   from items
   left outer join orders on items.number =
   orders.item_number
   group by items.number
   having items.number < 20000
   order by items.number desc
```

Although the difference in the syntax is marginal, Oracle 8.0.5 will not parse the standard syntax. Significant differences exist between the syntax of other SQL constructs, e.g. stored procedures and triggers. For instance, Oracle's support for SQLJ for stored procedures differs slightly from the standard syntax.

3.4.3 Reconciling the Differences between Dialects and Features of SQL Servers

Standardisation is unlikely to resolve the existing differences between the SQL dialects in the foreseeable future, although there have been attempts to improve interoperability by standardising "persistent modules" [29] (also called "stored procedures" in most major SQL servers or "functions" in PostgreSQL). However, some vendors still undermine standardisation by adding proprietary extensions in their products.

To use replication with diverse SQL servers, the differences between the servers must be reconciled. Two possibilities are:

- requiring the client applications to use the SQL sub-set which is common to all the SQL servers in the FT-node, and reconciling the differences between the dialects by implementing "translators" that translate the syntax used by the client applications to the syntax understood by the respective servers. Such "translators" can become part of the replication middleware (Fig 1). One may:
  - require the client applications to use ANSI SQL to work with the middleware, which will contain translators for all SQL dialects used in the FT-node;
  - allow the clients to use the SQL dialect of their choice (e.g. the dialect of a specific SQL server or ANSI SQL), to allow legacy applications written for a specific SQL server to be "ported" and run with diverse replication.
- expressing some of the missing SQL features through equivalent transformation of the client query to query(ies) supported by the SQL servers used in the FT-node (see 3.6).

In either case, translation between the dialects of the SQL servers is needed. Translation is certainly feasible. Surprisingly, though, we could not find off-the-shelf tools to assist with the translation even though "porting" database schema from one SQL server product to another is a common practice.

3.5 Replica Determinism: The Example of DDL Support

The differences between SQL servers also affect the Data Definition Language (DDL), i.e., the part of SQL that deals with the metadata (schema) of a database. The DDL does not require special attention with non-diverse replication: the same DDL statement is just copied to all replicas. We outline here an aspect of using DDL which may lead to data inconsistency: *auto numeric fields*.

SQL servers allow the clients to simplify the generation of unique numeric values by defining a data type, which is under the direct control of the server. These unique values are typically used for generating keys (primary and secondary) without too much overhead on the client side: the client does not need to explicitly provide values for these fields when inserting a new record. Implementations of this feature differ between servers (Identity() function in MS SQL, generators in Interbase, etc.), but this is not a serious problem. The real problem is that the different servers specify different behaviours of this feature when a transaction is aborted within which unique numbers were generated. In some servers, the values generated in a transaction that was rolled back are "lost" and will never appear in the fields of committed data. Other servers keep track of these "unused" values and generate them again in some later transactions, which will be committed. This difference affects data consistency across different SQL servers. The inconsistencies thus created must be handled explicitly, by the middleware [27], or by the client applications by not using auto fields at all.

This is just one case of diversity causing violations of *replica determinism* [31]; others may exist, depending on the specific combination of diverse servers.

3.6 Data Diversity

Although diversity can dramatically improve error detection rates it does not make them 100%, e.g. our study found four bugs causing identical non-self-evident failures in two servers.

To improve the situation, one could use the mechanism called "data diversity" by Ammann and Knight [32] (who studied it in a different context). The simplest example of the idea in [32] would refer to computation of a continuous function of a continuous parameter. The values of the function computed for two close values of the parameter are also close to each other. Thus, failures in the form of dramatic jumps of the function on close values of the parameter can not only be detected but also corrected by computing a "pseudo correct" value. This is done by trying slightly different values of the parameter until a value of the function is calculated which is close to the one before the failure. This was found [32] to be an effective way of masking failures, i.e. delivering fault-tolerance. Data diversity thus can help not only with error detection but with recovery as well, and thus to tolerate some failures due to design faults without the cost of design diversity.

Data diversity seems applicable to SQL servers because most queries can be "re-phrased" into different, but logically equivalent [sequences of] queries. There are cases where a particular query causes a failure in a server but a *re-phrased* version of the same query does not. Examples of such queries often appear in bug reports as “*workarounds*”. The example below is a bug script for PostgreSQL v7.0.0, producing a non-self-evident failure (incorrect result) by returning one row instead of six.

```
create table employee (name varchar(10) not null, age integer,
salary float, deptname varchar(10), manager varchar(10), primary
key(name));
The following data exists in the table:
```

Name	Age	Salary	Deptname	Manager
Mike	28	1500.00	Shoe	Edna
Sally	42	877.50	Toy	Ted
Georgia	22		Book	
Ted		2615.73	Toy	Malcolm
Edna	39	2000.00	Shoe	Malcolm
Malcolm	50	2750.00	Admin	

```
CREATE VIEW avg_int AS SELECT AVG(salary) AS avg_sal FROM
employee;
CREATE VIEW average AS SELECT employee.name, employee.salary,
avg_int.avg_sal, (salary-avg_sal) as sal_diff FROM employee,
avg_int;

SELECT * FROM average;

name  | salary | avg_sal  | sal_diff
------+--------+----------+----------
 Mike |   1500 | 1948.646 | -448.646
```

A workaround exists which is based on using a TEMP (temporary) table instead of a view (in this case to hold the average salaries). The same table schema definition and data given above are used together with the code below, and then the result is correct.

