City Research Online

City, University of London Institutional Repository

Citation: Gashi, I. (2007). Software dependability with off-the-shelf components. (Unpublished Doctoral thesis, City, University of London)

This is the accepted 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/30436/

Link to published version:

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

SOFTWARE DEPENDABILITY WITH OFF-THE-SHELF COMPONENTS

Ilir Gashi

i.gashi@city.ac.uk

Centre for Software Reliability

City University

London EC1V 0HB

United Kingdom

PhD Thesis

June, 2007

Table of Contents

List of Tables

List of Figures

ACKNOWLEDGMENTS

I am indebt to my supervisor Dr. Peter Popov for his continuous help, advice, support, encouragement and friendship. The completion of this thesis would not have been possible without his continuous guidance and constructive suggestions.

I would also like to thank other colleagues at CSR that have provided valuable help and advice on various parts that form part of this thesis: Prof. Peter Bishop, Prof. Bev Littlewood, Prof. Lorenzo Strigini and Dr. David Wright.

I thank Mr. Vladimir Stankovic for allowing me to reuse and modify the experimental harness he had developed, which allowed for performance measurements of rephrasing to be calculated in Paper 4 of chapter V, as well as for his continuous friendship and support.

Dr. Andrey Povyakalo provided help with statistical functions in *R*. Mrs. Basi Issacs also provided valuable help and support with the administrative parts of the PhD and thesis. I would also like to thank CSR and Prof. Robin Bloomfield for securing the funding for my PhD, as well as other present and previous colleagues at CSR.

Last, but not least, I would like to thank my family (especially my aunties here in the UK) for many years of support. They have always been part of my education and have encouraged me all along.

I grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Abstract

When systems are built out of "off-the-shelf" (OTS) products, fault tolerance is often the only viable way of obtaining the required system dependability. Due to low acquisition costs, even using multiple versions of software in a parallel architecture, a scheme formerly reserved for few and highly critical applications, may become viable for many other applications. A wide range of solutions for employing fault tolerance are known in the literature, but the difficulty remains in assessing the possible dependability gains that may be achieved.

The research detailed in this thesis will aim to provide a new approach to assessing the dependability gains that may be achieved through software fault tolerance via modular redundancy with diversity in complex OTS software. OTS SQL database server products have been used in the studies: they are a very complex, widely-used category of off-the-shelf products meaning the results reported in this thesis may be of immediate interest to practitioners dealing with complex software systems. Bug reports of the servers were used as evidence in the assessment: they were the only direct dependability evidence that was found for these products. A sample of bug reports from four OTS SQL database server products and later releases of two of them have been studied to check whether they would cause coincident failures in more than one of the products. Very few bugs were found to affect more than one product, and none caused failures in more than two. Many of these faults caused systematic, non-crash failures, a category ignored by most studies and standard implementations of fault tolerance for databases. Use of different releases of the same product was also found to tolerate a significant fraction of the faults for one of the products used in the study. Therefore, a fault-tolerant server, built with diverse OTS servers products, seems to have a good chance of delivering improvements in availability and failure rates compared with the individual OTS server products or their replicated, non-diverse configurations.

Data diversity in the form of "SQL rephrasing rules" was also found to be a very useful fault tolerance mechanism. Data diversity is possible with these products thanks to the redundancy that exists in the SQL language: a statement can be specified in multiple different but logically equivalent ways. The results of all these studies are reported in this thesis and their implications, the architectural options available for exploiting them, and the difficulties that they may present are discussed.

Two reliability models developed previously by colleagues at the Centre for Software Reliability, City University have been extended to enable their use in assessing a fault-tolerant 1-out-of-2 diverse server. The bug reports were used as evidence in the assessment with one of these models which enables an assessor to choose the pair of servers, from the possibly many pairs available, which will yield the highest reliability gains. The other model that was extended required additional data that was not available for the database servers. Therefore another approach was studied in which bug reports data alone can be used to derive estimates of possible reliability gains that may be expected from employing a 1-out-of-2 diverse server in comparison to a non-diverse one.

I. Introduction

1. Motivation and Aims

The use of "off-the-shelf" (OTS) software is ubiquitous. Their use, rather than custom-built products, is attractive in terms of acquisition costs and time to deployment but brings concerns about dependability and "total cost of ownership". For safety- or business-critical applications, in particular, purpose-built products traditionally would come with extensive documentation, evidence of good development practice and of extensive verification and validation; with mass-distributed OTS products on the other hand, users (system designers or end users) invariably find not only a lack of this documentation, but anecdotal evidence of serious failures and/or bugs that undermines trust in the product. Despite the large-scale adoption of some products, there is usually no formal statistical documentation of achieved dependability levels, from which a user could attempt to extrapolate the levels to be achieved in his/her own usage environment.

For all these reasons software fault tolerance is often the only viable way of obtaining the required system dependability when systems are built out of OTS products (Popov, Strigini et al. 2000), (Valdes, M. Almgren et al. 2003), (Hiltunen, Schlichting et al. 2000). Fault tolerance may take multiple forms, with examples ranging from simple error detection and recovery add-ons (e.g. "wrappers" (Popov, Strigini et al. 2000)) to "diverse modular redundancy" (e.g. "N-version programming": replication with diverse versions of the products) (Avizienis and Kelly 1984), (Laprie, Arlat et al. 1990). With OTS products the latter approach becomes more viable due to the availability of a large number of similar products which may even be open-source and/or freely distributed.

These approaches are well known from the literature. Questions remain, however, for the developers of systems using OTS products:

- what dependability gains may be achieved from the use of fault tolerance mechanisms with OTS products?
- more specifically, what dependability information/data exists for OTS products and how can this information be used to assess the dependability gains that may be achieved from employing fault tolerance mechanisms.

- for fault tolerance configurations employing diverse modular redundancy, which OTS products, from the (possibly many) available ones, should be chosen to achieve the highest dependability gains?
- what are the implementation difficulties?
- what costs (developmental, procurement, operational, maintenance etc.) may be expected?

The purpose of this thesis is to attempt to answer some of these questions for a category of OTS products: SQL database servers, or "database management systems" (DBMSs). This category of products offers a realistic case study of the advantages and challenges of software fault tolerance in OTS products. DBMS products are complex, mature enough for widespread adoption, and yet with many faults in each release. Studying the implications of fault tolerance with this complex category of OTS products will therefore likely have important practical implications as practitioners may find the results relevant to their applications, while other smaller or experimental products may be faced with scepticism due to their limitations in practical use.

2. Summary of work

Developing an SQL server using diverse modular redundancy (i.e. several OTS DBMS products and suitably adapted "middleware" (Bakken 2003) for "replication" (Bernstein, Hadzilacos et al. 1987) management) requires strong evidence of dependability benefits it can yield: for example empirical evidence that likely failures of the DBMS products, which may lead to serious consequences, are unlikely to be tolerated without diversity. To investigate such empirical evidence two studies were carried out with four OTS DBMS products (both open-source and closed-development) and later releases of two of these products (the open-source ones). The DBMS products used in the first study were:

- Open source:
  - PostgreSQL 7.0
  - Interbase 6.0
- Closed development
  - Microsoft SQL Server 7
  - Oracle 8.0.5

In the second study the following later releases of the open-source DBMS products were used:

- PostgreSQL 7.2
- Firebird 1.0 (Firebird is the open-source descendent of Interbase 6.0)

The purpose of the studies was to investigate whether diverse modular redundancy has a potential to deliver significant improvement of dependability of DBMS products, compared to solutions for data replication that can only tolerate crash failures. The only direct dependability evidence that is available for the DBMS products are their *fault reports*. Therefore a preliminary evaluation step concerns fault diversity rather than failure diversity. By manual selection of test cases, one can check whether the diverse modular redundant configuration would tolerate the known bugs in the repositories of bugs reported for the various DBMS products. To this end, in the first study, a total of 181 bug (fault) reports were collected for the DBMS products used. For each bug, the test case that would trigger it was run on all four DBMS products (if possible), to check for coincident failures. The number of coincident failures was found to be very low.

The results of the first study were very intriguing and pointed to potential for serious dependability gains from using diverse off-the-shelf DBMS products. However these results concern only a specific snapshot in the evolution of these products. Therefore the study was repeated for later releases of these DBMS products. A further 92 new bug reports for the later releases of the open-source DBMS products were collected (no further bugs were collected for the closed-development DBMS products as the reproduction scripts needed to trigger the fault were missing in most of them - but the new bug scripts were still run on the two closed-development DBMS products used in the first study). The results of the second study substantially confirmed those of the first: very few bugs scripts are again found to cause coincident failures.

The bugs reported for the new releases were also run on the older releases of those DBMS products, and vice versa. The results for PostgreSQL are very interesting: most of the old bugs had been fixed in the new release and a large proportion of the newly reported bugs did not cause failure (or could not be run at all) in the old release. This would suggest that dependability improvements can be gained by employing this more limited form of diversity: running different releases of a DBMS product from the same

vendor. Similar practices have been applied for embedded and safety critical systems (Cook and Dage 1999), (Tai, Tso et al. 2002). Note that the idea of using old and new releases of the same program to improve dependability was first mentioned by Brian Randell in his work on recovery blocks (Randell 1975): one possible setup of the recovery block scheme would use the earlier releases of the primary alternate as sources of secondary alternates.

The mechanism of "data diversity" (Ammann and Knight 1988) in the form of "rephrasing" was also studied. This approach is applicable due to the natural redundancy that exists in the SQL language: statements may be formulated in different, but logically equivalent, ways. A limited number of "rephrasing rules" were defined and applied to bug reports of the open-source DBMS products examined in the two studies; the results show that the rephrasing rules would tolerate at least 60% of these bugs that could be run on more than one DBMS product. Data diversity may be used with or without diversity of DBMS products; when diverse products are used it can be especially useful for aiding with *diagnosis* of the failed product and *recovery* of the state of the failed product (through the re-execution of a rephrased statement which fails in its original form, i.e. through *forward error recovery*) in addition to failure *detection.*

The results from the two studies with the bugs point to a potential for serious dependability gains from assembling a fault-tolerant server from two or more DBMS products. But they are not definitive evidence. An extensive discussion is presented to clarify to what extent the observations like those reported in this thesis allow one to predict such gains. Three modelling approaches are also presented which show how the assessment of the gains in dependability that may be achieved through diversity can be performed utilising the bug reports.

In the first modelling approach an existing model developed at our centre (Littlewood, Popov et al. 2000) has been adapted and extended which allows for an optimal selection of a pair of OTS products to be used in a diverse fault-tolerant server. In this model the assessment results are subject to uncertainty which can impact the decisions about which pair of OTS products is chosen. The model also enables representing the dependencies that exists between uncertainties associated with the reliability of each OTS product in the pair. The use of the model for selection of an optimal pair of DBMS products in the

study is then shown. The evidence used in the assessment are the collected bug reports. It is also shown that using the same mathematical model, but redefining the variables, the assessment of single products may be carried out in which both the *correctness* (*reliability*) and *timeliness* (*performance*) attributes of the results of the product are considered and the best product is selected with respect to both of these attributes.

Research has also been done on the possibilities of modelling the reliability of a diverse 1-out-of-2 system of DBMS products. This work has been in two strands:

- Adapting and extending a previous *reliability growth model* (*RGM*) (Littlewood 1981) (originally developed for modelling the reliability growth of a single bespoke software system) to allow for modelling the reliability growth of a diverse 1-out-of-2 system of DBMS products. A detailed description is given on how the model may be used to perform the assessment but, due to unavailability of the necessary data, no empirical work with the model was performed.
- Developing a new modelling approach in which statements about the reliability of the diverse 1-out-of-2 system can be made (under certain assumptions) purely using bug reports data as evidence. The application of the model is illustrated using the data from the study with the bug reports of the DBMS products.

3. Contributions of the Thesis

The main contributions of this thesis are the following:

- Two studies have been completed with a total of 273 bug reports for 6 OTS DBMS products which indicate that bugs triggered in one DBMS product would cause failures in another DBMS product only in *very few cases*. Even using different releases of the same DBMS product it is shown to have some benefit in terms of improved dependability. To the best of our knowledge we are not aware of any similar work which has studied *fault* diversity with OTS products (and this is also confirmed by reviewers of our papers).
- Data diversity (Ammann and Knight 1988) in the form of "SQL rephrasing rules" is shown to be a useful fault tolerance mechanism with or without diverse modular redundancy and may especially help in a diverse product configuration with failure diagnosis and state recovery of the failed product.

- A previous model developed within our centre (Littlewood, Popov et al. 2000) is adapted and extended for assessment of DBMS products in two different setups:
  - to select an optimal pair of DBMS products, when multiple (> 2) DBMS products are available, for use in a diverse fault-tolerant server configuration. The model is Bayesian and allows the expert to specify the *prior* distribution of the *probability of failure on demand* (*pfd*) of each of the DBMS products individually as well as the *pfd* of common failures of the two DBMS products. The data from the bugs study are then used as *evidence* and a *posterior* probability distribution is derived which quantifies the uncertainty and allows the assessor to make a selection of the optimal pair.
  - to select an optimal single DBMS product but taking into account both the product's output *correctness* (*reliability*) and *timeliness* (*performance*). Even though the mathematical details of the model remain the same as in the diverse setup above the variables of the model are redefined to cater for both types of failure, namely correctness and timeliness.
- A previous reliability growth model (Littlewood 1981), originally developed for modelling the growth of reliability in bespoke software systems, was extended for use in modelling the reliability growth of a 1-out-of-2 system of OTS products.
- And finally, another reliability modelling approach is presented which enables the assessor to make statements regarding the reliability of a 1-out-of-2 system based purely on the bug counts alone. The use of the model is then illustrated using the bug reports of the DBMS products.

With respect to the contributions listed above, the research outlined in this thesis could be most beneficial to the following people / organisations:

- *Developers of fault-tolerant systems which are constructed with OTS products*: the research outlined in this thesis provides evidence that potentially significant dependability gains may be obtained from employing diverse DBMS products; evidence is also shown on the effectiveness of data diversity through rephrasing and, for one DBMS product, the potential for limited, but significant, gains in dependability from using different releases of even the same product. The architectures that would enable the use and mixing of these approaches are also

extensively discussed. Therefore developers of fault-tolerant systems who seek options for enhancing the system dependability may benefit from the research presented in this thesis.

- *Organisations that require methods to enable selection (or ranking) of (either diverse or single) OTS products from the viewpoint of Dependability and/or Performance*: a modelling approach is presented which allows assessors to make selections of optimal diverse or single components from the viewpoint of dependability and/or performance. The use of the model is illustrated with the results from the bugs study as this was the only direct dependability evidence that was found for these products; however the model is general enough to allow for other types of evidence (such as data from statistical testing or dependability benchmarking results (Kanoun, Madeira et al. 2004)) to be utilised in the inference.
- *Vendors of DBMS products*: by running bug scripts reported for a DBMS product A, faults were uncovered in another DBMS product B which were not reported in the bug repositories (for our collection period) of product B. Vendors are therefore advised to test their products with bugs reported for products of other vendors.

4. Thesis outline

The main chapters of the thesis (chapters IV-VI) will be presented as a collection of seven papers, identified as *Paper-1* to *Paper-7* respectively (details to follow below and in more detail in chapter III). Each paper contains a "Related work" section in which the relevant literature for that paper is reviewed. However to improve the readability of the thesis a "Literature review" and a "Research outline" chapter will also be given before the main chapters with the papers. Therefore chapter I-III of this thesis may be useful to a reader who is interested in the main methodology and results of the thesis without the low level details. These details are then presented in Chapters IV-VI and the Appendix.

In summary:

Chapter I – Introduction: outlines the motivation for the work, and summarises the main results and outcomes of the thesis.

Chapter II - Literature review: contains a critical review of the related literature relevant to this thesis. Since the papers that form part of chapter IV-VI also contain related work sections, to avoid duplication the Literature review chapter only reviews in detail the relevant literature not discussed in those chapters.

Chapter III – Research overview: contains an overview of the research done in this thesis written at a level which is more comprehensive than an abstract but without the details given in the relevant papers and appendices in the thesis. The purpose of this chapter is to outline the main structure of the thesis and therefore clarify how the various parts of the work detailed in the papers are linked together.

Chapter IV – Fault diversity study: contains two papers which detail the two studies conducted with the bugs reported for (our sample of) DBMS products.

Chapter V – Architectural aspects of a diverse fault-tolerant server: contains two papers which detail the architecture of a diverse fault-tolerant server, including the data diversity mechanism (via SQL rephrasing) and empirical results from the application of rephrasing to the bug reports of the open-source DBMS products used in the two studies.

Chapter VI - Optimal selection of COTS components: contains three papers which detail the models and their applications with (our sample of) DBMS products and bug reports for the following purposes:

- selection of an optimal diverse pair of products
- selection of an optimal product, from the viewpoint of both correctness (reliability) and timeliness (performance)
- reliability modelling of a 1-out-of-2 diverse server

Chapter VII – Conclusions: outlines the main conclusions of this thesis and provisions for further work.

Appendix A - contains the full details of all of the bug reports used in the studies, the rephrasing rules defined and their application to the bug reports of (our sample of) open-source DBMS products.

References

Ammann, P. E. and J. C. Knight (1988), "*Data Diversity: An Approach to Software Fault Tolerance*", IEEE Transactions on Computers 37(4), pp: 418-425.

Avizienis, A. and J. P. J. Kelly (1984), "*Fault Tolerance by Design Diversity: Concepts and Experiments*", IEEE Computer 17(8), pp: 67-80.

Bakken, D. (2003), "*Middleware: What it is, and How it Enables Adaptively and Dependability*", *in proc.* 43 Meeting of IFIP WG 10.4 Dependable Computing and Fault Tolerance, Santa Maria, Sal Island, Cape Verde, pp: 13-40.

Bernstein, P. A., V. Hadzilacos and N. Goodman (1987), "*Concurrency Control and Recovery in Database Systems*", Reading, Mass., Addison-Wesley.

Cook, J. E. and J. A. Dage (1999), "*Highly Reliable Upgrading of Components*", *in proc.* Int. Conf. on Software Engineering (ICSE '99), IEEE-ACM, pp: 203-212.

Hiltunen, M. A., R. D. Schlichting, C. A. Ugarte and G. T. Wong (2000), "*Survivability Through Customization and Adaptability: The Cactus Approach*", *in proc.* DARPA Information Survivability Conference & Exposition.

Kanoun, K., H. Madeira, et al. (2004), "*DBench Dependability Benchmarks*", IST-2000-25425, http://www.laas.fr/DBench/Final/DBench-complete-report.pdf.

Laprie, J. C., J. Arlat, C. Beounes and K. Kanoun (1990), "*Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures*", IEEE Computer 23(7), pp: 39-51.

Littlewood, B. (1981), "*Stochastic Reliability Growth: a Model for Fault-Removal in Computer Programs and Hardware Designs*", IEEE Transactions on Reliability R-30(4), pp: 313-320.

Littlewood, B., P. Popov and L. Strigini (2000), "*Assessment of the Reliability of Fault-Tolerant Software: a Bayesian Approach*", *in proc.* Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP '00), Rotterdam, the Netherlands, Springer, pp: 294-308.

Popov, P., L. Strigini and A. Romanovsky (2000), "*Diversity for Off-The-Shelf Components*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '00) - Fast Abstracts supplement, New York, NY, USA, IEEE Computer Society Press, pp: B60-B61.

Randell, B. (1975), "*System Structure for Software Fault Tolerance*", IEEE Transactions on Software Engineering 1(2), pp: 220-232.

Tai, A. T., K. S. Tso, L. Alkalai, S. N. Chau and W. H. Sanders (2002), "*Low-Cost Error Containment and Recovery for Onboard Guarded Software Upgrading and Beyond*", IEEE Transactions on Computers 51(2), pp: 121-137.

Valdes, A., M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saidi, V. Stavridou and T. E. Uribe (2003), "*An Architecture for an Adaptive Intrusion-Tolerant Server*", *in* Lecture Notes in Computer Science (LNCS) 2845 - Selected Papers from 10th Int. Workshop on Security Protocols '02, B. Christianson, Crispo, B., Malcolm, J. A., Roe, M. (Eds.), Springer, pp: 158-178.

II. Literature Review

1. Introduction

The purpose of this chapter is to give an outline of the main topics of previous work which this thesis references or extends. However it must be emphasized that each of the papers that will be detailed in chapters IV-VI will contain reviews of the previous literature relevant for the topic being discussed in that respective paper. Therefore, to avoid duplication and repetition, this chapter will only contain additional details about topics which due to various length restrictions could not be discussed in the papers, as well as a review of more recent literature than that discussed in the papers.

2. Definitions

This section provides some concise definitions of terms that are encountered frequently in this thesis but have only been partly defined in the papers that form part of chapters IV-VI.

2.1 Database terms

The definitions of database terms given in this subsection are based on those given in (Bernstein, Hadzilacos et al. 1987). A *database* may be defined as consisting of named *data items*. Each of the data items has a *value*. The values of the data items at a given time comprise the *state* of the database. A *database management system* (DBMS) product may be defined as a collection of hardware and software modules that support the commands to access the database. These can be called *database operations* or *statements* which are usually split into two categories: *Data Manipulation Language* (*DML*) statements which are *read* and *write* statements for manipulating the data stored in the database; and the *Data Definition Language* (*DDL*) statements which are used to define the structure of the database (e.g. *Create Table* statement).

The DBMS must also support the *transaction operations (Start, Commit* and *Abort).* In its simplest form a *transaction* may be thought of as an execution of a program that accesses a shared database. However multiple operations may be executed *concurrently* therefore the DBMS product must allow for effective *concurrency control* and *recovery.* Ideally the DBMS product must guarantee all of the *ACID* properties of a transaction:

- *Atomicity*: A transaction either executes in its entirety or it is aborted
- *Consistency*: the state of the database before and after the transaction execution must remain consistent (the definition of consistency is application specific)
- *Isolation*: modifications of the transaction are not visible to other resources before it finishes
- *Durability*: a completed transaction is always persistent.

In practice these properties may be relaxed somewhat as a trade-off to better performance. Most commonly the *Isolation* property is relaxed. ANSI/ISO (American National Standards Institute / International Organisation for Standardisation) SQL standard (ANSI/ISO 2003) defines four *Isolation* levels for transactions (the list is given in increasing order of transaction isolation):

- *Read uncommitted*: a transaction A can read uncommitted changes made by another concurrent transaction B before transaction B has *commit*ted (so called *dirty reads* are allowed to occur).
- *Read committed*: a transaction A can read data which is then changed by another concurrent transaction B before transaction A *commit*s. Therefore transaction A will not be able to see the same data as before if it reissues the same read statement (so called *unrepeatable reads* are allowed to occur)
- *Repeatable read*: a transaction A can read from tables to which another concurrent transaction B is inserting data. Therefore transaction A will retrieve different results if it reissues a read statement that fulfils the condition of reading the newly inserted data (so called *phantom reads* are allowed to occur)
- *Serialisable*: all transactions are executed in complete isolation, i.e. the transactions produce the same output and have the same effect on the database as some *serial* execution of transactions.

These definitions have been criticised as ambiguous and not reflecting accurately the isolation provided by many DBMS products (Berenson, Bernstein et al. 1995). In the architectural proposals discussed in chapters IV and V of this thesis the *serialisable* isolation of transactions has been assumed.

A *replicated* database is a distributed database in which multiple copies of the same data items are stored in multiple sites. The main reason for using replicated data is to increase

DBMS availability. A DBMS product that manages a replicated database should behave like a DBMS product that manages a *one-copy* (i.e. non-replicated) database insofar as users can tell. In a one-copy database, users expect the interleaved execution of their transactions to be equivalent to a *serial* execution of those transactions. Since replicated data should be transparent to them, they would like the interleaved execution of their transactions on a replicated database to be equivalent to a serial execution of those transactions on a one-copy database. Such executions are called *one-copy serialisable*. The goal of concurrency control for replicated data is to guarantee the *same* one-copy serialisability amongst the replicas otherwise the replicas may become inconsistent. The replication can be *eager* – the coordination between the replicas happens before transaction *commit*s – or *lazy* – to improve performance, the coordination amongst the replicas may happen after the transaction *commit*s. Many replication algorithms have been proposed in the literature, and some of these together with more detailed explanations of data replication are discussed at length in the papers that form parts of chapters IV and V of this thesis. In the architectural proposals discussed in this thesis *eager* replication has been assumed.

2.2 Faults, errors and failures

The definitions of the terms *fault* (or *bug*), *error* and *failure* given in this sub-section are based on (Avizienis, Laprie et al. 2004).

A *failure* is said to occur when the system stops performing its required functions. An *error* is an erroneous (defective) internal state in the system, which propagates through the system and causes the failure. A *fault* is the triggering condition which when activated causes an error.

Therefore the event of the system *failure* lies in the end of a causal chain that begins with the activation of a *fault* under certain operating conditions and followed by the propagation of an *erroneous* internal state through the system.

In this thesis the term software *fault* and *bug* have been used interchangeably.

2.3 Fault tolerance terms

The definitions of the fault tolerance terms and mechanisms defined in this sub-section are based on (Anderson and Lee 1990).

A *fault-tolerant* system is a system that can continue in operation after some system faults have manifested themselves. *Fault tolerance* is therefore based on the premise that faults exist and that it is possible for the computer system to handle them without external interventions.

The goal of fault tolerance is to ensure that system faults do not result in system failure. However due to the higher cost associated with the use of fault tolerance it tends to be mostly employed in applications where a system failure would cause catastrophic accidents which would lead to loss of life, or where a system failure would lead to large economic losses.

(Anderson and Lee 1990) identify four constituent phases, which taken together, provide the general means of preventing faults from leading to a system failure:

1. *Error detection*: The system must detect that a particular state combination has occurred and could lead to a system failure.
2. *Damage assessment*: The parts of the system state, which have been affected by the fault, must be identified.
3. *Error recovery*: The system must restore its state to a known "safe" state. This may be achieved by correcting the damaged state (*forward error recovery*) or by restoring the system to a known safe state (*backward error recovery*).
4. *Fault treatment and continued system service*: This involves modifying the system so that the fault does not recur. In many cases, software faults manifest themselves as transient states (e.g. "*Heisenbugs*" (Gray 1986)). They are due to a peculiar combination of system inputs. For these faults no repair is necessary as normal processing can resume immediately after error recovery.

Fault tolerance can be achieved through both software and hardware, but throughout this thesis the software mechanisms will be discussed unless otherwise stated. Apart from (Anderson and Lee 1990) other references which provide extensive coverage of software fault tolerance are (Lyu 1995) and (Pullum 2001).

Other well-known *dependability* attribute terms (such as *reliability*, *availability* etc.) that will be used in this thesis are based on the definitions provided in (Avizienis, Laprie et al. 2004).

Implementations and practical examples of the use of the fault tolerance mechanisms defined above and their application to DBMS products will be referenced and discussed in the papers that form part of chapters IV and V of this thesis.

3. Design diversity

The central theme of this thesis is the use of *diverse* off-the-shelf products to increase the dependability of a system. Software *design diversity* is the phenomenon of *bespoke development* or *reuse* of multiple diverse versions of a software *program* (or existing *product*) from a common requirement specification with the goal of increasing the system reliability or availability. The intuitive underlying principle of design diversity is the simple longstanding belief that "two heads are better than one" and its advocacy for use with computer systems may be thought of as first being proposed by Charles Babbage (Babbage 1974), although by computer he meant a person.

The main reason for employing design diversity in software is due to software suffering exclusively from *design faults* (Littlewood, Popov et al. 2001) and not *physical faults* (such as wear-and-tear for example) which are hardware specific. A *design fault* in its simplest definition is a fault that is introduced in the software during its development (hence the word *design* in this context is used for the whole software *development* process). If non-diverse redundant copies of the same software product are used then these design faults will be simply replicated across the copies. Such replication of faulty software elements fails to enhance the fault tolerance of the system with respect to design faults.

The ideal goal of employing design diversity is to achieve *negative dependence* between the failure modes of the software products (i.e. whenever one fails the other one does not). *Independent failure* modes of the channels that constitute the diverse system would also be highly desirable as they would enable an assessor to easily calculate the probability of failure of the diverse system: the product of the failure probabilities of the individual channels in the diverse system would give the failure probability of the diverse

system. However virtually all of the experimental studies performed for measuring the benefits of design diversity (Chen and Avizienis 1978), (Kelly and Avizienis 1983), (Knight and Leveson 1986), (Avizienis, Lyu et al. 1988), (Eckhardt, Caglayan et al. 1991) have found faults, which cause coincident failures in more than one version with probability significantly higher than would be expected if the versions truly had independent failure modes.

The Knight and Leveson experiment (Knight and Leveson 1986) is the best known of these experiments. The experiment was carried out to test the hypothesis of independence of failures of diverse versions of software programs. Nine students from University of Virginia and eighteen from University of California in Irvine were asked to write programs from a single requirements specification. The result was twenty-seven programs. No overall software development methodology was imposed on the students. However they were required to write the programs in Pascal and to use only a specified compiler and operating system. The resulting programs were then subjected to an acceptance test. Once all the versions had passed their acceptance test the versions were subjected to the experimental treatment, which consisted of simulation of a production environment. A total of one million tests were run on each of the twenty-seven versions.

It was found in (Knight and Leveson 1986) that test cases occurred in which eight of the twenty-seven versions failed. They also found that the coincident failures did not necessarily occur in the versions supplied by one university but they also occurred in the versions supplied by different universities (in this case two universities).

For the particular programs that were written for this experiment, Knight and Leveson concluded that the assumption of independence of failures does not hold. And based on a probabilistic independence model, their results indicated that the null hypothesis of versions failing independently has to be rejected at the 99 percent confidence level (even though they stress that these results are conditional on the application that they used and that other experiments should be carried out before drawing general conclusions). An important point made by Knight and Leveson (Knight and Leveson 1986) in the conclusion of this experiment is that certain parts of any problem are just more difficult to solve than others and will lead to the same faults by different programmers. Littlewood

and Miller (Littlewood and Miller 1989) call this the *variability in difficulty* of processing different inputs.

Eckhardt and Lee (Eckhardt and Lee 1985) dealt with the case of several versions using a single common development methodology. The most important achievement of their work was to demonstrate analytically that truly independently developed versions would necessarily fail dependently. Thus, the empirical observation by Knight and Leveson (Knight and Leveson 1986) was reinforced theoretically.

Littlewood and Miller (Littlewood and Miller 1989) extended the Eckhardt and Lee model for diverse development methodologies. The notion of diverse methodologies in its widest sense to mean, for example, different personnel, rescues the multiversion programming from the Eckhardt and Lee claim of necessary dependent failure, which, according to the Littlewood and Miller model, turns out to be a worst case scenario. This study rather than concentrating on the goal of independent failure behaviours for the versions, showed instead that the important idea at all levels is diversity. In particular the role of *covariance* between methodologies was studied. (Littlewood and Miller 1989) conclude that there is an advantage gained from forcing diversity at all levels of development from design, programming, testing, as well as using different personnel and at different sites during development. If three methodologies are available they should be used; if only two are available they should be used (Littlewood and Miller 1989).

A generalisation of (Littlewood and Miller 1989) and (Eckhardt and Lee 1985) in which the versions are allowed to evolve (and their reliability to grow) through debugging is provided in (Popov and Littlewood 2004).

There is a vast literature on design diversity: a more thorough review of the effectiveness of design diversity (both experimental results and probabilistic modelling) is given in (Littlewood, Popov et al. 2001); design aspects are discussed in (Strigini 2005). Other literature will also be discussed and referenced in the papers that form parts of chapters IV to VI of this thesis.

4. Reliability growth modelling

Modelling the reliability of the software products and specifically *predicting* reliability growth as the faults in the software are fixed is the main purpose of the field of research

called *reliability growth modelling*. Many models have been proposed over the years. The assumptions and details of one of these models (specifically the Littlewood model (Littlewood 1981)) will be discussed in the second paper that forms part of chapter VI (specifically Section 2.3 of Paper-6). A comprehensive overview of this research field is given in (Lyu 1996). Chapter 3 of (Lyu 1996) contains a wide ranging survey of the models whereas Appendix B of the same reference contains a review of the reliability theory, analytical techniques and basic statistics.

5. Other relevant literature already referenced in chapters IV to VI

This section lists other related literature topics which are referenced and described in chapters IV to VI of this thesis.

Database replication: both classical and more recent replication techniques have been described and referenced in the papers that form parts of chapters IV and V (specifically Section 2.1 of Paper-1, Section 2.1 of Paper-2 and Sections 3.1 and 5 of Paper-3).

Data diversity: the concept of "data diversity" first described in (Ammann and Knight 1988) has been thoroughly described in both papers that form part of chapter V (specifically Section 3.6 of Paper-3 and Section 2.4 of Paper-4).

Diversity with off-the-shelf applications: related work which has explored the use of diverse off-the-shelf applications for increasing the dependability and security of a system are described and referenced in the two papers that form part of chapter IV (specifically Section 2.3 of Paper-1 and Section 5.5 of Paper-2).

Empirical studies with faults and failures: the relatively few studies of faults and failures of software products (both off-the-shelf and bespoke ones) are described in the two papers of chapter IV (specifically Section 2.2 of Paper-1 and Section 5.4 of Paper-2)

Reproducibility of failures: Jim Gray's classification of faults (*Bohrbugs* and *Heisenbugs*) (Gray 1986) has been referenced and described in the papers forming parts of chapters IV and V (specifically Section 3.2 of Paper-1, Section 3.1.1 of Paper-2 and Section 2 of Paper-3).

COTS assessment and selection: there are a myriad of approaches that have been proposed in the COTS community (primarily in the International Conference on COTS-

Based Software Systems (ICCBSS)) for assessment and ultimately selection of COTS products. Some of these approaches are referenced and described in two of the papers that form part of chapter VI (specifically Section 2 of Paper-5 and Section 6 of Paper-7)

6. Other relevant literature not referenced elsewhere in the thesis

6.1 ReSIST D12

Another important source of very recent relevant literature, not referenced in the papers that form parts of chapters IV-VI, is deliverable D12 (ReSIST 2006) of the network of excellence ReSIST (Resilience for Survivability in Information Society Technologies) sponsored by the European Union Framework Programme 6. Deliverable D12 (ReSIST 2006) describes the state of knowledge, in the ReSIST partners, of the technologies for building *resilience* (Hollnagel, Woods et al. 2006) and it was published in autumn 2006. D12 is in three layers: a brief overview of the state of knowledge; five survey-style parts which detail the *architectural (Part Arch)* solutions and *algorithms (Part Algo)* used to build resilient systems, the resilience of *socio-technical systems (Part Socio)* and methods for *evaluation (Part Eval)* and *verification (Part Verif)* of the resilience of systems; and an appendix which contains a collection of papers produced by the ReSIST partner sites during the year 2006 which contain more in depth analysis and results of the topics detailed in the five parts of the second layer.

With reference to this thesis, the most relevant parts of D12 (ReSIST 2006) are *"Part Arch"* (specifically Section 3 titled "Building resilient architectures with off-the-shelf components") and *"Part-Eval"* (specifically Section 4 titled "Diversity"), though these topics have also been covered (or will be covered in chapters IV-VI) in ample detail already in this thesis. D12 (ReSIST 2006) also contains two papers that form part of this thesis, specifically Paper-2 and Paper-4.