```
/* This is the temporary table*/
SELECT AVG(salary) AS avg_sal INTO TEMP TABLE avg_int FROM
employee;

/* This view is same as above. */
CREATE VIEW average AS SELECT employee.name, employee.salary,
avg_int.avg_sal, (salary-avg_sal) as sal_diff FROM employee,
avg_int;

SELECT * FROM average;
name    | salary  | avg_sal  | sal_diff
--------+---------+----------+-----------
Mike    |    1500 | 1948.646 |  -448.646
Sally   |   877.5 | 1948.646 | -1071.146
Georgia |         | 1948.646 |
Ted     | 2615.73 | 1948.646 |   667.084
Edna    |    2000 | 1948.646 |    51.354
Malcolm |    2750 | 1948.646 |   801.354
                                 (6 rows)
```

Data diversity could be implemented via an algorithm in the middleware that rephrases queries according to predefined rules. For instance, one such rule could be to break-up all complex nested SELECT queries so that the inner part of the query is saved in a temporary table, and the outer part then uses the temporary table to generate the final result.[5]

Data diversity can be used *with* or *without* design diversity. In the case of databases it would be attractive alone as it would for instance allow applications to use the full set of features of an SQL server, including the proprietary ones. Architectural schemes using data diversity are similar to those using design diversity. For instance, Amman and Knight in [32] describe two schemes, which they call “retry block” and “n-copy programming”, which can also be used for SQL servers. The “retry block” is based on backward recovery. A query is only re-phrased if either the server “fail-stops” or its output fails an acceptance test. In “n-copy programming”, a copy of the query as issued by the client is sent to one of the servers and re-phrased variant[s] are sent to the others; their results are voted to mask failures. The techniques for error detection and state recovery would also be similar to the design diversity case (Section 3.2). In the “retry block” scheme (backward error recovery), applied to one of the servers, a failed transaction would be rolled back, and the rephrased queries executed from the rolled-back state thus obtained. In the “n-copy programming” scheme, the state of a server diagnosed to be correct would be copied to the faulty server (forward error recovery). Another possibility is not to use “re-phrasing” unless diverse replicas produce different outputs with no majority. Then, the middleware could abort the transaction and replay the queries, after “re-phrasing” them, to all or some of the servers. Fig. 2 shows, at a high level, an example of architecture using both data diversity and design diversity with SQL servers. This example assumes a combination of “N-version programming” and “n-copy programming”, with a single voter in the middleware.

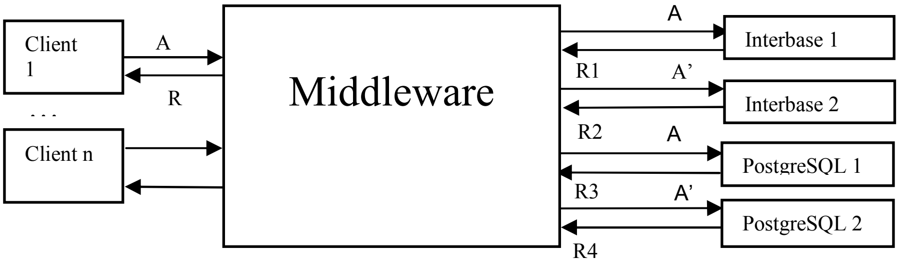

Fig. 2. A possible design for a fault-tolerant server using diverse SQL servers and data diversity. The original query (*A)* is sent to the pair {*Interbase 1*, *PostgreSQL 1*}, the re-phrased query (*A'*) is sent to the pair {*Interbase 2*, *PostgreSQL 2*}. The *middleware* compares/votes the results in one of the ways described in Section 3.2 for solutions without data diversity

A designer would choose a combination of design diversity and data diversity as a trade-off between the conflicting requirements of dependability, performance and

[5] Re-phrasing algorithms can also be part of the translators for the different SQL dialects. A complex statement which can be directly executed with some servers but not others may need to be re-phrased as a logically equivalent sequence of simpler statements for the latter.

cost. At one extreme, combining both design and data diversity and re-phrasing all those queries for which re-phrasing is possible would give the maximum potential for failure detection, but with high cost.

3.7 Performance of Diverse-Replicated SQL Servers

Database replication with diverse SQL servers improves dependability, as discussed in the previous sections. What are its implications for system performance? In Fig. 3 we sketch a timing diagram of the sequence of events associated with a query being processed by an FT-node which includes two diverse SQL servers.