Part-Eval of D12 (ReSIST 2006) contains descriptions of other related approaches for evaluation of computer-based systems. A brief description of one of these methods, namely Dependability Benchmarking, which is most related to the approaches proposed in this thesis is given next.

6.2 Dependability benchmarking

Dependability benchmarking of a system involves the evaluation of dependability and/or performance attributes of a system either experimentally or with a combination of experimentation and modelling (Kanoun and Crouzet 2006). Dependability benchmarking combines the *workload* defined by existing performance benchmarks (e.g. TPC-C (TPC 2002)) with a *faultload*. The faultload defines the types of faults that are used with the workload to derive dependability measures for the system. A closely related field is *robustness testing* (which is described in *Part-Verif* of D12 (ReSIST 2006)).

To be meaningful the benchmark needs to satisfy a set of properties (e.g. representativeness, repeatability, non-intrusiveness etc.): these are explained in *Part-Eval* of D12 (ReSIST 2006) and in (Kanoun, Madeira et al. 2004). In (Kanoun and Crouzet 2006) it is stated that complex analysis is required to combine the effects of the workload and the faultload. The *representativeness* of the benchmark remains a key issue, i.e. how well the benchmark represents the typical use of the target system (also referred to as "operational profile" (Musa 1993) in reliability growth modelling). The definition and *representativeness* of the *faultload* is considered the most difficult part of defining a Dependability Benchmark (Kanoun, Madeira et al. 2004). The results from the bug studies that are presented in this thesis would at first glance seem to be good evidence to calibrate the *faultload* of the dependability benchmarks for DBMS products (as part of the DBench project (Kanoun, Madeira et al. 2004) a Benchmark Specification for Online Transaction Processing (OLAP) has been developed which uses the TPC-C benchmark as a *workload*). However DBench does not allow injecting faults in the system being tested as it would violate the *non-intrusiveness* property of the benchmark: in DBench faults are not injected directly in the *Benchmark Target* (*BT*) (e.g. a DBMS product), only on the *System Under Benchmark (SUB)* (e.g. the underlying Operating System and hardware over which the *BT* runs). Another difficulty stems from the fact that what has been studied in this thesis are failure points or regions (especially coincident failure points or regions in more than on DBMS product) rather than defects in source code: a better analysis of defects in the source code is given in (Duraes and Madeira 2006) which was the evidence used for the definition of the faultload in DBench (Kanoun, Madeira et al. 2004). However, the bug reports that have been collected as part of this thesis contain

very detailed *bug scripts* for reproducing the *failure* (the erroneous behaviour that the reporter of the bug observed) and could be potentially very useful to define more "stressful" *workloads* in Dependability Benchmarking (the workload of DBench for OLAP is based on TPC-C (TPC 2002) - a benchmark defined for performance rather than dependability which contains very simple statements that may not be highly representative of a real operational use and are unlikely to cause any failure).

As is acknowledged in *Part-Eval* of D12 (ReSIST 2006), the development and use of dependability benchmarks are still at an early stage.

6.3 Performability

Performability is a unification of performance and dependability (Tai, Meyer et al. 1996), (Haverkort, Marie et al. 2001); the concept arose from the need to model and evaluate systems that exhibit degradable performance in the presence of faults (Haverkort, Marie et al. 2001). A specific performability measure is obtained by defining what performance means for a given situation: in (Haverkort, Marie et al. 2001) it is detailed that there exist a variety of choices about the measures for performance, ranging from binary-valued performance (*on-time* or *late* with respect to a predefined timeout value) to continuous-valued performance such as throughput rates and processing delays. Existing models on evaluating performability of systems tend to specify performance in continuous time spectrum which would then give more detailed measures of performability of a system (Haverkort, Marie et al. 2001). In this thesis (more precisely in Paper-7) we present a simplified approach to considering both performance (timeliness) and reliability (correctness) attributes of an OTS product in which both of these attributes take binary values (on-time vs late and correct vs incorrect) leading to four possible outcomes for each demand sent to the OTS product. An existing Bayesian model (Littlewood, Popov et al. 2000), defined previously at the Centre for Software Reliability for modelling the reliability of a diverse system, was then adapted to model both performance and reliability of a single OTS product.

6.4 Related work on fault-tolerant middleware architectures

An extensive discussion of the architectural solutions for a fault tolerant SQL database server will be given in the papers that form part of chapters IV and V of this thesis. These papers also contain references to related work on middleware for fault tolerance. In this section one of the papers not referenced elsewhere in the thesis will be elaborated as it contains a complementary approach to what will be presented in chapters IV and V of the thesis. In (Bondavalli, Chiaradonna et al. 2004) a three tier architecture is proposed in which clients communicate with a legacy application (written in C which uses PostgreSQL DBMS product for stable storage) via a layer of middleware. The middleware layer contains components which perform standard fault tolerance and replication duties such as replication management, adjudication and recovery (these will also be discussed in chapters IV and V of this thesis) but also evidence-accruing diagnosis mechanisms which collect data about failures of the legacy application and allow for on-line selection of more optimal recovery strategies depending on the severity of the fault. A performability analysis was then conducted by the authors (via both direct measurements and analytical modelling) and their results suggest that substantial performability gains may be achieved from using the diagnostic system. The types of faults injected in the system for the experimental studies were those that cause hardware faults and software aging faults and the recovery mechanisms were primarily based on rejuvenation (with rejuvenation performed at different levels of granularity – host level, application level or database level). The main differences between the research pertinent to the fault-tolerant architecture presented in (Bondavalli, Chiaradonna et al. 2004) and what is presented in this thesis are highlighted next. In this thesis:

- The benefits of DBMS product diversity against *actual* bugs in these products have been explored (rather than using transient fault injection)
- To aid with diagnosis as well as state recovery of faulty channel(s) in diverse product configurations, data diversity (Ammann and Knight 1988) in the form of SQL rephrasing has also been studied
- Operating system and hardware faults and the benefits of diversity against these types of faults have *not* been studied.

This can be contrasted with the work of (Bondavalli, Chiaradonna et al. 2004) where:

- The legacy application is replicated (in Triple Modular Redundant (non-diverse) configuration) in diverse hardware and operating systems
- An implementation of an evidence-accruing diagnosis and recovery mechanism is presented
- The effects of the hardware and software aging type faults are studied.

The two strands of research therefore complement each other rather well. For examples of other research on the middleware organisation for replication the interested reader is referred to (Baldoni, Marchetti et al. 2002), (Marchetti, Baldoni et al. 2006).

References

Ammann, P. E. and J. C. Knight (1988), "*Data Diversity: An Approach to Software Fault Tolerance*", IEEE Transactions on Computers 37(4), pp: 418-425.

Anderson, T. and P. A. Lee (1990), "*Fault Tolerance: Principles and Practice (Dependable Computing and Fault Tolerant Systems, Vol 3)*", Springer Verlag.

ANSI/ISO (2003), "*Information technology - Database languages - SQL*", INCITS/ISO/IEC 9075.

Avizienis, A., J.-C. Laprie, B. Randell and C. Landwehr (2004), "*Basic Concepts and Taxonomy of Dependable and Secure Computing*", IEEE Transactions on Dependable and Secure Computing 1(1), pp: 11-33.

Avizienis, A., M. R. Lyu and W. Schuetz (1988), "*In search of effective diversity: A six-language study of fault-tolerant flight control software*", *in proc.* Int. Symp. on Fault-Tolerant Computing (FTCS '88), Tokyo, Japan, pp: 15-22.

Babbage, C. (1974), "*On the Mathematical Powers of the Calculating Engine (Unpublished manuscript, December 1837)*", *in* The Origins of Digital Computers: Selected Papers, B. Randell (Ed.), Springer, pp: 17-52.

Baldoni, R., C. Marchetti, et al. (2002), "*Active Software Replication through a Three-Tier Approach*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS'02), Osaka, Japan, IEEE Computer Society Press, pp: 109-118.

Berenson, H., P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil (1995), "*A Critique of ANSI SQL Isolation Levels*", *in proc.* Int. Conf. on Management of Data (SIGMOD '95).

Bernstein, P. A., V. Hadzilacos and N. Goodman (1987), "*Concurrency Control and Recovery in Database Systems*", Reading, Mass., Addison-Wesley.

Bondavalli, A., S. Chiaradonna, et al. (2004), "*Effective Fault Treatment for Improving the Dependability of COTS and Legacy-Based Applications*", IEEE Transactions on Dependable and Secure Computing 1(4), pp: 223-237.

Chen, L. and A. Avizienis (1978), "*N-version Programming: A Fault Tolerance Approach to Reliability of Software Operation*", *in proc.* Int. Symp. on Fault-Tolerant Computing (FTCS '78), Toulouse, France, IEEE Computer Society Press, pp: 3-9.

Duraes, J. A. and H. S. Madeira (2006), "*Emulation of Software Faults: A Field Data Study and a Practical Approach*", IEEE Transactions on Software Engineering 32(11), pp: 849-867.

Eckhardt, D. E., A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk and J. P. J. Kelly (1991), "*An experimental evaluation of software redundancy as a strategy for improving reliability*", IEEE Transactions on Software Engineering 17(7), pp: 692-702.

Eckhardt, D. E. and L. D. Lee (1985), "*A theoretical basis for the analysis of multiversion software subject to coincident errors*", IEEE Transactions on Software Engineering 11(12), pp: 1511-1517.

Gray, J. (1986), "*Why Do Computers Stop and What Can be Done About it?*" *in proc.* Int. Symp. on Reliability in Distributed Software and Database Systems (SRDSDS '86), Los Angeles, CA, USA, IEEE Computer Society Press, pp: 3-12.

Haverkort, B. R., R. Marie, et al., Eds. (2001), "*Performability Modelling: Techniques and Tools*", Wiley, Chichester, England.

Hollnagel, E., D. D. Woods and N. Leveson, Eds. (2006), "*Resilience engineering: concepts and precepts*", Ashgate Pub Co.

Kanoun, K. and Y. Crouzet (2006), "*Dependability Benchmarks for Operating Systems*", International Journal of Performability Engineering 2(3), pp: 277 - 289.

Kanoun, K., H. Madeira, et al. (2004), "*DBench Dependability Benchmarks*", IST-2000-25425, http://www.laas.fr/DBench/Final/DBench-complete-report.pdf.

Kelly, J. P. J. and A. Avizienis (1983), "*A Specification-Oriented Multi-Version Software Experiment*", *in proc.* Int. Symp. on Fault-Tolerant Computing (FTCS '83), Milano, Italy, pp: 120-126.

Knight, J. C. and N. G. Leveson (1986), "*An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming*", IEEE Transactions on Software Engineering 12(1), pp: 96-109.

Littlewood, B. (1981), "*Stochastic Reliability Growth: a Model for Fault-Removal in Computer Programs and Hardware Designs*", IEEE Transactions on Reliability R-30(4), pp: 313-320.

Littlewood, B. and D. R. Miller (1989), "*Conceptual Modelling of Coincident Failures in Multi-Version Software*", IEEE Transactions on Software Engineering 15(12), pp: 1596-1614.

Littlewood, B., P. Popov, et al. (2000), "*Assessment of the Reliability of Fault-Tolerant Software: a Bayesian Approach*", *in proc.* SAFECOMP-2000, Rotterdam, the Netherlands, Springer, pp: 294-308.

Littlewood, B., P. Popov and L. Strigini (2001), "*Modelling software design diversity - a review*", ACM Computing Surveys 33(2), pp: 177-208.

Lyu, M. R., Ed. (1995), "*Software Fault Tolerance*", Trends in Software, Wiley.

Lyu, M. R., Ed. (1996), "*Handbook of Software Reliability Engineering*", McGraw-Hill and IEEE Computer Society Press.

Marchetti, C., R. Baldoni, et al. (2006), "*Fully Distributed Three-Tier Active Software Replication*", IEEE Transactions on Parallel Distributed Systems 17(7), pp: 633-645

Musa, J. D. (1993), "*Operational Profiles in Software-Reliability Engineering*", IEEE Software (March), pp: 14-32.

Popov, P. and B. Littlewood (2004), "*The effect of testing on the reliability of fault-tolerant software*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '04), Florence, Italy, IEEE Computer Society Press, pp: 265-274.

Pullum, L. (2001), "*Software Fault Tolerance Techniques and Implementation*", Artech House.

ReSIST (2006), "*D12-Resilience-Building Technologies: State of Knowledge*", http://www.laas.fr/RESIST/deliverables.html.

Strigini, L. (2005), "*Fault Tolerance Against Design Faults*", *in* Dependable Computing Systems: Paradigms, Performance Issues, and Applications, H. Diab and A. Zomaya (Eds.), J. Wiley & Sons, pp: 213-241.

Tai, A. T., J. F. Meyer, et al. (1996), "*Software Performability: from Concepts to Applications*", Boston / Dordrecht / London, Kluwer Academic Publishers.

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

III. Research Overview

1. Introduction

The main parts of this thesis, namely chapters IV to VI, will be given as a collection of papers. The papers deal with specific aspects of the research conducted. The purpose of this chapter is to give an overview of how the different parts of the research outlined in the papers are linked together. It will also give brief summaries of each of the papers that follow in the subsequent chapters, state the differences between the papers and provide a guide on the reading sequence of their contents.

2. Research overview

A summary of motivations, the work done and the contributions of this thesis were already provided in the Introduction chapter. This section will give an overview of how the different parts of the thesis are linked together.

The work detailed in this thesis began as part of the EPSRC (Engineering and Physical Sciences Research Council) sponsored research project "Diversity with Off-The-Shelf components (DOTS)" (DOTS 2000-2004). Even though the project had several stated aims and objectives (see the project summary for full details (DOTS 2000-2004)), the two main aims of this project were to explore the dependability benefits that may be gained through the use of diverse OTS products and the architectural issues that enable their effective use: this thesis details research done on these two strands. OTS DBMS products were the family of products chosen to study these issues: these are very complex products that are widely used and yet with many reported bugs in each release.

Employing diversity with OTS products requires strong evidence that diversity will provide protection against those types of failures that non-diverse solutions will not. There are comparatively few studies that have explored this issue for DBMS products (Lee and Iyer 1995), (Chandra and Chen 2000). The difficulty is compounded by the lack of available dependability data from the vendors. Detailed evidence about DBMS product failure rates would be the most useful for reliability analysis, but this data may not even be available to the vendors themselves (especially the open-source ones). The only direct dependability evidence that is available for these products are the fault (bug) reports. This is the evidence that was explored in this study to check for initial estimates of

dependability benefits of diversity with OTS DBMS products. A fault report details the triggering conditions that are required for the failure to be manifested as well as the details of the failure that the user observed. In a first study, 181 bug reports for four DBMS products were collected (both open-source and commercial). Each bug script (which is contained in the bug report) was run on both the DBMS product for which the bug was reported and (when possible) on the other DBMS products. The number of bugs that caused failures in more than one DBMS product was very low, and none of the bugs caused failures in more than two products. However, since the results of the first study concerned only a specific snapshot in the evolution of these products a second study with the open-source products was conducted. The results substantially confirmed those of the first study: very few bugs cause coincident failures. All of these results will be detailed in the two papers in chapter IV.

These results pointed to serious dependability gains that may be obtained from diversity and provided plenty of justification to explore the architectural solutions that would enable employing diversity with OTS DBMS products. Various trade-offs were identified depending on the amount of failure diversity and available functionality that a user may require for a specific installation. For example, using successive releases of the same product for fault tolerance appeared to bring substantial dependability benefits for one of the open-source DBMS products: most of the old bugs had been fixed in the new release; many of the newly reported bugs did not cause failure (or could not be run at all) in the old release. Another study was conducted on the benefits of data diversity (Ammann and Knight 1988). A small number of generic "rephrasing" rules were found to be useful workarounds for a number of the known bugs of the open-source DBMS products. All of these architectural details of a fault-tolerant server constructed from DBMS products will be detailed partly in chapter IV and in the two papers given in chapter V.

The evidence uncovered with the bug reports pointed to potential for significant dependability gains to be had from employing diversity. However they are not definitive evidence. It was therefore important to point out to what extent these results allow for dependability estimates to be obtained. Research in this area concentrated on, first, clarifying the limitations that an assessor will encounter if he/she wishes to obtain estimates of dependability gains that may be achieved from diversity with OTS products

by using bug reports data as evidence. Second, exploring to what extent the existing models that are used for estimating the reliability of diverse bespoke software systems (Littlewood, Popov et al. 2000) as well as the reliability growth modelling of non-diverse bespoke software systems (Littlewood 1981) may be used and or/extended for assessment of a diverse fault-tolerant SQL server. And third, how can the bug reports be best utilised as empirical evidence in these models. Discussions about the limitations of the bug reports as evidence in dependability estimation are discussed in both papers of chapter IV. The model extensions and the use of bug reports data as evidence in the assessment are detailed at length in the first two papers in chapter VI. The last paper in chapter VI contains details of how an existing model (Littlewood, Popov et al. 2000) may be adapted to assess a single product when both the products correctness and timeliness are of interest.

3. Summary of the papers

This section contains a brief summary of each of the papers that are given in the chapters IV, V and VI. A few general points that apply to all of the papers are:

- Each of the papers presented in this thesis contains the same text as their published/submitted version. The main differences are:
  - the format of the references: in the published/submitted versions of the papers the references were numbered; in this thesis, to improve the readability, the references are annotated, i.e. (*author year*).
  - numbering of tables, figures, footnotes and equations: to improve readability, tables, figures, footnotes and equations are numbered globally in the thesis, rather than the numbering being local to each paper.
- The papers will be numbered in order of appearance in the thesis, and the title will follow the paper identifier (e.g. ***Paper-1*** – Fault Diversity with Off-The-Shelf SQL Database Servers)
- Each paper may contain among its references another paper or Appendix that forms part of this thesis. When this is the case a brief note to inform the reader is written in ***bold italic underlined*** text next to the first occurrence of the reference

in the respective paper (e.g. (Gashi, Popov et al. 2004b) (***the preceding reference forms part of this thesis as Paper-1***)).

- Since the papers were written at different times during the evolution of the study with the bug reports, they may contain minor differences in the results that they present. These differences are also highlighted and explained in this section.

3.1 Fault diversity study (Chapter IV)

The first paper is titled "*Fault Diversity among Off-The-Shelf SQL Database Servers*" (Gashi, Popov et al. 2004b) and is co-authored with Dr Peter Popov and Prof. Lorenzo Strigini. The paper forms part of the proceedings of the IEEE DSN-04 (Dependable Systems and Networks) conference which was held in Florence, Italy in July 2004. This paper details the results of the first study with the bug reports of four DBMS products, namely PostgreSQL 7.0.0, Interbase 6.0, Microsoft SQL Server 7 and Oracle 8.0.5. It provides details of the definitions that were used to classify the bugs and the failures that these bugs caused, as well as a detailed discussion of the more "interesting" bugs that caused coincident failures. This paper will be referred to as *Paper-1*.

The second paper is titled "*Fault Tolerance via Diversity for Off-The-Shelf Products: a Study with SQL Database Servers*" (Gashi, Popov et al. 2007) and is also co-authored with Dr Peter Popov and Prof. Lorenzo Strigini. The paper was accepted (with minor revisions) by the IEEE TDSC (Transactions on Dependable and Secure Computing) on the 30th of January 2007, and the current text present in this thesis is that of the revision submitted to IEEE TDSC on the 9th of March 2007. The final acceptance of the paper and the publication date are yet to be confirmed. It provides details of the second study with the bugs of the later releases of two DBMS products used in the first study, namely PostgreSQL 7.2 and the Firebird 1.0 (which is the open-source descendent of Interbase 6.0). It also contains the analysis of fault diversity between releases of the same product. An updated summary of the results of the first study (without providing the details for the coincident failures, which are given in Paper-1 above) as well as an updated list of definitions of the classification of bugs and failures is also given. Finally, it also contains a comprehensive description of the architecture of a diverse fault-tolerant SQL server. This paper will be referred to as *Paper-2*.

3.1.1 Differences between the papers

Both papers present empirical results from the studies with bug reports of DBMS products. The paper published in DSN-04 (Gashi, Popov et al. 2004b) was written before the paper that was accepted in IEEE TDSC (Gashi, Popov et al. 2007). Therefore Paper-1 contains only the preliminary results of the first study. Subsequent analysis was also done with the bug reports which meant that the results changed slightly (even though the total number of coincident failures remained the same). These differences are due to some bugs which were classified as "further work" in Paper-1 (e.g. non-trivial translation of the bugs script from one dialect of SQL to another, incomplete bug scripts etc.) which prevented the bug being reproduced in one (or more) of the DBMS products; these issues were subsequently resolved and the updated results are shown in Paper-2, together with the results of the second study. Therefore the reader should be aware that the results in the Tables 1 and 3 (in Paper-1), even though they are based on the same raw data, differ slightly from the updated results presented in Tables 5 and 6 (in Paper-2).

Other differences between the papers are in the terminology used:

- In Paper-1 the term "server" is invariably used to refer to the DBMS products. In Paper-2 the more precise term "DBMS product" was used due to the ambiguity of the unqualified term "server" (e.g. web server, database server, mail server etc).
- Terminology definitions in Sections 4.1 and 4.3 of Paper-1 have been slightly modified and some new terms are defined in Section 3.1 of Paper-2:
  - Instead of using the more generic (but also more ambiguous) term "*Heisenbug*" a new term named "*Unreproduced*" has been defined for those bug reports that did not cause failures when the bugs script was run on the same DBMS product that the bug was reported.
  - Performance failures are defined more precisely in Paper-2
  - *Divergent* and *Non-Divergent* failure definitions are introduced in Section 3.1.2 of Paper-2, which did not exist in Paper-1.

3.1.2 Suggested reading sequence

Paper-1 presents a detailed account of the first study therefore it can be read as a whole.

Paper-2 contains a detailed sub-section on the proposed architecture of a fault-tolerant database server constructed from diverse DBMS products. The architectural issues will also be detailed in the two papers that form part of chapter V, therefore the reader is advised to postpone reading Section 2 of Paper-2 until chapter V. Section 3 of Paper-2 contains an overview of the updated results of the first study as well as detailed results of the second study. Section 4 is a summarised discussion of the implications of the observed results: a more comprehensive discussion is provided in Section 6 of Paper-1. Section 5 of Paper-2 contains a review of related work.

In summary, the reader is advised to read Paper-1 first as a whole, followed by Paper-2 (specifically Sections 1, 3, 4, 5 and 6 of Paper-2).

3.2 Architectural aspects of a fault-tolerant diverse SQL server (Chapter V)

This chapter contains two papers that detail the architecture of a diverse fault-tolerant SQL server. Additionally the reader should also read Section 2 of Paper-2 detailed in chapter IV.

The first paper is titled "*On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*" (Gashi, Popov et al. 2004a) and is co-authored with Dr Peter Popov, Mr. Vladimir Stankovic and Prof. Lorenzo Strigini. The paper forms a chapter of the book "Architecting Dependable Systems II " edited by Rogerio de Lemos, Christina Gacek and Alexander Romanovsky which is published by Springer-Verlag as part of the Lecture Notes on Computer Science Series (volume 3069), and was published in autumn 2004. This paper contains a summary of the results of the first study with the bugs (which were presented in greater detail in Paper-1) and a comprehensive description of the architecture of a diverse fault-tolerant server (containing some additional topics such as replica determinism which were only briefly summarised in Section 2 of Paper-2). This paper will be referred to as *Paper-3*.

The second paper is titled "*Rephrasing Rules for Off-The-Shelf SQL Database Servers*" (Gashi and Popov 2006) and is co-authored with Dr. Peter Popov. The paper forms part of the proceedings of IEEE EDCC-06 (European Dependable Computing Conference) which was held in Coimbra, Portugal in October 2006. This paper details the architectural

aspects of SQL rephrasing, rephrasing rules that have been defined and results of applying these rules to the bugs that were collected for the open-source DBMS products. This paper will be referred to as *Paper-4*.

3.2.1 Differences between the papers

Both papers in chapter V (as well as Section 2 of Paper-2 in chapter IV) contain descriptions (to varying levels of detail) of the architecture of a diverse fault-tolerant SQL server. Section 2 of Paper-2 in chapter IV contains a comprehensive description of the architecture with references to more recent work on database replication solutions. Paper-3 contains an older version of this description, but more detail and examples on issues of data consistency, the differences in the SQL dialects between the DBMS products, replica determinism and data diversity (detailed in Sections 3.3 to 3.6 of Paper-3). Paper-4 contains a brief overview of the architecture before concentrating in greater detail on the data diversity aspects of the architecture (detailed in Sections 2.2 and 2.4 of Paper-4).

There is a difference in the terminology used in the papers of chapter V:

- In Paper-3 and Paper-4 the term "server" (also "O-server" in Paper-4) is invariably used to refer to the DBMS products. In Section 2 of Paper-2 in chapter IV the more precise term "DBMS product" is used due to the ambiguity of the unqualified term "server" (e.g. web server, database server, mail server etc).
- The figure depicting the fault-tolerant server differs slightly in the level of detail and the notation used between the three papers. In Section 2 of Paper-2 in chapter IV and in Paper-3 a more "neutral" terminology was used; in Paper 4 a UML (Unified Modelling Language (OMG 2007)) Deployment Diagram was used to depict the architecture. However the constituent parts of the architecture remain the same.

3.2.2 Suggested reading sequence

Section 2 of Paper-2 (Gashi, Popov et al. 2007) in chapter IV presents a comprehensive overview and description of the architecture of a fault-tolerant SQL server, therefore the reader is advised to read that section first.

The reader may skip the following sections of the two papers in chapter V:

- Section 2 of Paper-3: it contains a summary of the results of the study with the bugs which were presented in much greater detail in Paper-1.
- Sections 3.1 and 3.2 of Paper-3 and Sections 2.1 and 2.3 of Paper-4: they contain an overview of the architecture of the fault-tolerant SQL server which is already described in greater detail in Section 2 of Paper-2 in chapter IV.
- Sections 3.7 and 4 in Paper-3: they detail experiments that measure the performance benefits of diversity using the TPC-C benchmark. These experiments were carried out by Mr. Vladimir Stankovic and they do not form part of this thesis.

Sections 3.3 to 3.6 of Paper-3 are recommended to the reader as they contain descriptions of some specific issues of the diverse fault-tolerant SQL server, as was mentioned before. Paper-4 can be read in full (apart from Sections 2.1 and 2.3 mentioned above) as it contains further details of the data diversity extensions of the architecture of the fault-tolerant SQL server, as well as examples of the rephrasing rules and results from applying them to the bug reports of the open-source DBMS products.

In summary, the reader is advised to first read Section 2 of Paper-2 in chapter IV, followed by Sections 3.3 to 3.6, 5, 6, 7 of Paper-3 and the complete Paper-4 (apart from Sections 2.1 and 2.3).

3.3 Optimal selection of COTS components (Chapter VI)

This chapter contains three papers that detail the research conducted on the interpretation of the results of the studies with the bugs. Two previous models, developed by colleagues at the Centre for Software Reliability (CSR), have been extended to enable their use, utilising the results of the bug studies, for optimal selection of either a diverse DBMS product pair or a single DBMS product.

The first paper is titled "*Uncertainty Explicit Assessment of Off-the-Shelf Software: Selection of an Optimal Diverse Pair*" (Gashi and Popov 2007) and is co-authored with Dr. Peter Popov. The paper forms part of the proceedings of the IEEE ICCBSS-07 (International Conference on COTS-Based Software Systems) which was held in Banff, Alberta, Canada in February 2007. This paper details an extension of a previous model developed at CSR (Littlewood, Popov et al. 2000) and illustrates its use for selection of

an optimal pair of products using the data from the first study with the bugs. This paper will be referred to as *Paper-5*.

The second paper is titled "*Reliability Growth Modelling of a 1-Out-Of-2 System: Research with Diverse Off-The-Shelf SQL Database Servers*" (Bishop, Gashi et al. 2007) and is co-authored with Prof. Peter Bishop, Prof. Bev Littlewood and Dr. David Wright. The text in the thesis is the current version of a paper being prepared for submission to IEEE ISSRE-07 (International Symposium on Software Reliability Engineering) the deadline for which is 16-April-2007. This paper details two approaches that were studied to construct reliability models for a 1-out-of-2 fault-tolerant SQL server utilising the data from the studies with the bugs:

- in the first approach, the Littlewood model (Littlewood 1981) was extended for reliability growth modelling of a 1-out-of-2 fault-tolerant SQL server.
- the second approach attempts to get away from the need to quantify actual usage time of the SQL servers and uses just the available bug counts to make predictions regarding the likely improvements in reliability that may be expected from the use of a diverse server with two DBMS products instead of a non-diverse setup.

This paper will be referred to as *Paper-6*.

The third paper is titled "*Uncertainty Explicit Assessment of Off-The-Shelf Software*" (Gashi, Popov et al. 2006) and is co-authored with Dr. Peter Popov and Mr. Vladimir Stankovic. The paper was submitted for publication to the Elsevier Journal of Information and Software Technology (JIST) on 19-Dec-2006. The paper is currently under review. This paper illustrates how the model developed previously at CSR (Littlewood, Popov et al. 2000) can be extended to asses a single COTS product in terms of both its correctness and timeliness. The assessor can then use the results of the assessment to select the most optimal product for a given setup. The mathematical details of the model and its extensions are identical to what is presented in Paper-5. This paper will be referred to as *Paper-7*.

3.3.1 Differences between the papers

All three papers utilise the results from the studies with the bug reports of the DBMS products, though the way in which the bug reports data are used differs between the papers, depending on the capabilities of the models and their assumptions.

Mathematically the models presented in Paper-5 and in Paper-7 are identical. However the definitions of the model parameters differ in the two papers:

- In Paper-5 the model is used for choosing an optimal diverse *pair* of DBMS products from the viewpoint of correctness.
- In Paper-7 the model is used for choosing an optimal *single* DBMS product from two viewpoints (attributes): correctness and timeliness.

Note that the results for the commercial DBMS products had to be anonimised in Paper-7 due to restrictive "End User License Agreements" on reporting performance-related data. However permission was obtained from the two vendors of commercial DBMS products used in the studies with the bug reports, and the empirical results in Paper-7 which are relevant to this thesis are based on the results of the studies with the bug reports. Therefore it can be divulged that the anonimised *CS1* product in Paper-7 is Oracle 8.0.5 and the anonimised *CS2* product is Microsoft SQL Server 7.

3.3.2 Suggested reading sequence

The reader is advised to read all three papers in full. As described above, the mathematical details of the models presented in Paper-5 and Paper-7 are identical, but the parameters of the model are defined differently in the two papers. One small note:

- Section 4.1 of Paper-7 was written by Mr Vladimir Stankovic and therefore will not form part of this thesis.

The ordering in which the papers should be read is Paper-5, Paper-6 and then Paper-7 (apart from Section 4.1 of Paper-7).

References

Ammann, P. E. and J. C. Knight (1988), "*Data Diversity: An Approach to Software Fault Tolerance*", IEEE Transactions on Computers 37(4), pp: 418-425.

Bishop, P., I. Gashi, B. Littlewood and D. Wright (2007), "*Reliability Growth Modelling of a 1-Out-Of-2 System: Research with Diverse Off-The-Shelf SQL Database Servers*", *in proc.* Int. Symp. on Software Reliability Engineering (ISSRE '07), Trollhattan, Sweden, IEEE Computer Society Press, pp: to be submitted for publication.

Chandra, S. and P. M. Chen (2000), "*Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '00), NY, USA, IEEE Computer Society Press, pp: 97-106.

DOTS (2000-2004), "*Diversity with off-the-shelf components (DOTS) project*", Centre for Software Reliability of City University and University of Newcastle-upon-Tyne, http://www.csr.ncl.ac.uk/dots/.

Gashi, I. and P. Popov (2006), "*Rephrasing Rules for Off-The-Shelf SQL Database Servers*", *in proc.* 6th European Dependable Computing Conf. (EDCC '06), Coimbra, Portugal, IEEE Computer Society Press, pp: 139-148.

Gashi, I. and P. Popov (2007), "*Uncertainty Explicit Assessment of Off-the-Shelf Software: Selection of an Optimal Diverse Pair*", *in proc.* Int. Conf. on COTS-Based Software Systems (ICCBSS '07), Banff, Alberta, Canada, IEEE Computer Society Press, pp: 93-102.

Gashi, I., P. Popov and V. Stankovic (2006), "*Uncertainty Conscious Assessment of Off-The-Shelf Software*", Submitted for publication, http://www.csr.city.ac.uk/people/ilir.gashi/COTS/.

Gashi, I., P. Popov, V. Stankovic and L. Strigini (2004a), "*On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*", *in* Architecting Dependable Systems II, R. de Lemos, Gacek, C., Romanovsky, A. (Eds.), Springer-Verlag, 3069, pp: 191-214.

Gashi, I., P. Popov and L. Strigini (2004b), "*Fault Diversity Among Off-The-Shelf SQL Database Servers*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '04), Florence, Italy, IEEE Computer Society Press, pp: 389-398.

Gashi, I., P. Popov and L. Strigini (2007), "*Fault tolerance via diversity for off-the-shelf products: a study with SQL database servers*", IEEE Transactions on Dependable and Secure Computing, to appear.

Lee, I. and R. K. Iyer (1995), "*Software Dependability in the Tandem GUARDIAN System*", IEEE Transactions on Software Engineering 21(5), pp: 455-467.

Littlewood, B. (1981), "*Stochastic Reliability Growth: a Model for Fault-Removal in Computer Programs and Hardware Designs*", IEEE Transactions on Reliability R-30(4), pp: 313-320.

Littlewood, B., P. Popov and L. Strigini (2000), "*Assessment of the Reliability of Fault-Tolerant Software: a Bayesian Approach*", *in proc.* Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP '00), Rotterdam, the Netherlands, Springer, pp: 294-308.

OMG (2007), "*Unified Modeling Language (UML), version 2.1.1*", http://www.omg.org/technology/documents/formal/uml.htm.

IV. Fault Diversity Study

Paper-1. Fault Diversity among Off-The-Shelf SQL Database Servers

Abstract: *Fault tolerance is often the only viable way of obtaining the required system dependability from systems built out of "off-the-shelf" (OTS) products. We have studied a sample of bug reports from four off-the-shelf SQL servers so as to estimate the possible advantages of software fault tolerance - in the form of modular redundancy with diversity - in complex off-the-shelf software. We checked whether these bugs would cause coincident failures in more than one of the servers. We found that very few bugs affected two of the four servers, and none caused failures in more than two. We also found that only four of these bugs would cause identical, undetectable failures in two servers. Therefore, a fault-tolerant server, built with diverse off-the-shelf servers, seems to have a good chance of delivering improvements in availability and failure rates compared with the individual off-the-shelf servers or their replicated, non-diverse configurations.*

Co-authors: Dr. Peter Popov, Prof. Lorenzo Strigini
Conference: Dependable Systems and Networks 2004 (DSN-04)
Date of submission: December-2003
Status: *Published*
Number of reviewers: 5
Publication date: July-2004

Full citation: Gashi I., Popov P., Strigini L., "*Fault diversity among off-the-shelf SQL database servers*", Proc. *DSN 2004, International Conference on Dependable Systems and Networks*, Florence, Italy, IEEE Computer Society Press, pp:389-398, 2004

1. Introduction

When systems are built out of "off-the shelf" (OTS) products, fault tolerance is often the only viable way of obtaining the required system dependability (Popov, Strigini et al. 2000), (Valdes, Almgren et al. 1999), (Hiltunen, Schlichting et al. 2000). Fault tolerance may take multiple forms, from simple error detection and recovery add-ons (e.g. wrappers (Popov, Strigini et al. 2001)) to full-fledged "diverse modular redundancy" (Laprie, Arlat et al. 1990): replication with diverse versions of the components. Even this latter class of solutions becomes affordable with many OTS products and has the advantage of a fairly simple architecture. The cost of procuring two or even more OTS products (some of which may be free) would still be far less than that of developing one's own.