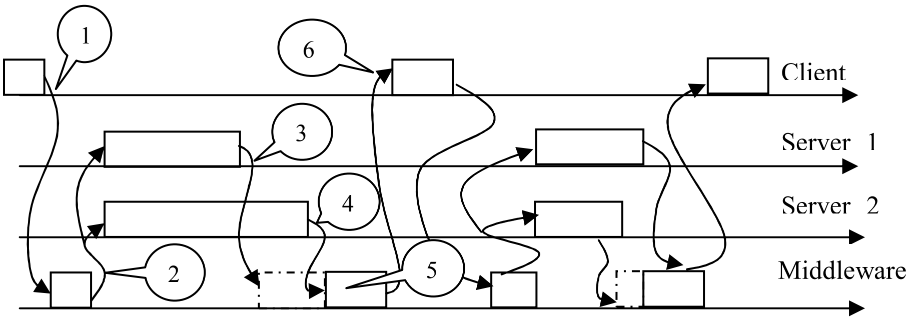

Fig. 3. Timing diagram with two diverse servers and middleware running in pessimistic regime. The meaning of the *arrows* is: *1* – the client sends a query to the middleware; *2* – the middleware translates the request to the dialects of the servers and sends the resulting queries, or sequences of queries, to the respective servers; *3* – the faster response is received by the middleware; *4* – the slower response is received by the middleware; *5* – the middleware adjudicates the two responses; *6* – the middleware sends the result back to the client or if none exists initiates recovery or signals a failure

Processing every query will involve some synchronisation overhead. To “validate” the results of executing each query, the middleware should wait for responses from both servers, check if the two responses are identical and, in case they differ, initiate recovery. We will use the term “pessimistic” for this regime of operation. If the response times are close, the overhead due to differences in the performance of the servers (shown in the diagram as dashed boxes) will be low. If the difference is significant, then this overhead may become significant. If one of the servers is the *slower one on all queries*, this slower server dictates the pace of processing. The service offered by the FT node will be as fast as the service from a non-replicated node implemented with the slower server, provided the extra overhead due to the middleware is negligible compared to the processing time of the slower server. If, however, the slower response may come from either server, the service provided by the FT-node will be slower than if a non-replicated node with the slower server was used. This slow-down due to the pessimistic regime is the cost of the extra dependability assurance.

Many see performance (e.g. the server’s response time) as the most important non-functional requirement of SQL servers. Is diversity always a bad news for those for whom performance is more important than dependability? Fig. 4 depicts a scenario, referred to as the “optimistic” regime. For this regime the only function of the middleware is to translate the client requests, send them to the servers and as soon as the first response is received, return it back to the client.

Therefore, if the client is prepared to accept a higher risk of incorrect responses diversity can, in principle, improve performance compared with non-diverse solutions.

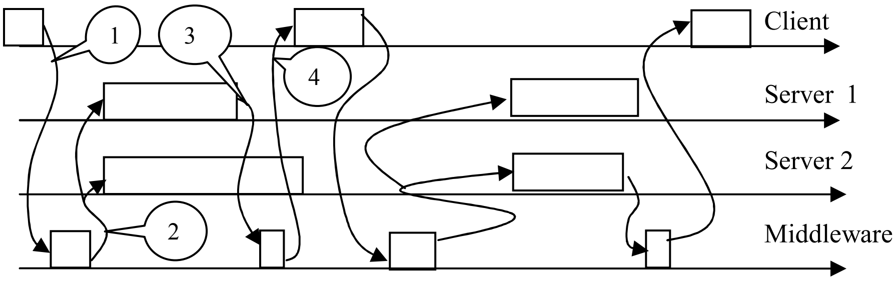

Fig. 4. Timing diagram with two diverse servers and middleware running in optimistic regime. The meaning of the *arrows* is: *1* – the client sends a query to the middleware, *2* – the middleware translates the request to the dialects of the servers and sends the resulting queries, or sequences of queries, to the respective servers; *3* – the fastest response is received by the middleware; 4 - the middleware sends the response to the client

How does the optimistic regime compare in terms of performance (e.g. response time) with the two diverse servers used? If one of the servers is faster on *every* query, diversity with the optimistic regime does not provide any improvement compared with the faster server. If, however, the faster response comes from different servers depending on the query, then the optimistic regime will give a faster service than the faster of the two servers (provided the overhead of the middleware is not too high compared with the response times of the servers).

The faster response for a query may come from either server (as shown in Fig. 4). A similar effect is observed when accepting the faster response between those of two or more *identical* servers. Similarly, in mirrored disk configurations one can take advantage of the random difference between the physical disks' response times to reduce the average response time on reads [33]. What changes with diverse servers is that they may *systematically differ* in their response times for different types of transactions/queries, yielding a greater performance gain. The next section shows experimental evidence of this effect.

4 Increasing Performance via Diversity

4.1 Performance Measures of Diverse SQL Servers

We conducted an empirical study to assess the performance effects of the pessimistic and optimistic regimes using two open-source SQL servers, PostgreSQL 7.2.4 and Interbase 6.0 (licenses for commercial SQL servers constrain the users' rights to publish performance related results).

For this study, we used a client implementing the TPC-C industry-standard benchmark for on-line transaction processing [34]. TPC-C defines 5 types of transactions: *New-Order, Payment*, *Order-Status, Delivery* and *Stock-Level* and sets the probability of execution of each. The specified measure of throughput is the number of *New-Order* transactions completed per minute (while all five types of transactions are executing). The benchmark provides for performance comparisons of SQL servers from different vendors, with different hardware configurations and operating systems.

We used several identical machines with different operating systems: Intel Pentium 4 (1.4 GHz), 640MB RAMBUS RAM, Microsoft Windows 2000 Professional for the client(s) and the Interbase servers, Linux Red Hat 6.0 for the PostgreSQL servers. The servers ran on four machines: 2 replicas of Interbase and two replicas of PostgreSQL. Before the measurement sessions, the databases on all four servers were populated as specified by the standard.

The client, implemented in Java, used JDBC drivers to connect to the servers. We ran two experiments with different loads on the servers:

Experiment 1: A single TPC-C client for each server;

Experiment 2: 10 TPC-C clients for each server, each client using one of 10 TPC-C databases managed by the same server, so that we could measure the servers' performance under increased load while preserving 1-copy serialisability.

Our objective of the study was not just to repeat the benchmark tests for these servers, but also to get preliminary indications about the performance of an FT-node using diverse servers, compared to one using identical servers and to a single server. Our measurements were more detailed than the ones required by the TPC-C standard. We recorded the response times for each individual transaction, for each server. We were specifically interested in comparing two architectures:

- two *diverse servers* concurrently process the same stream of transactions (Fig. 1) translated into their respective SQL dialects: the smallest possible configuration with diverse redundancy.
- a reference, non-diverse architecture in which two *identical servers* concurrently process the same stream of transactions.

All four servers were run concurrently, receiving the same stream of transactions from the test harness, which produced four copies of each transaction/query. The overhead that the test harness introduces (mainly due to using multi-threading for communication with the different SQL servers) is the same with and without design diversity.

Instead of translating the queries into the SQL dialects of the two servers on the fly, the queries were hard-coded in the test harness. The comparison between the two architectures is based on the *transaction response times*, neglecting all extra overheads that the FT-node's middleware would introduce. This simplification may somewhat distort the results, but also allows us to compare the potential of the two architectures, and to look at possible trade-offs between dependability and performance, without the effects of the detailed implementation of the middleware.

We compare the performance of the two servers with each other and with the two regimes, pessimistic (Fig. 3) and optimistic (Fig. 4). The performance measure we calculated for the pessimistic regime represents the upper bound of the response time for this particular mix of transactions while performance measure for the optimistic regime represents the lower bound.

We used the following measures of interest:

- mean transaction response times for all five transaction types (Fig. 5)
- mean response times per transaction of each type (Fig. 6).

With *two identical SQL servers* (last two server pairs in Fig. 5), the difference between the mean times is minimal, within 10%. The mean times under the optimistic and pessimistic regimes of operation remain very close (differences of <10% for Interbase and <15% for PostgreSQL). Interbase is the faster server, being almost twice as fast as PostgreSQL, for this set of transactions.

When we combine *two diverse SQL servers* we get a very different picture. Now the optimistic regime can deliver dramatically better performance than the faster server, Interbase. The mean response time is almost 3 times shorter than for Interbase alone (compare the first two bars for the first four pairs). When the pessimistic regime is used, the value of the mean response time is larger than the respective value of the slower server, PostgreSQL, but the slow down is within 40% of PostgreSQL's mean response time - the cost of the improved dependability assurance.

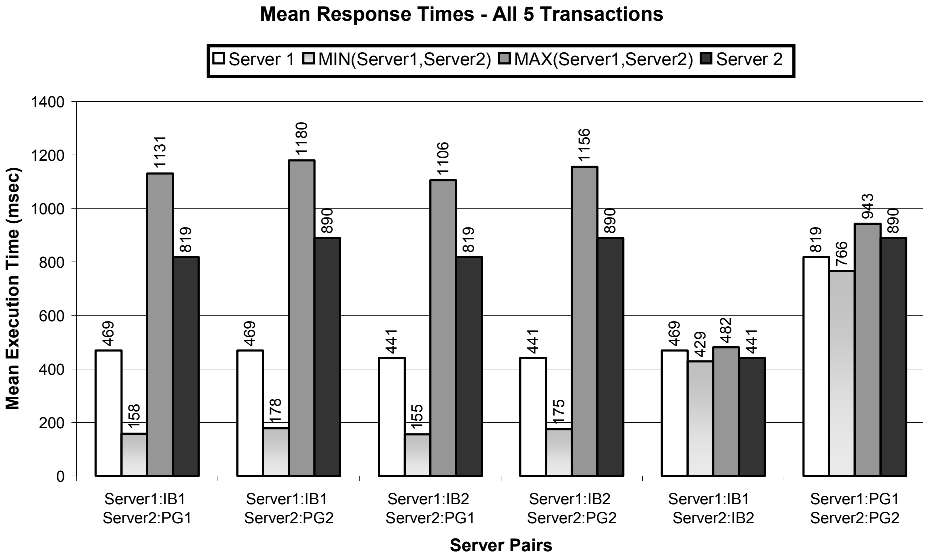

Fig. 5. Mean response time for all five transaction types over 10,000 transactions for two replicas of Interbase 6.0 and two of PostgreSQL 7.2.4. The X-axis lists the servers grouped as pairs (*Server 1 and Server 2*). Each server may be of type Interbase (*IB*) or PostgreSQL (*PG*). For each of the 6 server pairs the vertical bars show: – the mean response times of the individual servers and the mean response times calculated for the two regimes of operation of an FT-node (optimistic and pessimistic)

In order to understand why a diverse pair is so different from a non-diverse pair we looked at the individual transaction types. The mean response times of the five transaction types individually are shown in Fig. 6. The figure indicates that the servers "complement" each other in the sense that when Interbase is slow (on average) to process one type of transaction PostgreSQL is fast (*New-Order* and *Stock-Level*) and vice versa (*Payment*, *Order-Status* and *Delivery*). This illustrates why a diverse pair outperforms a non-diverse one so much when the optimistic regime is used, and why it is worse than the slower server when the pessimistic regime is used (Fig. 5).

In addition to the mean execution times, we have calculated the percentage of the faster responses coming from either Interbase or PostgreSQL for each transaction. For three transaction types the situation is clear-cut. Interbase is always the faster server for *Order-Status* and *Delivery* transactions, while PostgreSQL is always the faster for *Stock-Level* transactions. For *New-Order* and *Payment* transactions instead, the server that is faster on average does not provide the faster response for each individual transaction. Consider the pair {IB1, PG1}. For *New-Order* transaction, PG1 is faster than IB1 on 81.2% of the transactions but slower on 15.6% (3.2% of the response times were equal). The situation is reversed for *Payment* transactions: 77.2% of the faster responses come from IB1, 15.3% from PG1. This fluctuation is further revealed in Fig. 7. Both observations confirm that diverse servers under the optimistic regime would have performed better (for this transaction mix and load) than a pair of identical servers.