All these design solutions are well known from the literature. The questions, for the developers of a system using OTS components, are about the dependability gains, implementation difficulties and extra cost that they would bring for that specific system.

To study the issues for a realistic category of OTS products we have chosen SQL database servers. These are complex products, with many faults in each release, and even features that imply an accepted possibility of an incorrect behaviour, albeit rare. An example of the latter is the known "write skew" (Berenson, Bernstein et al. 1995) problem with some optimistic concurrency control architectures (Fekete, Liarokapis et al. 2000). Further dependability improvement of OTS SQL servers seems only possible if fault tolerance through design diversity is used (Gray 2000). Given the many available OTS SQL servers and the standardisation of their functionality (SQL 92 and SQL 99), it seems reasonable to build a fault-tolerant SQL server from available OTS servers.

The effort of developing an SQL server using design diversity (e.g. several of-the-shelf SQL servers and suitably adapted "middleware" for replication management) would require strong evidence of its usefulness: for example empirical evidence that likely failures of the SQL servers, which may lead to serious consequences, are unlikely to be tolerated without diversity. This paper starts to investigate such empirical evidence. We seek to demonstrate whether design diversity has a potential to deliver significant improvement of dependability of SQL servers, compared to solutions for data replication

that can only tolerate crash failures. To this aim we are running experiments to determine the dependability gains achieved through fault tolerance.

A preliminary evaluation step concerns *fault* diversity rather than *failure* diversity. By manual selection of test cases, one can check whether the diverse redundant configuration would tolerate the known bugs in the repositories of bugs reported for the various OTS servers. We have conducted a study on four SQL servers, both commercial and open-source. We collected known bug reports for these servers. For each bug, we took the test case that would trigger it and ran it on all four servers (if possible), to check for coincident failures. We found the number of coincident failures to be very low.

We use the following terminology. The known bugs for the OTS servers are documented in bug report repositories (i.e. bug databases, mailing lists etc). Each *bug report* contains the description of what the bug is and the *bug script* (SQL code that contains the failure triggering conditions) required to reproduce the *failure* (the erroneous output that the reporter of the bug observed). In our study we collected these bug reports and *ran* the bug scripts in the servers (we will use the phrase "*running a bug*" for the sake of brevity).

This paper is structured as follows: In Section 2 we describe the background and motivation of the study and related work from the literature. In Section 3 we describe how the study was conducted and the terminology for classification of faults. In Section 4 we present the quantitative results obtained. In Section 5 we describe the bugs that caused coincident failures. In Section 6 we discuss the possible reliability gains to be had from using diverse OTS SQL servers and in Section 7 we present conclusions and possible further work.

2. Background and related work

2.1 Fault tolerance in databases

Software fault tolerance has been thoroughly studied and successfully applied in many sectors, including databases. For example, standard database mechanisms such as transaction "rollback and retry" and "checkpointing" can be used to tolerate faults that are due to transient conditions. These techniques can be used with or without data replication in the databases.

There are many solutions for data replication (Bernstein, Hadzilacos et al. 1987), (Weismann, Pedone et al. 2000), (Pedone and Frolund 2000) as a feature of many commercial SQL servers or as middleware that can be used with a variety of SQL servers. Typically, these replication solutions work with sets of identical servers. Jimenez-Peris et al (Jimenez-Peris and Patino-Martinez 2003) present a relevant discussion of the various ways in which database replication with OTS servers can be organised, namely treating the servers as white, grey or black boxes. All commercial offerings are of the white-box kind, where code necessary for replication is added inside the server product. The grey-box approach, as implemented in (Jimenez-Peris, Patino-Martinez et al. 2002), assumes that servers provide specific services to assist with replication. The black-box approach uses the standard interfaces of the servers. Both the grey and black box approaches are implemented via middleware on top of the existing servers. To the best of our knowledge, a common assumption is made in the known replication solutions that the SQL servers will fail in a "fail-stop" manner (Schneider 1984), with detectable clean crashes, and leaving a copy of a correct state for use in recovery. Apart from simplifying the protocols for data replication, the assumption of crash failures also allows for some performance optimisation such as executing the modifying queries on a single server, which then propagates the updates to all other servers involved in the replication, a solution considered adequate by the standardising bodies (Sutter 2000).

These approaches have shortcomings, i.e., they do not protect against failures that are not easily detectable (non-fail-stop), and incorrect updates would be propagated to all the replicas. Using diverse SQL servers instead of servers of the same type would improve error detection and thus reduce the risk from incorrect results. Availability could also be improved because servers that are diagnosed as correct can continue operation while recovery is performed on the faulty server[s]. Elsewhere (Popov, Strigini et al. 2004), (Gashi, Popov et al. 2004) (***the preceding reference forms part of this thesis as Paper-3***) we describe some initial steps toward implementing middleware for data replication with diverse SQL servers. There, we also discuss some difficulties of data replication with diverse servers, such as the need to use the subset of SQL that is common to all servers used, and to translate all queries into the SQL "dialects" of these servers.

2.2 Studies of faults and failures

The usefulness of diversity depends on the frequency of those failures that cannot be tolerated without it. There have been comparatively few related studies.

Gray studied the TANDEM NonStop system (Gray 1986) and observed that over an (unspecified) measured period only one out of 132 faults caused failures deterministically, i.e. the same failure was observed on retry. Gray calls these "Bohrbugs". The others, which he calls "Heisenbugs" only caused failures under special conditions (e.g. created by a combination of the state of the operating system and other software), difficult to reproduce artificially. Heisenbugs – so long as their failures are detected – can be tolerated by replication without diversity, as in the Tandem system. A later study, (Lee and Iyer 1995) of field software failures for the Tandem Guardian90 operating system found that 82 % of the reported field software faults were tolerated. However, 18 % of the faults did lead to both non-diverse processes in a Tandem process failing and therefore leading to a system failure.

Related studies exist on determinism and fail-stop properties of database failures, but they, like our study, concern faults rather than failure measurements. A study (Chandra and Chen 2000) examined fault reports of three applications (Apache Web server, GNOME and MySQL server). Only a small fraction of the faults (5-14%) were Heisenbugs triggered by transient conditions that would be tolerated by a simple "rollback and retry" approach. However the reason why there are few Heisenbugs here, and indeed in our study, might be that people are less likely to report faults that they cannot reproduce, and this is acknowledged by the authors in (Chandra and Chen 2000). In another study (Chandra and Chen 1998) the same authors found (via fault injection) that a significant number of faults (7%) violated the fail-stop model by writing incorrect data to stable storage. Even though they report that this number falls to 2% when applying the Postgres95 transaction mechanism, this number still remains high for applications with stringent reliability requirements.

2.3 Diversity with off-the-shelf applications

Other researchers have also considered the potential of diversity for improving the dependability of OTS software. Various architectures have been proposed that use

diversity for intrusion tolerance: e.g. HACQIT (Reynolds, Just et al. 2002), which demonstrates diverse replication (with two OTS web servers - Microsoft's IIS and Apache web server) to detect failures (especially maliciously caused ones) and initiate recovery; SITAR (Wang, Gong et al. 2001), an intrusion tolerant architecture for distributed services and especially COTS servers; or the Cactus architecture (Hiltunen, Schlichting et al. 2000), intended to enhance survivability of applications which support diversity among application modules.

Another example (Adobe 2004) uses diverse Java virtual machines for interoperability rather than for tolerating failures.

3. Description of the study

3.1 Bug reports

Two commercial (Oracle 8.0.5 and Microsoft SQL Server 7 (without any service packs applied)) and two open-source (PostgreSQL Version 7.0.0 and Interbase Version 6.0), SQL servers were used in this study. Interbase, Oracle and MSSQL were all run on the Windows 2000 Professional operating system, whereas PostgreSQL (which is not available for Windows) was run on RedHat Linux 6.0 (Hedwig).

We only used bugs that caused failure of a server's *core engine*. We did not consider other bugs such as those that caused failure to a client application tool or various connectivity API's (JDBC/ODBC etc.), because these functions in a future fault tolerant architecture would be provided by the middleware.

For each of these servers there is an accessible repository of reports of known bugs. We collected: Interbase bugs (SourceForge) reported in the period between August 2000 and August 2001; PostgreSQL bugs (PostgreSQL) reported between May 2000 and January 2001; Oracle bugs (Oracle) reported between September 1998 and December 2002. Bug reports for MSSQL (Microsoft) do not specify dates; we used all reports for both MSSQL 7 and MSSQL 2000, available as of August 2003, that included "bug scripts" and were core engine bugs. For Oracle and MSSQL we collected reports from longer periods, because for these two servers (both "closed development" servers) some reports do not include bug scripts and we could not check whether the bug was present in other servers.

By extending the collection period we obtained reasonably large (though obviously imperfect) samples of bug reports. Despite this, the sample that we could use for Oracle contained only 18 bugs, since most reports omitted the bug scripts.

For each reported bug we attempted to run the corresponding bug script. Full details are available in (Gashi 2003) (***and also provided as Appendix A of this thesis***).

3.2 Reproducibility of failures

All these servers offer features that are extensions to the basic SQL standard, and these extensions differ between the servers. Bugs affecting one of these extensions thus literally cannot exist in a server that lacks the extension. We called these "dialect-specific" bugs. For example, Interbase bug 217138 (Gashi 2003) uses the UNION operator in views, which PostgreSQL 7.0.0 views do not offer, and thus cannot be run in PostgreSQL: it is a dialect-specific bug.

Another "reproducibility" issue arises when a bug script does not cause failure in the server for which the bug was reported. We called these bugs Heisenbugs, borrowing Gray's terminology (Gray 1986). We intend to run the Heisenbugs again in a more stressful simulated environment (Popov, Strigini et al. 2004) (with multiple clients and large number of transactions) to see whether repeated trials will give incorrect results.

4. Quantitative results

4.1 Detailed results

In total we included in the study 181 bug reports: 55 for Interbase, 57 for PostgreSQL, 51 for MSSQL and 18 for Oracle. Out of these 181 bugs, 76 were "dialect-specific" (could be run in only one of the four servers); 47 could be run in all four servers; 26 could be run in only two servers and 32 in only three servers.

Each bug was first run on the server for which it was reported, and (after translating the script into the SQL dialect of the respective server) on the other servers. The bugs were classified into dialect-specific and non-dialect-specific bugs; the latter were then further classified into Bohrbugs or Heisenbugs as explained previously. The failures were also

classified into different categories according to their effects, as different failure types require different recovery mechanisms:

Engine Crash failures: crashes or halts of the core engine.

Incorrect Result failures: incorrect outputs without engine crashes: the outputs do not conform to the server's specification or to the SQL standard.

Performance failures: correct output, but with an unacceptable time penalty for the particular input.

Other failures.

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.

Table 1 contains the results of this step of the study. Each grey column lists the results produced when the bugs reported for a certain server were run on that server. For example, we collected 55 known Interbase bugs, of which, when run on our installation of the Interbase server, 8 did not cause failures (possible Heisenbugs). The 47 bugs that caused failures are further classified in the part of the column below the double vertical lines, after the "Failure observed" row. All the performance failures and all the engine crashes are self-evident. Incorrect Result failures and "Other" failures can be self-evident or non-self-evident depending on whether the server gives an error message.

The three columns to the right of the grey one present the results of running the Interbase bugs on the other three servers. For example, we can see that 23 of the Interbase bugs cannot be run in PostgreSQL (dialect-specific bugs). Then we have the bugs that "require further work": this means that we have not managed yet to translate the bug script in the PostgreSQL dialect of SQL, or are listed as "performance bugs" but we could not decide whether performance improves by changing servers. We plan to resolve this uncertainty via a testing infrastructure (Popov, Strigini et al. 2004) to measure the precise execution times of the queries.

Out of 55 Interbase bugs we managed to run 27 in PostgreSQL; only one caused a failure in both Interbase and PostgreSQL. This particular failure was a non-self-evident incorrect result as can be seen from the table.

As for the failure types, we can see that most of the bugs cause incorrect result failures. This will be discussed further in the Section 6.

We observed a higher number of Heisenbugs in MSSQL and Oracle than in the other servers. This was documented by some of the bug reports, which indicated: "may cause a failure".

Table 1 - Results of running the bug scripts on all four servers. *IB* stands for Interbase, *PG* for PostgreSQL, *OR* for Oracle and *MS* for MSSQL

			IB	PG	OR	MS	**PG**	IB	OR	MS	**OR**	IB	MS	PG	**MS**	IB	OR	PG
Total bug scripts			55	55	55	55	57	57	57	57	18	18	18	18	51	51	51	51
Bug 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 bug 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

4.2 Summary of observed fault diversity

Table 2 contains a summary from the viewpoint of the probable effects on a fault-tolerant server. Of the 47 bugs that could be run on all four servers, 12 did not cause failures in any of the servers: they are Heisenbugs for the server for which they were reported, and non-existent or Heisenbugs for the other three servers. 31 of these only caused a failure in the server for which they were reported and not in the others; and 4 bugs caused a coincident failure in two servers.

Table 2 - The number of bug scripts run and the effects on different combinations of servers

The server(s) in which the bug script was run	IB, PG, OR, MS	IB, PG, OR only	IB, PG, MS only	IB, OR, MS only	PG, OR, MS only	IB, PG Only	IB, MS Only	IB, OR Only	PG, OR Only	PG, MS Only	MS, OR Only	IB Only	PG Only	MS Only	OR Only
Total number of bug scripts run	47	3	7	12	10	5	3	0	4	12	2	17	18	28	13
Failure not observed in any server	12	0	1	2	0	0	0	0	0	0	1	1	2	5	3
Failure observed in one server only	31	3	6	9	9	5	3	0	3	7	1	16	16	23	10
Failure observed in two servers	4	0	0	1	1	0	0	0	1	5	0	N/A	N/A	N/A	N/A
None of the bugs caused a failure in more than two servers															

In addition to these 47, we have many bugs that could be run only on a subset of the four servers and thus on a fault-tolerant server built out of this subset. The following sections in the table show the number of bugs that could be run in each of these different combinations (4 three-version combinations and 6 two-version combinations), and how many caused failures or coincident failures.

The last four columns show the 76 dialect-specific bugs, which could only be run in the server for which they were reported and therefore affect functionality that would not be available on a fault-tolerant diverse server.

4.3 Two-version combinations

We now look more closely at the two-version combinations of the four different servers in our study, to see 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, e.g. different numbers of digits in the representation of floating point numbers, padding of characters in character strings etc.). All failures affecting only one out of two (or at most n-1 out of n) versions are detectable.

Non-Detectable failures: the ones for which two (or more) servers return identical incorrect results.

Table 3 contains a summary of the results on each of the six possible two-version combinations. Here we only include bugs that could be run on both servers, i.e. we

exclude dialect-specific bugs. Only four of the 12 coincident failures we observed are non-detectable. We can see that diversity allows detection of failures for at least 94% of these bugs.

Table 3 - Summary of results for the two-version combinations

Pairs of servers	Total 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

5. Common faults

We now discuss the bugs that caused coincident failures, listed in Table 4. We give some details about the functions affected and conjectures about the probable severity and frequency of failure as a function of the environment of use of the server.

There were 13 bugs in total that were originally reported for one server but caused failure in another. *12 caused a failure in both the server for which they were reported and another server*. One bug (MSSQL bug report 56775) was reported for MSSQL, did not cause failure in MSSQL (possible Heisenbug) but did cause failure in PostgreSQL.

Table 4 - Bugs that cause coincident failures. The table should be read horizontally to know for which server the bug was reported, and vertically to know in which other server it caused a failure.

	IB	PG	OR	MS
IB	N/A	1 - (Bug ID 223512)	0	2– (BugID's 217042(3), 222476)
PG	0	N/A	0	2 – (BugID's 43 and 77)
OR	0	1 – (Bug ID 1059835)	N/A	0
MS	1-(BugID 58544)	5 – (BugID's 54428, 56516, 58158, 58253, 351180)	0	N/A

Arithmetic-related bugs

PostgreSQL bug report 77 and Oracle bug report 1059835 (Gashi 2003) describe arithmetic precision problems, causing incorrect result failures. The Oracle bug 1059835 affects the MOD (modular arithmetic) operator, probably causing higher consequence failures. The failure rates for these bugs would only be expected to be high in applications with high use of mathematical functions, not a typical use of SQL servers.