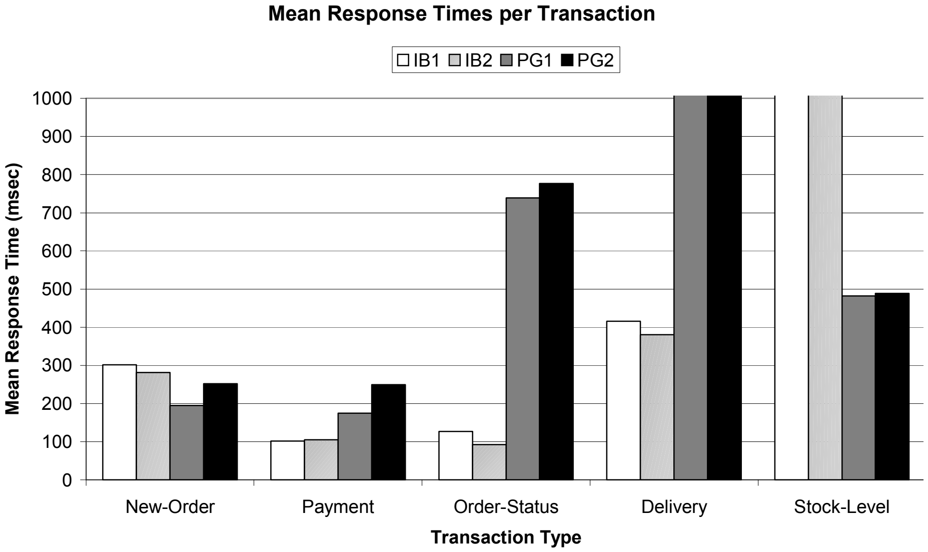

Fig. 6. Mean response times by two replicas of Interbase 6.0 and PostgreSQL 7.2.4 for all five transactions. The X-axis lists the transaction types (*New-Order, Payment, Order-Status, Delivery and Stock-Level*). The Y-axis gives the values of the mean response time in milliseconds for each of the servers (*IB1, IB2, PG1 and PG2*) for a particular transaction type

This pattern of the two SQL servers "complementing" each other was also observed in *Experiment 2* under increased load with 10 TPC-C clients. During this experiment the servers were "stretched" so much that the virtual memories of the machines were exhausted. Similarly to the observations of *Experiment 1*, when *two identical servers* are used the difference between the mean response times is minimal, within 10%, and

the difference between the mean response times of the optimistic and pessimistic regime remain less than 10% for both servers. Again Interbase is the faster server.
The mean response times when *two diverse servers* are considered under the optimistic regime are around four times shorter than for Interbase alone. Under the pessimistic regime, the mean response time is of course larger than the value of the slower server (on average), PostgreSQL, but the slow down is within 60% of PostgreSQL's mean response time (it was 40% in *Experiment 1*, when a single client was used).

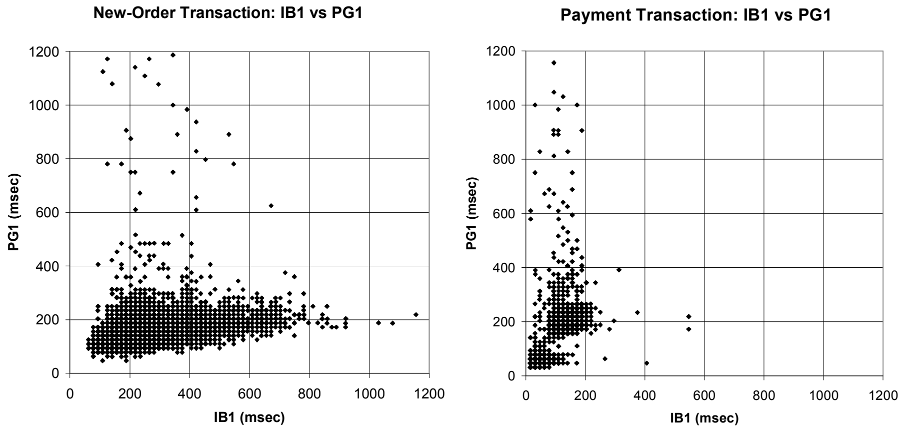

Fig. 7. Response times for the *New-Order* and *Payment* transactions. Every *dot* in the plots represents the response times of two servers for an *instance* of the respective transaction type. If the times were close to each other most of the dots would be concentrated around the unit slope (observed for the pairs of identical servers, IB1 vs IB2 and PG1 vs PG2). If the dots are mostly below the slope, Interbase is slower (as with the *New-Order*). If the dots are concentrated above the unit slope – PostgreSQL is slower (as with the *Payment*). Similar results were obtained for the other three diverse server pairs

4.2 Design Solutions for the Optimistic Regime

Under the optimistic regime, diversity offers better performance than each of the diverse SQL servers used. Various design solutions are possible, with different trade-offs between dependability and performance. We discuss two in more detail, for an FT-node with two or more servers:
Non fault-tolerant solution: For each query, the middleware forwards the first response to the client and discards all later responses. The performance gain depends on whether, by the time the middleware relays a query to the servers, all servers have finished processing the previous query.[6] If the slowest server is still processing the previous query, there are two options:

[6] This happens if the sum of the transport delay to deliver the fastest response to the client, the client's own processing time to produce the next query, and the transport delay to deliver the next query to the middleware is longer than the extra time needed by the slower server to

- the middleware waits until the slowest server completes (aborting the query is not an option because it will compromise data consistency); this delay may seriously limit the performance gain given by the optimistic regime;
- the middleware forwards each query, of a transaction, immediately to those servers that are done processing the previous one, but buffers it for servers that are not. If the middleware only behaves like this within transactions, while on commits of transactions it, inevitably, waits for the slowest server, 1-copy serialisability is preserved.