Bugs affecting complex queries

PostgreSQL bug 43 (Gashi 2003) causes a failure in both PostgreSQL and MSSQL. The complex SELECT statement below, with nested sub-queries, causes the failure:

```
SELECT  P.ID AS ID, P.NAME AS NAME FROM PRODUCT P WHERE P.ID IN
(SELECT ID FROM  PRODUCT WHERE  PRICE >= '9.00' AND  PRICE <= '50' AND  ID NOT IN
((SELECT PRODUCT_ID FROM PRODUCT_SPECIAL WHERE START_DATE <= '2000-9-6' AND END_DATE
>= '2000-9-6')
UNION
(SELECT PRODUCT_ID AS ID  FROM PRODUCT_SPECIAL WHERE PRICE >= '9.00' AND PRICE <= '50'
AND START_DATE <= '2000-9-6' AND END_DATE >= '2000-9-6')))
```

Interestingly, for this same bug the two servers fail with different patterns. PostgreSQL fails returning a parsing error. MSSQL does not, but subsequently gives an incorrect result, probably because it built an incorrect parsing tree.

MSSQL Bug 58544 (Gashi 2003) causes failures in both MSSQL and Interbase. Using a LEFT OUTER JOIN on a VIEW that uses the DISTINCT keyword causes the failure. A left outer join is a special type of outer join where if you have a join between tables T1 and T2 then the joined table unconditionally has a row for each row in T1 (as opposed to a Full Outer Join where the joined table has a row for each row present in both tables T1 and T2). The DISTINCT keyword subsequently eliminates all the duplicate rows from the joined table. Complex queries would be common on large databases with many tables, leading probably to a comparatively high failure rate, with possibly high failure severity, especially for incorrect result failures.

Miscellaneous bugs

Interbase Bug 223512(2) (Gashi 2003) causes a failure in the Data Definition Language (DDL) part of SQL which is used to create/modify database objects (i.e. tables, views, users, procedures etc). It causes failures in both Interbase and PostgreSQL: both incorrectly allow a client to drop Views using the Drop Table statement. This violates the SQL-92 standard, which allows Views to be dropped only via the Drop View statement. This bug would seem to cause infrequent failures in operation and it would normally require an error by an administrator. The severity of failures would also be expected to be low since a view is just a 'virtual table' (or a stored SELECT statement), which represents the data from one or more tables. No data are lost by dropping a view,

although a runtime error will be generated each time a client attempts to access the dropped view.

Interbase bug 217042(3) (Gashi 2003) causes both Interbase and MSSQL to fail to validate the default values upon creation of tables. Therefore a statement like:

```
CREATE TABLE TEST (A INT DEFAULT 'ABC')
```

is allowed in both Interbase and MSSQL, even though an error should be raised since a string value (ABC) cannot be stored in an Integer type attribute. The DEFAULT attributes are used often in operation but it is not clear how often database users will define DEFAULT values of the wrong type. The failure to detect that an incorrect type default value is being assigned to a particular column at table creation time is non-detectable. However, a runtime error will occur, generating an error message, every time an attempt is made to insert the default value into the table: the failure will be detected, albeit with high latency[1].

Interbase bug 222476 (Gashi 2003) causes a failure in MSSQL as well. Both servers give empty field names for *avg* (average) and *sum* SQL functions, although they return correct results in these fields. This would be a serious problem for client applications that construct their output from the field names and results returned by the server.

Five of the MSSQL bug scripts also caused failure in PostgreSQL, but with the difference that PostgreSQL fails at the beginning of the bug script. This implies that the causes are probably different for the two products, and the "failure regions" (sets of demands that would trigger the bug) identified by such scripts for the two servers only partially overlap: there are variations of the script for which PostgreSQL fails but MSSQL does not. For example, MSSQL bug 54428 causes an incorrect "primary key constraint" failure in MSSQL. The same bug causes failure (at the beginning of the bug script) when an attempt is made to create a clustered index in PostgreSQL. The latter is a known bug for PostgreSQL, and its correction in the later release 7.0.3 causes PostgreSQL not to fail on any of these five scripts.

[1] If we classify the database as part of the server system, the common terminology recommended in (Laprie 1991) would imply that assigning the wrong type is an internal error, which only becomes a failure and is detected when the attempt is made to insert the default value.

6. Discussion

6.1 Extrapolating from the counts of common bugs to reliability of a diverse server

These numbers are intriguing and point to a potential for serious dependability gains from assembling a fault tolerant server from two or more of these off-the-shelf servers. But they are not definitive evidence. Apart from the sampling difficulties caused e.g. by lack of certain bug scripts, it is important to clarify to what extent our observations allow us to predict such gains.

For brevity, we consider the simplest case: suppose that users of a certain database server product A try to obtain a more dependable service by using a fault-tolerant, replicated, diverse server AB, built from product A plus another product B (for discussion of the feasibility and design problems, see (Popov, Strigini et al. 2004)). The number of bugs reported over a certain reference period (say one year) for product A is m_A. Our study then finds that of these m_A bugs, only m_{AB} also caused failure of B. We may then expect that, had these users been using AB instead of A, only those failures of A that were due to those m_{AB} bugs could have caused complete service failures. How much more reliable would this have made the AB server, compared to the A server?

Before proceeding, we introduce some more simplifications. The possible effects of individual server failures on system failures have been discussed in Sections 4.1 and 4.3, under the definitions of "self-evident" and "detectable" failures. Here, for the sake of brevity, we use a simplified scenario: failures of both servers A and B on the same demand are "system failures", and failures of a single one of them are not[2]. In addition, we only consider the effects on reliability of the factor that we have studied: the diversity between faults of the two products A and B. We thus ignore any effects of the middleware needed in the AB server, which adds complexity and thus possibly faults; and of added complexity in client applications that used complex vendor-specific features of server A, if they must be adapted to use the more restricted feature set of server AB.

[2] This simplified model is still realistic if either: i) we are only concerned with interruptions of service, and all failures of A and/or B are detectable (crashes, self-detected errors, or different erroneous results if both A and B fail); or ii) we are concerned with undetected erroneous results, and all failures of both A and B on the same demand are pessimistically assumed to produce such results.

With these simplifications, the AB server is certain to be at least as reliable as the single A server because it only fails if both A and B fail. We still need to assess the size of the probable reliability gain. To this end, we need to take into account various complications: the difference between fault records and failure records; imperfect failure reporting; variety of usage profiles.

We can start with a scenario in which our data would be sufficient for trustworthy predictions, and then discuss the effects of these assumptions not holding in practice. This ideal scenario is as follows: we are interested in the reliability gains for a database installation using server A, if it were to switch to a diverse server AB, assuming that this installation has a usage profile (probabilities of all possible demands on the server) similar to the average of all the bug-reporting installations of server A[3]. We assume that users neither change their patterns of usage of the databases (demand profile) nor upgrade to new releases of the database servers[4]; that all failures that affected installations of A during the reference year were noticed and reported; and that there is exactly one bug report for each failure that occurred.

Then, we can state that the bug reports describe a one-year sample of operation of the system, and our best reliability prediction is that the same set of users, during another year of operation, would experience a mean number m_A of system failures if they used A, but only m_{AB} if they used AB. With the numbers we observed, the ratio m_{AB} / m_A is quite small, so the expected reliability gain would be large. Given that the reports come from millions of installations, each submitting many demands[5], we might even trust that the true failure probability per demand is close to the observed frequency of failures.

The first difficulty with this analysis is that reports concern bugs, not how many failures each caused. They do not tell us whether a bug has a large or a small effect on reliability, although the faults that did not cause failures would tend to have stochastically lower

---

[3] Or, from a market-assessment viewpoint, we may consider the average reliability gains for the population of all database installations which depend on server A, if they switched to using AB.

[4] Because we wish to reason about the reliability effects of diversity alone. This scenario also has practical interest, though. Usage patterns vary over time, but periods of very slow variations must exist; users do upgrade to new versions, but upgrades bring expense and new problems, so that it is interesting to see whether diversity would be a more cost-effective way of achieving good average dependability over a system's lifetime than frequent upgrades.

[5] How to define a "demand" to a state-rich system like a database server, for the purpose of inference about reliability, is a tricky theoretical and practical issue. For this informal discussion of other difficulties in inference, we ask the reader to accept that a practical solution can be found, somewhere between a single command and the whole sequence of commands over the lifetime of an installation. (cf e.g. (Tian, Peng et al. 1995) for examples of useful compromises).

effect on reliability than those that caused failures. Thus, the m_{AB} bugs which still cause the fault tolerant server AB to fail may account for a large (perhaps close to 100%) or a small fraction (perhaps close to 0) of the failures observed in A's operation. The actual reliability gain may be anywhere between negligible and very high.

Software is often assessed in terms of number of bugs remaining. But it is easily seen that the bug reports do not give us any information on this number: the m_A bugs reported may be the only bugs in the products, or they may be a fraction of them (perhaps minimal), which happened to be the ones causing failures during the reference year.

Another difficulty is not knowing how many of the failures that occur are actually reported. This fraction is certainly less than 100%. If all failures had the same probability of being reported, the ratio between our predicted failure counts for AB and A would still be the ratio m_{AB} / m_A. Reporting is probably biased, for instance towards bugs that cause higher frequency or higher severity of failures. Some failures – like crashes – are more noticeable than others, like storing incorrect data in some data fields, which may not produce visible effects for a long time (also making it more difficult to trace the visible problem back to its cause). Some users are more assiduous at producing failure reports, so the bugs that affect them more are also more likely to be reported, even if not so important for other users.

In the end, we do not know in detail how failure reporting differs between different bugs, but bug reports are likely to be better evidence about bugs that cause blatant failures than about subtle (arguably more dangerous) failures. This prompts another consideration: as reported bugs are corrected and products mature, more of their failures are likely to be of the subtler types, unlikely to be reported. Therefore failure underreporting probably causes a bias towards *under*estimating the frequency of failures for which diversity would help. This makes diversity a more attractive defence, but it also means that bug reports will become a less and less accurate representation of the set of failures actually occurring.

Last, we have the problem of usage profiles. A single user organisation needs predictions about the dependability of its specific installation of server AB or A (i.e., with or without diversity), which depends on its specific usage profile, which differs – perhaps by much – from the aggregate profile of the user population which generated the bug reports.

Installations that manage different databases, with different user needs, are subjected to different usage profiles. It is then plausible that different bugs are important for different installations; this conjecture is also supported by a possible interpretation of Adams' findings (Adams 1984) about the surprisingly small average failure rates of many bugs, when averaged over many installations. Then, the number of bugs whose effects can be tolerated (what we have counted here) gives little information about the resulting dependability gains. The actual effect can only be determined empirically. The user organisation may seek indirect evidence from the publicly available bug reports: if they generally match the failures experienced locally, the local effects of tolerating those bugs can be assessed. However if it does not, little insight is gained, and the exercise is time-consuming.

6.2 Decisions about deploying diversity

We have underscored that these results are only prima facie evidence for the usefulness of diversity.

A better analysis would be obtained from the actual failure reports (including failure counts), available to the vendors, especially if they use automatic failure reporting mechanisms (users are biased towards under-reporting of failures from bugs they have reported before, or for which they have successful workarounds or recovery mechanisms), and even better if they also have indications about the users' usage profile (from rough measures like the size of the database managed, to detailed monitoring as proposed in (Voas 2000)). However, vendors are often wary of sharing such detailed dependability information with their customers.

How can then individual user organisations decide whether diversity is a suitable option for them, with their specific requirements and usage profiles? As usual for dependability-enhancing measures, the cost is reasonably easy to assess: costs of the software products, the required middleware, difficulties with client applications that require vendor-specific features, hardware costs, run-time cost of the synchronisation and consistency enforcing mechanisms, and possibly more complex recovery after some failures. The gains in improved reliability and availability (from fewer system failures and easier recovery from some failures, set against possible extra failures due to the added middleware), and

possibly less frequent upgrades, are difficult to predict except empirically. This uncertainty will be compounded, for many user organisations, by the lack of trustworthy estimates of their baseline reliability with respect to subtle failures: databases are used with implicit confidence that failures will be self-evident.

We note that for some users the evidence we have presented would already indicate a diverse server to be a reasonable and relatively cheap precautionary choice, even without good predictions of its effects. These are users who have: serious concerns about dependability (e.g., high costs for interruptions of service or undetected incorrect data being stored); applications which use mostly the core features common to multiple off-the-shelf products (recommended by practitioners to improve portability of the applications); modest throughput requirements for updates, which make it easy to accept the synchronisation delays of a fault-tolerant server.

7. Conclusions

To estimate the possible advantages of modular-redundant diversity in complex off-the-shelf software, we studied a sample of bug reports from four popular off-the-shelf SQL database server products. We checked whether more than one product exhibited bugs that would cause common-mode failures if the products were used in a diverse redundant architecture. It appears that such common bugs are rare. We found very few bugs that affected two of the four servers, and none that affected more than two. Moreover only four of these bugs would cause identical, undetectable failures in two servers. Fault-tolerant, diverse servers seem to have a good chance of improving failure rates and availability.

These preliminary results must be taken with caution, as discussed in Section 6, but are certainly interesting and indicate that this topic deserves further study. Their immediate implications vary between users, but there are classes of database server installations for which even these preliminary results seem to recommend diversity as a prudent and cost-effective strategy. Decisions would of course involve many other considerations which we could not discuss here: performance, total cost of ownership including updates, risks of dependence on one vendor, etc.

The practical obstacle would be the need for "middleware": most users would need an off-the-shelf middleware package, which in turn is not likely to be developed until there are enough users. On the other hand, a dedicated user could develop a middleware package in the hope of seeing his investment amplified through the creation of an open-source community of user/developers. But once the diverse server is running, the dependability changes due to diversity could be directly assessed. The user could decide on an ongoing basis which architecture is giving the best trade-off between performance and dependability, from a single server to the most pessimistic fault-tolerant configuration (with tight synchronisation and comparison of results at each query).

Some other interesting observations include:

- it may be worthwhile for vendors to test their servers using the known bug reports for other servers. For example, we observed 4 MSSQL bugs that had not been reported in the MSSQL service packs (previous to our observation period). Oracle was the only server that never failed when running on it the reported bugs of the other servers;
- the majority of bugs reported, for all servers, led to "incorrect result" failures (64.5%) rather than crashes (17.1%) (despite crashes being more obvious to the user). This is contrary to the common assumption that the majority of bugs lead to an engine crash, and warrants more attention by users to fault-tolerant solutions, and by designers of fault-tolerant solutions to tolerating subtle and non fail-silent failures.

Future work that is desirable includes:

- repeating this study on later releases of the servers, to verify whether the general conclusions drawn here are repeated, indicating that they are the consequences of factors that do not disappear with the evolution of the software products;
- statistical testing to assess the actual reliability gains. This is already under way. We have run a few million queries with various loads including experiments based on the TPC-C benchmark. We have not observed any failures so far (however, with the TPC-C load we found that a significant gain in performance can be obtained with diverse servers (Gashi, Popov et al. 2004)). We plan to continue these experiments with more complete test loads. These are important

for their own sake, as evidence for decision-making, but also for the side benefit of checking how far the data confirm the impressions gained from this study, and thus how accurate a picture fault reports paint for these products;

- studying alternative options for software fault tolerance with OTS servers, e.g. wrappers rephrasing queries into alternative, logically equivalent sets of statements to be sent to replicated, even non-diverse servers (Gashi, Popov et al. 2004);
- developing the necessary components for users to be able to try out diversity in their own installations, since the main obstacle now is the lack of popular off-the-shelf "middleware" packages for data replication with diverse SQL servers.

Acknowledgment

This work was supported in part by the "Diversity with Off-The-Shelf components" (DOTS) Project funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC). We would also like to thank Bev Littlewood, Peter Bishop and the anonymous DSN reviewers for comments on an earlier version of this paper.

References

Adams, E. N. (1984), "*Optimizing Preventive Service of Software Products*", IBM Journal of Research and Development 28(1), pp: 2-14.

Adobe (2004), "*Macromedia JRun*", http://www.adobe.com/products/jrun/productinfo/overview/.

Berenson, H., P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil (1995), "*A Critique of ANSI SQL Isolation Levels*", *in proc.* Int. Conf. on Management of Data (SIGMOD '95).

Bernstein, P. A., V. Hadzilacos and N. Goodman (1987), "*Concurrency Control and Recovery in Database Systems*", Reading, Mass., Addison-Wesley.

Chandra, S. and P. M. Chen (1998), "*How Fail-Stop are Programs*", *in proc.* Int. Symp. on Fault-Tolerant Computing (FTCS '98), IEEE Computer Society Press, pp: 240-249.

Chandra, S. and P. M. Chen (2000), "*Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '00), NY, USA, IEEE Computer Society Press, pp: 97-106.

Fekete, A., D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha (2000), "*Making Snapshots Isolation Serialisable*", http://www.cs.umb.edu/~isotest/snaptest/snaptest.pdf.

Gashi, I. (2003), "*Tables containing known bug scripts of Interbase, PostgreSQL, Oracle and MSSQL.*" http://www.csr.city.ac.uk/people/ilir.gashi/DBMSBugReports/.

Gashi, I., P. Popov, V. Stankovic and L. Strigini (2004), "*On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*", *in* Architecting Dependable Systems II, R. de Lemos, Gacek, C., Romanovsky, A. (Eds.), Springer-Verlag, 3069, pp: 191-214.

Gray, J. (1986), "*Why Do Computers Stop and What Can be Done About it?*" *in proc.* Int. Symp. on Reliability in Distributed Software and Database Systems (SRDSDS '86), Los Angeles, CA, USA, IEEE Computer Society Press, pp: 3-12.

Gray, J. (2000), "*FT101: Talk at UC Berkeley on Fault-Tolerance*", http://research.microsoft.com/~Gray/talks/UCBerkeley_Gray_FT_Avialiability_talk.ppt.

Hiltunen, M. A., R. D. Schlichting, C. A. Ugarte and G. T. Wong (2000), "*Survivability Through Customization and Adaptability: The Cactus Approach*", *in proc.* DARPA Information Survivability Conference & Exposition.

Jimenez-Peris, R. and M. Patino-Martinez (2003), "*D5: Transaction Support*", ADAPT Middleware Technologies for Adaptive and Composable Distributed Components, Deliverable IST-2001-37126.

Jimenez-Peris, R., M. Patino-Martinez, G. Alonso and B. Kemme (2002), "*Scalable Database Replication Middleware*", *in proc.* 22nd Int. Conf. on Distributed Computing Systems, Vienna, Austria, IEEE Computer Society Press, pp: 477-484.

Laprie, J. C., Ed. (1991), "*Dependability: Basic Concepts and Associated Terminology*", Dependable Computing and Fault-Tolerant Systems Series, Springer-Verlag.

Laprie, J. C., J. Arlat, C. Beounes and K. Kanoun (1990), "*Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures*", IEEE Computer 23(7), pp: 39-51.

Lee, I. and R. K. Iyer (1995), "*Software Dependability in the Tandem GUARDIAN System*", IEEE Transactions on Software Engineering 21(5), pp: 455-467.

Microsoft, "*List of Bugs Fixed by SQL Server 7.0 Service Packs*", http://support.microsoft.com/default.aspx?scid=kb;EN=US;313980.

Oracle, "*Oracle Metalink*", http://metalink.oracle.com/metalink/plsql/ml2_gui.startup.

Pedone, F. and S. Frolund (2000), "*Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '00), Nurnberg, Germany, IEEE Computer Society, pp: 176-85.

Popov, P., L. Strigini, A. Kostov, V. Mollov and D. Selensky (2004), "*Software Fault-Tolerance with Off-the-Shelf SQL Servers*", *in proc.* Int. Conf. on COTS-based Software Systems (ICCBSS'04), Redondo Beach, CA USA, Springer, pp: 117-126.

Popov, P., L. Strigini, S. Riddle and A. Romanovsky (2001), "*Protective Wrapping of OTS Components*", *in proc.* 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction, Toronto.

Popov, P., L. Strigini and A. Romanovsky (2000), "*Diversity for Off-The-Shelf Components*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '00) - Fast Abstracts supplement, New York, NY, USA, IEEE Computer Society Press, pp: B60-B61.

PostgreSQL, "*PostgreSQL Bugs Mailing List Archives*", http://archives.postgresql.org/pgsql-bugs/.

Reynolds, J., J. Just, E. Lawson, L. Clough, R. Maglich and K. Levitt (2002), "*The Design and Implementation of an Intrusion Tolerant System*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '02), Washington, D.C., USA, IEEE Computer Society Press, pp: 285-292.

Schneider, F. (1984), "*Byzantine Generals in Action: Implementing Fail-Stop Processors*", ACM Transactions on Computer Systems 2(2), pp: 145-154.

SourceForge, "*Interbase (Firebird) Bug tracker*", http://sourceforge.net/tracker/?atid=109028&group_id=9028&func=browse.

Sutter, H. (2000), "*SQL/Replication Scope and Requirements Document*", ISO/IEC JTC 1/SC 32 Data Management and Interchange WG3 Database Languages, H2-2000-568.

Tian, J., L. Peng and J. Palma (1995), "*Test-Execution-Based Reliability Measurement and Modeling for Large Commercial Software*", IEEE Transactions on Software Engineering 21(5), pp: 405-414.

Valdes, A., M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saidi, V. Stavridou and T. E. Uribe (1999), "*An Adaptive Intrusion-Tolerant Server Architecture*", http://www.sdl.sri.com/users/valdes/DIT_arch.pdf.

Voas, J. (2000), "*Deriving Accurate Operational Profiles for Mass-Marketed Software*", http://www.cigital.com/papers/download/profile.pdf.

Wang, F., F. Gong, C. Sargor, K. Goseva-Popstojanova, K. Trivedi and F. Jou (2001), "*SITAR: A Scalable Intrusion-Tolerant Architecture for Distributed Services*", *in proc.* 2001 IEEE Workshop on Information Assurance and Security, West Point, New York, U.S.A.

Weismann, M., F. Pedone and A. Schiper (2000), "*Database Replication Techniques: a Three Parameter Classification*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '00), Nurnberg, Germany, IEEE Computer Society Press, pp: 206-217.

Paper-2. Fault Tolerance via Diversity for Off-The-Shelf Products: a Study with SQL Database Servers

Abstract: *If an off-the-shelf software product exhibits poor dependability due to design faults, software fault tolerance is often the only way available to users and system integrators to alleviate the problem. Thanks to low acquisition costs, even using multiple versions of software in a parallel architecture, a scheme formerly reserved for few and highly critical applications, may become viable for many applications. We have studied the potential dependability gains from these solutions for off-the-shelf database servers. We based the study on the bug reports available for four off-the-shelf SQL servers, plus later releases of two of them. We found that many of these faults cause systematic, non-crash failures, a category ignored by most studies and standard implementations of fault tolerance for databases. Our observations suggest that diverse redundancy would be effective for tolerating design faults in this category of products. Only in very few cases would demands that triggered a bug in one server cause failures in another one, and there were no coincident failures in more than two of the servers. Use of different releases of the same product would also tolerate a significant fraction of the faults. We report our results and discuss their implications, the architectural options available for exploiting them and the difficulties that they may present.*

Co-authors: Dr. Peter Popov, Prof. Lorenzo Strigini

Journal: IEEE Transactions on Dependable and Secure Computing (IEEE TDSC)

Date of submission: 24-October-2006

Status: *Accepted in May 2007. To be published in the July-September 2007 issue.*

Number of reviewers: 3

Publication date: TBC

Full citation: Gashi I., Popov P., Strigini L., "*Fault diversity among off-the-shelf SQL database servers*", in *IEEE Transactions on Dependable and Secure Computing,* IEEE Computer Society Press, to appear, 2007

1. Introduction

The use of "off-the-shelf" (OTS) – rather than custom-built – products is attractive in terms of acquisition costs and time to deployment but brings concerns about dependability and "total cost of ownership". For safety- or business-critical applications, in particular, purpose-built products would normally come with extensive documentation of good development practice and extensive verification and validation; when switching to mass-distributed OTS systems, users – system designers or end users – often find not only a lack of this documentation, but anecdotal evidence of serious failures and/or bugs that undermines trust in the product. Despite the large-scale adoption of some products, there is usually no formal statistical documentation of achieved dependability levels, from which a user could attempt to extrapolate the levels to be achieved in his/her own usage environment.

For all these reasons, when systems are built out of OTS products, fault tolerance is often the only viable way of obtaining the required system dependability (Popov, Strigini et al. 2000), (Valdes, M. Almgren et al. 2003), (Hiltunen, Schlichting et al. 2000). These considerations apply not only to OTS software, but also to hardware, like microprocessors, or complete hardware-plus-software systems. In this paper we will consider "software fault tolerance" (by which we mean "fault tolerance against software faults"), focusing on a specific category of software products. Fault tolerance may take multiple forms (Strigini 2005), from simple error detection and recovery add-ons (e.g. wrappers) (Popov, Strigini et al. 2001) to full-fledged "diverse modular redundancy" (e.g. "N-version programming": replication with diverse versions of the components) (Strigini 2005). Even this latter class of solutions becomes affordable with many OTS products and has the advantage of a fairly simple architecture. The cost of procuring two or even more OTS products (some of which may be free) would still be far less than that of developing one's own product.

All these design solutions are well known from the literature. The questions, for the developers of a system using OTS components, are about the dependability gains, implementation difficulties and extra cost that they would bring for that specific system. We report here some evidence about potential gains, and briefly discuss the architectural

issues that would determine feasibility and costs, for a specific category of OTS products: SQL database servers, or "database management systems" (DBMSs)[6].

This category of products offers a realistic case study of the advantages and challenges of software fault tolerance in OTS products. DBMS products are complex, mature enough for widespread adoption, and yet with many faults in each release[7]. Fault tolerance in DBMS products is a thoroughly studied subject, with standard recognized solutions, some of which are commercially available. But these solutions do not give full protection against software faults, because they assume fail-stop (Schneider 1984) or at least self-evident failures[8]: errors are detected promptly enough that the database contents are not corrupted, or that a suitable correct checkpoint can be identified and used for rollback. There is no guarantee that software faults in the OTS DBMS products themselves will satisfy this assumption. As we document here, they do not, and we know of no published statistical evidence of the frequency of violations, which one could use as evidence that the assumption is satisfied with high enough probability for a specific application of one of these OTS products.

There are many OTS SQL DBMS products, obeying (at least nominally) common standards (SQL 92 and SQL 99), which makes diverse redundancy feasible in principle. For instance, a parallel-redundant architecture using two replicas of a database, managed by two diverse DBMS products, would allow error detection via comparison of results from the two DBMS products. A fault-tolerant server capable of tolerating server software faults can be built from installations of two or more diverse DBMS products, connected by middleware that makes them appear to clients as a single database server. There are clearly problems as well: in particular, existing DBMS products have certain concurrency control and fault tolerance features that rely on lack of diversity between

---

[6] Everyday terms may be ambiguous when discussing redundant and diverse architectures. We will apply these conventions: a *DBMS product* is a specific software package; a fault-tolerant database server includes one or more *channels* (each performing the database server function), each including an installation of a DBMS product (these may be the same product or different ones - different *versions*) and a *replica* of the database. Two replicas of the database will be physically different if they are in channels that use different DBMS products. They may also exhibit temporary differences due to the asynchronous operation of the channels. We follow the popular usage of the word "bug" as synonym for "software fault" or "defect".

[7] And even features that imply an accepted possibility of an incorrect behaviour, albeit rare. An example of the latter is the known "write skew" (Berenson, Bernstein et al. 1995) problem with some optimistic concurrency control architectures (Fekete, Liarokapis et al. 2005).

[8] By "self-evident failures" we will mean failures that a generic client of the DBMS product can detect without depending on knowledge of the specific database and its semantics. They are those failures that – as seen by the client – consist in issuing an error message to the client, spontaneously aborting a transaction, "hanging" or crashing.

replicated executions for their proper and efficient operation. However, it is worth exploring the costs and benefits of solutions that accept the drawbacks of diversity in return for improved dependability. For many users, there is no practical alternative to OTS DBMS products, and performance losses may well be acceptable in return for improved assurance. In addition to tolerating faults in general, users may look at software fault tolerance as a way of guaranteeing good service during upgrades of the DBMS products, when new bugs might appear that are serious under the usage profile of their specific installation, and/or of delaying "patches" and upgrades, thus reducing the total cost of ownership of DBMS products.

As a preliminary assessment of the potential effectiveness of software fault tolerance with DBMS products, we have studied publicly available fault reports for four DBMS products (two open-source and two closed-development). We ask questions about the potential effectiveness of *design diversity* – deploying two different products. Fault reports are the only publicly available dependability evidence for these products, so our study concerns *fault diversity* among them. Complete failure logs would be much more useful as statistical evidence, but they are not available. Many vendors discourage users from reporting already known bugs; detailed failure data are rarely available even to the software vendors themselves. This scarcity of data also makes it difficult to estimate how dependable a DBMS product will be for a specific installation. But the many reports of failures of DBMS products suggest that some users need reliability improvements.

In a first study (Gashi, Popov et al. 2004b) (***the preceding reference forms part of this thesis as Paper-1***), we looked at the set of bugs reported for one release of each DBMS product. For each bug, we took the bug script (a sequence of SQL statements) that would trigger it and ran it on all four DBMS products (if possible), to check for coincident failures: if the bug script does not trigger failures in the other DBMS product, we take this as evidence that software fault tolerance would tolerate that fault. We found that a high number of reported faults would not be tolerated (or even detected) by existing, non-diverse fault-tolerant schemes but did not cause coincident failures in any two DBMS products, offering a way of tolerating them.

These intriguing results suggested a potential for considerable dependability gains from using diverse OTS DBMS products, but they only concerned a *specific snapshot* in the

evolution of these products. We therefore ran a follow-up study with later releases of DBMS products (thus with different set of bug reports), with results that substantially confirm the previous ones. This paper reports the complete results of the two studies.

The rest of the paper is organized as follows: in Section 2, we briefly discuss the architectural issues in software fault tolerance with DBMS products – feasibility, design alternatives and performance issues – since they determine the usefulness of the empirical results we report; Section 3 presents the results of the two empirical studies of known bugs of DBMS products, including the comparisons between older and newer releases of two DBMS products; Section 4 contains a discussion of the implications of our studies; Section 5 contains a review of related work on database replication, interoperability of databases, empirical evidence on DBMS products' faults and failures and diversity with off-the-shelf components and Section 6 contains conclusions and outlines of further work.

2. Architectural considerations

2.1 Current solutions for DBMS replication

Standard solutions for automatic fault tolerance in databases use the mechanisms of atomic transactions and/or checkpointing to support backward recovery, which can be followed by retry of the failed transactions. These solutions will tolerate transient faults, if detected, and if combined with replication will mask permanent faults, without service interruption.

Various data replication solutions exist (Bernstein, Hadzilacos et al. 1987), (Weismann, Pedone et al. 2000), (Pedone and Frolund 2000), (Patino-Martinez, Jiménez-Peris et al. 2005), (Lin, Kemme et al. 2005). In commercial DBMS products, they are often called "fail-over" solutions: following a (crash) failure of the primary DBMS product, the load is transparently taken over by a separate installation of the DBMS product holding a redundant copy of the database, at the cost of aborting the transactions affected by the crash. Multiple copies may be used. The code for fault tolerance is integrated inside the DBMS product. A recent survey (Jimenez-Peris and Patino-Martinez 2003) calls this a "white box" solution. Alternatively, replication can be managed by middleware separate

from the DBMS products: "black box" solutions (fault tolerance is entirely the responsibility of the middleware), or "grey box" (the middleware exploits useful functions available from the DBMS products (Jimenez-Peris, Patino-Martinez et al. 2002)). Our discussion here will refer to "black box" solutions: the only ones that can be built without access to OTS source code, and most convenient for studying the design issues in the use of redundancy and diversity. We will assume that fault tolerance is managed by a layer of middleware; clients see the fault-tolerant database server via this middleware layer, which co-ordinates the redundant channels.

Existing data replication solutions use sophisticated schemes for reducing the overhead involved in keeping the copies up to date. Their common weakness is their dependence on the assumption of "fail-stop" or at least "self-evident" failures. This assumption simplifies the protocols for data replication, and allows some performance optimisation. For instance, in the Read Once Write All Available (ROWAA) (Bernstein, Hadzilacos et al. 1987) replication protocol the read statements[9] are executed by a single replica while the write statements are executed by all replicas. These fault-tolerant solutions are considered adequate by standardizing bodies (Sutter 2000), despite the assumption being false in principle. Some recent solutions (Kemme and Alonso 2000) seek further optimisation by executing the write statements on a single replica, which then propagates the changes to all the (available) replicas.

As we shall see, current OTS DBMS products suffer from many bugs that cause non-crash, non-self-evident failures. The failures that these cause may be undetected erroneous responses to read statements, and/or incorrect writes to all the replicas of the database.

For these kinds of failure, the current data replication solutions are deficient, in the first place from the viewpoint of error detection. Two kinds of remedy are possible:

- *database-*, or *client-specific* solutions that depend on the client (an automatic process or a human operator) to run reasonableness checks on the outputs of the DBMS product and order recovery actions if it detects errors. Good error detection may be achieved by exploiting knowledge of the semantics of the data

[9] We will use the term "statement" to refer to the SQL requests that are sent to the DBMS product. These may be read or write *data manipulation language* (DML) statements or *data definition language* (DDL) statements

stored and the processes that update them. This knowledge may also support more efficient error recovery than simple rollback and retry. The main disadvantages are high implementation cost (especially with a workforce generally unaware of the need for fault tolerance), high run-time cost, at least for human-run checks, and the possibility of low error detection coverage if the database is – as common – the sole repository of the data[10].

- *generic* solutions that use active replication (Gashi, Popov et al. 2004a) (***the preceding reference forms part of this thesis as Paper-3***) for error detection, so that errors can be detected by comparing the results of redundant executions, and/or corrected, via voting or copying the results of correct executions.

2.2 Diversity

Replication will give a basis for effective fault tolerance if the multiple channels do not usually fail together on the same demand, or at least they tend not to fail with identical erroneous results. To pursue such *failure diversity*, a designer building a fault-tolerant database server can use various forms of diversity:

- simple separation of redundant executions. This is the weakest form, but it may yet tolerate some failures. It is well known that many bugs in complex, mature software products are "Heisenbugs"[11] (Gray 1986), i.e., they cause apparently non-deterministic failures. When a database fails, its identical copy may not fail, even with the same sequence of inputs. Even repeating the same operations on the same copy of a database after rollback may in principle not replicate the same failure;
- *design diversity*, the typical form of parallel redundancy for fault tolerance against design faults: the multiple replicas of the database are managed by diverse DBMS products;

---

[10] Simple reasonableness or "safety" checks are often available, but have limited efficacy against some failure scenarios. E.g., reasonableness checks may prevent the posting of incredibly large movements in a company's accounts, yet allow many small systematic errors, allowing large cumulative errors to build up before the problem comes to light.

[11] The name introduced by Gray (Gray 1986) for bugs that are difficult to reproduce, as they only cause failures under special conditions: "strange hardware conditions (rare or transient device fault), limit conditions (out of storage, counter overflow, lost interrupt, etc.) or race conditions", "Bohrbugs" instead appear to be deterministic (the failures they cause are easy to reproduce in testing).