The transport delays and the client's own processing delays are the two key factors, which decide how much time will be gained using the optimistic regime. The transport delays are implementation-specific and likely to be significant in multi-tier systems. Similarly, the client's own delay is application specific. For interactive applications, it is very likely to be significant.

Fault-tolerant solution: The middleware optimistically forwards the first response to the client, and keeps a copy to compare with later responses when they arrive. If they differ, it initiates recovery. This is easily accomplished within a transaction: the transaction is rolled back, and the client is notified just as for any other transaction rollback decided by a server. This optimistic fault-tolerant scheme will be almost as fast as discarding the late responses, except in the presumably rare case of discrepancy between the servers' responses. The previous considerations about the impact of transport delays and of the client's processing delays still apply.

5 Related Work

Replicated databases are common, but most designs are not suitable for diverse redundancy. We have referred in the previous section to some of the standard solutions [10], [11], [19], [22] and [20].

Recent surveys exist of the mechanisms for eager replication of databases [35], and for the replication mechanisms (mainly lazy replication) implemented in various SQL servers [36]. The Pronto protocol [37] attempts to reduce the negative effects of lazy replication using ideas typical for eager replication. One of its selling points is that it can be used with off-the-shelf SQL servers, but it is unclear whether this includes diverse servers. A potential problem is the need to broadcast the SQL statement from the primary to the replicas. The syntax of SQL statements varies between SQL servers, as discussed in Section 3.

A relevant discussion of the various ways of implementing database replication with off-the-shelf SQL servers is in [38]. Three forms are discussed, treating the SQL servers as black, white or grey boxes. All commercial vendors of SQL servers use the white-box approach, where a suite necessary for replication is added to the code of the non-replicated server. The black-box and the grey-box approaches are implemented in the form of middleware on top of the existing SQL servers. The black-box approach, like the design solutions discussed here, uses the standard interfaces of the servers and its main advantage is applicability to a wide range of servers. The grey-box approach,

complete query processing. In this case, both (or all) servers will be ready to take the next query and the race between them will start over.

implemented in [39] and [40], assumes that the servers provide services specifically to assist replication.

Comparisons of various replication protocols from the point of view of their performance and feasibility are presented in [18], [19].

The problem of on-line recovery is scrutinised in [41] and [24] and cost-effective solutions are proposed.

6 Discussion

The fault diversity figures (presented in Section 2) point to a serious potential gain in reliability from using a fault tolerant SQL server built from two or more off-the-shelf servers. There are limitations to what can be speculated from the bug reports alone, because these do not address the *frequency* of the failures caused. The actual failure reports would be more informative, especially if the vendors used automatic failure reporting mechanisms. An even better analysis could be obtained if these mechanisms gave indications about the users' usage profile as proposed in [42]. However such detailed dependability information is difficult to obtain from the vendors. Based on the evidence of fault diversity presented in Section 2, using a diverse fault-tolerant server would already appear a reasonable and relatively cheap precautionary decision (even without good predictions of its effects) for a user that had: serious concerns about dependability (e.g., interruptions of service or undetected incorrect data being stored are very costly); client applications using mostly the core features common to multiple off-the-shelf products (for instance a user who required portability of applications); modest throughput requirements for database updates which make it easy to accept the synchronisation delays of a fault-tolerant server.

We have provided a more detailed discussion of the fault diversity results in [14].

Data diversity has been proposed as a possibility to detect failures that would otherwise be un-detectable in some diverse server replication settings. We have provided examples of this in Section 3.6. The possible benefits of this approach could be its relatively lower cost (especially if OTS re-phrasing software becomes available) in comparison with design diversity, and also that it can be used with or without design diversity allowing for various cost-dependability trade-offs.

In Section 4 we presented the results from our experiments on the performance of two open-source SQL servers. We estimated the likely performance effect of diversity under optimistic and pessimistic regime of operation.

The *Quality of service* provided by a database server can be defined to include both performance and dependability. Clients with conflicting needs may benefit from design diversity according to their own priorities because an FT-node can apply different regimes for different databases or different clients. When performance is top priority the optimistic regime can be used, possibly even in the non-fault-tolerant variation, which discards the slower responses. In many practical cases this is likely to produce significant improvement. At the other end of the spectrum, when dependability is top priority, the pessimistic regime with a fully featured middleware for fault-tolerance will provide significantly improved dependability assurance. Several intermediate solutions are possible with different trade-offs between performance and dependability. The optimistic regime can be used together with functionality for fault-tolerance using the responses from all servers as discussed in Section 4.2.

7 Conclusions

Most users of SQL servers see performance as the most critical requirement. Dependability, although important, is often assumed not to be a problem, and users who seek to improve it are apparently satisfied with redundant solutions meant to tolerate crash failures only.

We have argued that non-diverse replication is a limited solution, since many server failures are non-self-evident and cannot be tolerated by non-diverse replication. We have shown evidence of this problem from our "fault diversity" measurements. To provide extended protection against non-self-evident failures, we have argued in favour of using diverse SQL servers and outlined a range of possible architectural solutions.

We have presented some encouraging empirical results which suggest that diversity can improve the performance of a fault-tolerant server. To the best of our knowledge, similar results have not been reported before. This possibility is due to the fact that different SQL server may "complement" each other, as we have established empirically for Interbase and PostgreSQL: one of the server is systematically faster in processing some types of transactions while the other server is faster processing other types of transactions. This is similar to the intuitive idea of forming teams of individuals who have different skills, which is an accepted view in various areas. Diversity can improve both aspects of the service provided by the SQL servers, dependability and performance.

We have outlined some design problems in implementing middleware for diverse SQL servers. However, the technical benefits of having such a solution for data replication could be significant. There remain open questions worth studying in the future:

- the work on fault diversity can be extended by finding out whether the same proportion of crash/non-crash failures will be observed with later versions of the servers, or even including other servers e.g. DB2, MySQL, etc.
- evidence of actual failure diversity (or lack thereof) in actual use is also to be sought. We are currently running experiments to assess statistically the actual reliability gains. We have so far run a few million queries on a configuration with three off-the-shelf SQL servers (Interbase, Oracle and MSSQL), with various loads without failures. We plan to continue these experiments for more complete test loads
- demonstrating the feasibility of automatic translation of SQL queries from, say ANSI/ISO SQL syntax to the SQL dialect implemented by the deployed SQL servers.
- empirical evaluation of whether the "optimistic" regime, discussed in Section 4, is practicable for a range of widely used clients;
- implementing configurable middleware, deployable on diverse SQL servers, to allow the clients to request quality of service in line with their specific requirements for performance and dependability, is a possibility for future work

Acknowledgement

This work was supported in part by the Engineering and Physical Sciences Research Council (EPSRC) of the United Kingdom through the Interdisciplinary Research Collaboration in Dependability (DIRC) and the DOTS (Diversity with Off-The-Shelf Components) projects. We wish to thank Peter Bishop for comments on an earlier version of this paper.

References

1. Babbage, C., *On the Mathematical Powers of the Calculating Engine (Unpublished manuscript, December 1837)*, in *The Origins of Digital Computers: Selected Papers*, B. Randell, Editor, 1974, Springer, pp. 17-52.
2. Traverse, P.J., *AIRBUS and ATR System Architecture and Specification*, in *Software diversity in computerized control systems*, U. Voges, Editor, 1988, Springer-Verlag, pp. 95-104.
3. Randell, B. *System Structure for Software Fault-Tolerance*, in *International Conference on Reliable Software, Los Angeles, California, April 1975, (in ACM SIGPLAN Notices, Vol. 10, No. 6, June 1975)*, 1975, pp. 437-449.
4. Lyu, M.R., ed. *Software Fault Tolerance*. Trends in Software Series. 1995, Wiley
5. Avizienis, A. and J.P.J. Kelly, *Fault Tolerance by Design Diversity: Concepts and Experiments,* IEEE Computer, 1984, 17(8): pp. 67-80.
6. Laprie, J.C., et al., *Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures,* IEEE Computer, 1990, 23(7): pp. 39-51.
7. Voges, U., ed. *Software diversity in computerized control systems*. Dependable Computing and Fault-Tolerance series, ed. A. Avizienis, H. Kopetz, and J.C. Laprie. Vol. 2. 1988, Springer-Verlag: Wien.
8. Avizienis, A., et al. *The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software*, in *Proc. of 15th IEEE International Symposium on Fault-Tolerant Computing (FTCS-15)*, 1985, Ann Arbor, Michigan, USA, IEEE Computer Society Press, pp. 126-134.
9. Pullum, L., *Software Fault Tolerance Techniques and Implementation*, 2001, Artech House.
10. Bernstein, P.A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, 1987, Reading, Mass.: Addison-Wesley.
11. Sutter, H., *SQL/Replication Scope and Requirements document*, in *ISO/IEC JTC 1/SC 32 Data Management and Interchange WG3 Database Languages*, 2000, pp. 7.
12. Kalyanakrishnam, M., Z. Kalbarczyk, and R. Iyer. *Failure Data Analysis of LAN of Windows NT Based Computers*, in *Proc. of 18th Symposium on Reliable and Distributed Systems (SRDS '99)*, 1999, Lausanne, Switzerland, pp. 178-187.
13. Schneider, F., *Byzantine generals in action: Implementing fail-stop processors,* ACM Transactions on Computing Systems, 1984, 2(2): pp. 145-154.
14. Gashi, I., P. Popov, and L. Strigini. *Fault diversity among off-the-shelf SQL database servers*, in *Proc. of Inter. Conf. on Dependable Systems and Networks (DSN'04)*, 2004, Florence, Italy, IEEE Computer Society Press: to appear.
15. Chandra, S. and P.M. Chen. *How fail-stop are programs*, in *Proc. of 28th IEEE International Symposium on Fault-Tolerant Computing (FTCS-28)*, 1998, IEEE Computer Society Press, pp. 240-249.

16. Gray, J. *Why do computers stop and what can be done about it?*, in *Proc. of 5th Symp. on Reliability in Distributed Software and Database Systems (SRDSDS-5)*, 1986, Los Angeles, CA, USA, IEEE Computer Society Press, pp. 3-12.

17. Chandra, S. and P.M. Chen. *Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software*, in *Proc. of Inter. Conf. on Dependable Systems and Networks (DSN 2000)*, 2000, NY, USA, IEEE Computer Society Press, pp. 97-106.

18. Jimenez-Peris, R., et al., *Are Quorums an Alternative for Data Replication?,* ACM Transactions on Database Systems, 2003, 28(3): pp. 257-294.

19. Jimenez-Peris, R., et al. *How to Select a Replication Protocol According to Scalability, Availability and Communication Overhead*, in *Proc. of Int. Symp. on Reliable Distributed Systems (SRDS)*, 2001, New Orleans, Louisiana, IEEE Computer Society Press, pp. 24 -33.

20. Kemme, B. and G. Alonso. *Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication*, in *Proc. of Int. Conf. on Very Large Databases (VLDB)*, 2000, Cairo, Egypt.

21. Anderson, T. and P.A. Lee, *Fault Tolerance: Principles and Practice (Dependable Computing and Fault Tolerant Systems, Vol 3)*, 2nd Revised ed, 1990, Springer-Verlag.

22. Gray, J. and A. Reuter, *Transaction processing : concepts and techniques*, 1993, Morgan Kaufmann.

23. Tso, K.S. and A. and Avizienis. *Community Error Recovery in N-Version Software: A Design Study with Experimentation*, in *Proc. of 17th IEEE International Symposium on Fault-Tolerant Computing (FTCS-17), Pittsburgh, Pennsylvania, July 6-8 1987*, 1987, pp. 127-133.

24. Jimenez-Peris, R., Patino-Martinez, and G. Alonso. *An Algorithm for Non-Intrusive, Parallel Recovery of Replicated Data and its Correctness*, in *Proc. of 21st IEEE Int. Symp. on Reliable Distributed Systems (SRDS 2002)*, 2002, Osaka, Japan, pp. 150-159.

25. Poledna, S., *Replica Determinism in Distributed Real-Time Systems: A Brief Survey,* Real-Time Systems Journal, 1994, 6: pp. 289-316.

26. Powell, D., *Delta-4: A Generic Architecture for Dependable Distributed Computing*, Springer-Verlag Research Reports ESPRIT, 1992, Springer-Verlag.

27. Popov, P., et al. *Software Fault-Tolerance with Off-the-Shelf SQL Servers*, in *Proc. of 3rd International Conference on COTS-based Software Systems, ICCBSS'04*, 2004, Redondo Beach, CA USA, Springer: to appear.

28. Gruber, M., *Mastering SQL*, 2000, SYBEX.

29. Melton, J., *(ISO-ANSI Working Draft) Persistent Stored Modules (SQL/PSM)*, 2002, http://www.jtc1sc32.org/sc32/jtc1sc32.nsf/Attachments/9611E99B3901802188256D95005B0184/$FILE/32N1008-WD9075-04-PSM-2003-09.PDF

30. Microsoft, *SQL Server "Yukon"*, 2003, http://www.microsoft.com/sql/yukon/productinfo/default.asp

31. Poledna, S., *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, 1996, Kluwer Academic Publishers.

32. Ammann, P.E. and J.C. Knight. *Data Diversity: an Approach to Software Fault-Tolerance*, in *Proc. of 17th IEEE International Symposium on Fault-Tolerant Computing (FTCS-17)*, 1987, Pittsburgh, Pennsylvania, USA, IEEE Computer Society Press, pp. 122-126.

33. Chen, P.M., et al., *Raid: High-Performance, Reliable Secondary Storage,* ACM Computing Surveys, 1994, 26(2): pp. 145-185.

34. TPC, *TPC Benchmark C, Standard Specification, Version 5.0.*, 2002, http://www.tpc.org/tpcc/

35. Weismann, M., F. Pedone, and A. Schiper. *Database Replication Techniques: a Three Parameter Classification*, in *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, 2000, Nurnberg, Germany, IEEE Computer Society Press, pp. 206-217.

36. Vaysburd, A. *Fault Tolerance in Three-Tier Applications: Focusing on the Database Tier*, in *Proc. of 18th IEEE Symposium on Reliable Distributed Systems (SRDS'99)*, 1999, Lausanne, Switzerland, IEEE Computer Society Press, pp. 322-327.

37. Pedone, F. and S. Frolund. *Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases*, in *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, 2000, Nurnberg, Germany, IEEE Computer Society Press, pp. 176-185.

38. Jimenez-Peris, R. and M. Patino-Martinez, *D5: Transaction Support*, 2003, ADAPT Middleware Technologies for Adaptive and Composable Distributed Components, pp. 20.

39. Patino-Martinez, M., R. Jimenez-Peris, and G. Alonso. *Scalable Replication in Database Clusters*, in *Proc. of International Conference on Distributed Computing, DISC'00*, 2000, Springer, pp. 315-329.

40. Jimenez-Peris, R., et al. *Scalable Database Replication Middleware*, in *Proc. of 22nd IEEE Int Conf on Distributed Computing Systems*, 2002, Vienna, Austria, pp. 477-484.

41. Kemme, B., A. Bartoli, and O. Babaoglu. *Online Reconfiguration in Replicated Databases Based on Group Communication*, in *Proc. of Int. Conf. on Dependable Systems and Networks (DSN 2001)*, 2001, Goteborg, Sweden, IEEE Computer Society Press, pp. 117-126.

42. Voas, J., *Deriving Accurate Operational Profiles for Mass-Marketed Software*, 2000, http://www.cigitallabs.com/resources/papers/