- *data diversity* (Ammann and Knight 1988): thanks to the redundancy in the SQL language, a sequence of one or more SQL statements can be "rephrased" into a different but logically equivalent sequence to produce redundant executions, reducing the risk of a failure being repeated when the rephrased sequence is executed on the same or another replica of even the same DBMS product. Two of the present authors have reported elsewhere (Gashi and Popov 2006) (***the preceding reference forms part of this thesis as Paper-4***) on a set of "rephrasing rules" that would tolerate at least 60% of the bugs examined in our studies. Another possibility is varying the "hints" to the "query optimiser" of the DBMS that are included with SQL statements.
- *configuration diversity* (which can be seen as a special form of data diversity). DBMS products have many configuration parameters, affecting e.g. the amount of system resources they can use (amount of RAM and/or the "page size" used by the database), or the degree of optimisation to be applied to certain operations: given the same database contents, varying these parameters between two installations can produce different implementations of the data and the operation sequences on them, and thus decrease the risk of the same bug being triggered in two installations of the same DBMS product by the same sequence of SQL statements.

These precautions can in principle be combined (for instance, data diversity can be used with diverse DBMS products), and implemented in various ways, including manual application by a human operator.

Among the above forms of diversity, design diversity appears the most likely to avoid coincident failures in redundant executions, but it may impose substantial limitations or design costs. In the first place, OTS DBMS products, even if they nominally implement the operations of the standard SQL language, in practice use different "dialects": they use different syntax for commands that are semantically the same (this problem can be solved via automatic, on-the-fly translation); more importantly, each offers extra, non-standard features, which would require either more complex translation ("rephrasing" of statements, mentioned above as a form of data diversity, can be useful to overcome problems with translation, as we have shown (Gashi and Popov 2006)), and/or clients to

be limited to using a common subset among the features of the diverse DBMS products. In addition, many aspects of database operation are specified in a non-deterministic fashion, making the goal of ensuring consistency among replicas difficult even with same-product replication, and more so with diverse replication.

A special case of design diversity is using successive releases of the same DBMS product. This will avoid or greatly reduce the problems due to "dialect" differences. It may be expected to tolerate fewer faults, since the successive releases will share large portions of their code, including some bugs; but it may be attractive for "smoothing out" upgrades which may otherwise cause peaks of unreliability in a database installation, due to the new faults introduced, and at the same time evaluating the new release to decide when it has reached sufficient dependability to be used alone. Similar practices have been applied for embedded and safety critical systems (Cook and Dage 1999), (Tai, Tso et al. 2002).

We now discuss briefly the architectural options available in designing automated fault tolerance solutions with some form of diversity applied to OTS DBMS products. A basic "black box" replication architecture delegates the management of redundancy to a layer of middleware, as in Fig. 1, so that the multiple DBMS products appear to clients as a single server. There may be any number of channels, though typical values would be one (using "time redundancy" – repeating the execution on the single DBMS product – when needed), two or three (the minimum that allows error masking through voting). We will normally refer to systems with two replicas, unless otherwise noted.

This basic architecture can be used for various fault tolerance strategies, with different trade-offs between coverage for various types of failures, performance, ease of integration etc (Anderson and Lee 1990). The most serious design issues concern ensuring replica determinism, for those replication schemes that require it. The difficulty is that each DBMS product has its own concurrency control strategy, and these are non-deterministic and may be different between products. Proprietary replication solutions deal with this problem by using knowledge of the implementation of a DBMS product. For a middleware layer managing generic OTS products, this is more difficult, especially since commercial vendors may keep these details secret. The middleware can instead artificially serialize statements in the same way on all replicas (Popov, Strigini et al.

2004), (Jimenez-Peris, M. Patino-Martinez et al. 2002). There are performance costs, but these will be acceptable for many installations, though intolerable on others, depending on the amount and pattern of write transactions in a specific installation.

A separate requirement, easier to satisfy, is that any voting/comparison algorithm need to allow for "cosmetic" differences between equivalent correct results issued by different DBMS products, e.g. differences in the padding blank characters in character strings or different numbers of digits in the representations of floating point numbers. Trade-offs exist here between embedding in the algorithm more knowledge about the idiosyncrasies of each specific product, and keeping it more generic at the cost of possibly lower coverage.

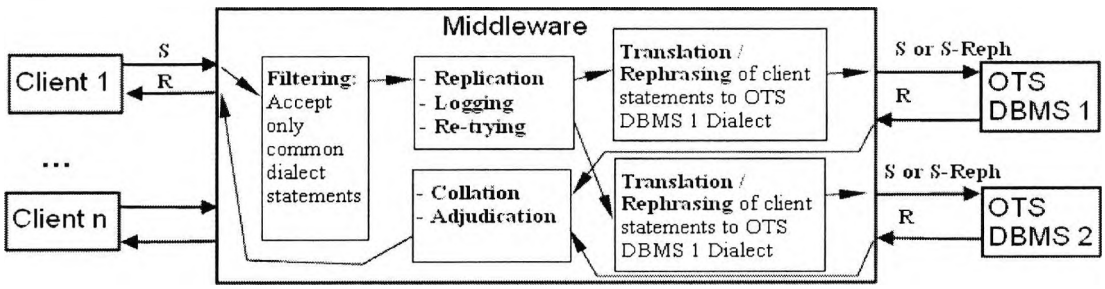

Fig. 1 - A stylised design of a fault-tolerant database server with two channels. Each channel includes an installation of an OTS DBMS product (these may be the same or different products, including different releases of the same product) and a replica of the database. The middleware must ensure connectivity between the clients and the DBMS products, some filtering of the statements sent by clients (e.g. returning error messages to the client for statements that are not supported by both the underlying OTS DBMS products), replication and concurrency control, management of fault tolerance (error detection; error containment, diagnosis and correction; state recovery), as well as translation of SQL statements ("S" in the figure) sent by the clients to the dialects of the respective OTS DBMS products (translation may be done in off-the-shelf add-on components). Support for "data diversity" through "rephrasing" may also form part of the same components which perform translation: rephrasing rules will produce rephrased versions – "S-reph" in the figure –of the statements sent by clients. The middleware must also adjudicate the results – "R" in the figure – from the OTS DBMS products and return a result to the client[s].

2.3 Design options for fault tolerance via diverse replication

2.3.1 Detection of server failures

Erroneous responses to read statements can be detected by comparing the outputs of the channels, detecting those non-self-evident failures that cause some discrepancy between these outputs.

Both design diversity and data diversity increase the chance of detection, compared to simple replication. Replica determinism is necessary, i.e., discrepancies between correct

results must be rare, as they may cause correct results to be treated as erroneous, and thus a performance penalty. Self-evident failures are detected as in non-diverse servers, via the server's error messages (i.e., via the existing error detection mechanisms of the DBMS products) and time-outs.

Erroneous updates to the databases that will only cause output discrepancies in the future are also a concern. To detect them, the middleware can compare the contents of the database replicas, via the standard read commands of the DBMS products. There is a degree of freedom in how much should be compared, allowing latency/performance trade-offs. The middleware could just ask each DBMS product for the list of the records modified in each write operation, and then read and compare their contents. In principle, though, a buggy DBMS product could omit some changed records from the list it returns. So, a designer could decide to compare a superset of the data that appear to be affected, trading off time for better error detection.

Another trade-off is possible between error latency and the overhead imposed by the fault-tolerant operation: error detection can be scheduled in a more or less pessimistic mode. In the most pessimistic mode, at each operation the middleware performs all its comparisons before forwarding to the client the response from the DBMS product[s]. More optimistically, it can forward most responses immediately, and run the checks in parallel with the subsequent operation of the client and DBMS products. A natural synchronization point is at transaction commit: the middleware only allows the transaction to commit if it detected no failures.

In addition, the middleware can use slack capacity for a background audit task, comparing the complete contents of the database replicas.

2.3.2 Error containment, diagnosis and correction

Error containment is tightly linked to detection. For read statements, the middleware receives multiple responses for each statement sent to the diverse channels, one from each of them, and must return a single response to the client. In general, the middleware will present to the client a DBMS product failure as a correct but possibly delayed response (masking), or as a self-evident failure (crash - the behaviour of a "fail-silent" fault-tolerant server; or an error message - a "self-checking" server). DBMS product

failures can be masked to the clients, if the middleware can select a result that has a high enough probability of being correct:

- if more than two redundant responses are available, it can use majority voting to choose a consensus result, and to identify the failed replica which may need a recovery action to correct its state.
- with only two redundant channels, if they give different results the middleware cannot decide which one is in error. A possibility is not to offer masking, but simply a clean failure to be followed by manual diagnosis of the problem. Alternatively, additional redundant execution can be run by replaying the statements, possibly with "data diversity", i.e., rephrasing the statements (Gashi and Popov 2006).

Depending on how redundant executions are organized, the middleware may need to resolve rather complex scenarios, e.g., two diverse DBMS products, A and B, may give different responses upon first submission for a read statement, while upon resubmission of a rephrased statement A produces an error message and B a result matching A's previous result; but this is a standard adjudication problem (Di Giandomenico and Strigini 1990), (Blough and Sullivan 1994), (Parhami 2005) for which the design options and trade-offs are well known.

Again, the need for replica determinism is the main design issue with these schemes.

2.3.3 State recovery

Besides selecting probably correct results, adjudication will identify probably failed channels in the fault-tolerant database server (diagnosis). This improves availability: the middleware can selectively direct recovery actions at the channel diagnosed as having failed, while letting the other channel(s) continue providing the service.

The state of a channel can be seen as composed of the state of permanent data in the database and that of volatile data in the DBMS product's variables. For erroneous states of the latter, since the middleware cannot see the internal state of each executing DBMS product, some form of "rejuvenation" (Bao, X. Sun et al. 2005) must be applied, e.g. stopping and restarting the DBMS product.

As for state recovery of the database contents, it can be obtained:

- via standard backward error recovery – rollback followed by retry of logged write statements –, which will sometimes be effective (failures due to Heisenbugs), at least if the failures did not violate the ACID properties in the affected transactions. "Data diversity" will extend the set of failures that can be recovered this way. To command backward error recovery, the middleware can use the standard database transaction mechanisms: aborting the failed transaction and replaying its statements may produce a correct execution. Alternatively or additionally, it can use checkpointing (Gray and Reuter 1993): the middleware orders the states of the database replicas to be saved at regular intervals (by database "backup" commands: e.g., in PostgreSQL the pg_dump command). After a failure, a database replica is restored to its last checkpointed state and the middleware replays the sequence of (all or just write) statements since then (the redo log provided in some DBMS products cannot be used because it might contain erroneous writes). For finer granularity of recovery, the checkpoint-rollback mechanism can be used within transactions: this allows the handling of exceptions within transactions, and should be applied when using data diversity through "rephrasing" (Gashi and Popov 2006);
- additionally, diversity allows one to achieve forward recovery by essentially copying the state of a correct database replica into the failed one (similarly to (Tso and Avizienis 1987)). Since the formats of database files differ between the DBMS products, the middleware would need to query the correct channel[s] for their database contents and command the failed channel to write them into the corresponding records in its database, similar to the solution proposed in (Castro and Liskov 1999). This would be time-consuming, 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 statements.

During any recovery phase, the fault-tolerant server would work with reduced redundancy. A two-channel fault-tolerant server would be reduced to a non-fault-tolerant configuration. Trade-offs open to the designer involve the duration of the recovery phase (it can be shortened by more efficient algorithms or by reducing the extent of the state

that is checked/corrected), and the degree of conservatism applied during non-fault-tolerant operation.

3. Our studies of bug reports for off-the-shelf DBMS products

3.1 Generalities

We use the following terminology. The known bugs for the OTS DBMS products are documented in bug report repositories (i.e. bug databases, mailing lists etc). Each *bug report* contains a description of the bug and a *bug script* for reproducing the *failure* (the erroneous behaviour that the reporter of the bug observed). The bug script may come with indications on the database states that are preconditions for the failure (e.g., in the form of statements to issue for the database to reach one such state), plus the statements (and values for their parameters) which reproduce the *failure*. In our study we collected these bug reports and *ran* the bug scripts on installations of each of the DBMS products we used (we will use the phrase "*running a bug*" for the sake of brevity).

What constitutes an individual bug is of course not definable by any a priori rule (Frankl, Hamlet et al. 1998): people characterize a bug in terms of the apparent mistakes made by the developers, of code changes required to fix it, and/or of circumstances on which the software fails. We define a "demand" as the complete circumstances (i.e. an initial state plus a series of statements) that would cause failure. A bug report does not necessarily identify the whole set of demands (the "failure region") on which the product fails and would no longer fail if the bug were corrected. When running a bug script, we usually tested all DBMS products in our study on at least one demand (the same for all) mentioned in the bug report, and listed the bug as present in all DBMS products that failed on that demand. In some cases, we also tested the DBMS products with other similar demands - variations of the statements and/or parameters specified in a bug script. We did this when a bug script did not seem to trigger a failure in the DBMS product to which it related, to check whether the bug did appear to be present, but the reporter may have been imprecise in characterizing the conditions for triggering it; and when a bug script caused failures in more than one DBMS products, to study and compare the

"failure regions" identified in the two products, especially to determine their overlap and whether the DBMS products fail identically throughout them.

3.1.1 Reproducibility of failures

As mentioned earlier, DBMS products offer features that extend the SQL standard, and these extensions differ between products. Bugs affecting these extensions literally cannot exist in a DBMS product that lacks them. We called these bugs "dialect-specific". For example, Interbase bug 217138 (Gashi 2006) affects the use of the UNION operator in VIEWs, which PostgreSQL 7.0 VIEWs do not offer, and thus cannot be run in PostgreSQL 7.0: it is a dialect-specific bug.

Another reproducibility issue arises when a bug script does not cause failure in the DBMS product for which the bug was reported. We called these bugs 'Unreproduced' bugs. They may be Heisenbugs (Gray 1986) or bugs reported without enough detail for reproducing them. Compared to our preliminary report (Gashi, Popov et al. 2004b), we have been able to trigger some more previously 'Unreproduced' bugs (and thus we report updated statistics): by running variations of the incomplete bug script, as mentioned above; or thanks to more complete bug scripts posted after our collection period or to mailing lists other than the main repository for the respective DBMS product.

3.1.2 Classifications of failures

We ran each bug first on the DBMS product for which it was reported, and then (after translating the script into the appropriate SQL dialect[s]) on the other DBMS product[s]. We classified bugs into Reproduced and Unreproduced and into dialect-specific and non-dialect-specific bugs, as explained previously, and failures into different categories that would require different fault tolerance mechanisms:

- *Engine Crash* failures: a crash or halt of the core engine of the DBMS product.
- *Incorrect Result* failures, which are not engine crashes but produce incorrect outputs: the results do not conform to the DBMS product's specification or to the SQL standard.
- *Performance*-related failures. We classified as performance failures: i) failures that are so classified in bug reports; ii) failures observed by us if either the DBMS product clearly "hung" or, whatever the observed latency, the bug script generated

a query plan indicating potential performance problems, e.g. with an un-utilized column "index" in a SELECT statement using that column.

- *Other* failures: e.g. security related failures, such as incorrect privileges for database objects (tables, views etc.)

We further classified failures according to their detectability by a client of the DBMS products:

- *Self-Evident Failure*: engine crash failures, internal failures signalled by DBMS product exceptions (error messages) or performance failures
- *Non-Self-Evident Failures*: incorrect result failures without DBMS product exceptions, with acceptable response time.

For clients with access to at least two diverse DBMS products the failures would be:

- *Divergent failures*: any failures where DBMS products return different results. All failures affecting only one out of two (or at most n-1 out of n) DBMS products are divergent. Even if all fail but 'differently' the failure will still be divergent.
- *Non-divergent failures*: the ones for which two (or more) DBMS products fail with identical symptoms. For some bugs, all demands we ran caused non-divergent failures, for others only some demands did. In the tables that follow we use the labels "non-divergent – *all demands*" and "non-divergent – *some demands*" for these two cases.

All the *divergent* or *self-evident* failures are *detectable* by a client of the database server when at least two replicas of the database are available, on different DBMS products. Failures that are *non-divergent* and *non-self-evident* are *non-detectable*.

3.2 The first study

3.2.1 Description of the study

In our first study (Gashi, Popov et al. 2004b) we used four DBMS products: two commercial (Oracle 8.0.5 and Microsoft SQL Server 7, without any service packs applied) and two open-source ones (PostgreSQL Version 7.0.0 and Interbase Version 6.0). Interbase, Oracle and MSSQL were all run on the Windows 2000 Professional operating system; PostgreSQL 7.0.0 (not available for Windows) was run on RedHat Linux 6.0 (Hedwig). We use the following abbreviated identifiers (for PostgreSQL we

include the release number in the identifier since we will report later on results of one of its later releases):

- *PG 7.0* - for PostgreSQL 7.0.0
- *IB* - for Interbase 6.0
- *OR* - for Oracle 8.0.5
- *MS* - for Microsoft SQL Server 7

For each of these DBMS products there is an accessible repository of reports of known bugs. We collected the IB bugs from SourceForge (SourceForge), the PG 7.0 bugs from its mailing list, (PostgreSQL), MS bugs from its service packs site (Microsoft) and OR bugs from the Oracle Metalink (Oracle).

We only used bugs that caused failure of a DBMS product's core engine. Other bugs, e.g. causing failures of a client application tool, various connectivity (JDBC/ODBC etc.) or installation-specific bugs were not included in the study, because in a future fault-tolerant architecture these functions would be provided by the middleware.

For each reported bug, we attempted to run the corresponding bug script. Full details are available in (Gashi 2006) (***and also provided as Appendix A of this thesis***).

3.2.2 Detailed results

In total, we included in the study 181 bug reports: 55 for IB, 57 for PG, 51 for MS and 18 for OR. None of these bugs was reported for more than one DBMS product. Out of these 181 bugs, 70 were dialect-specific (could be run in only one of the four DBMS products); 58 could be run in all four DBMS products; 26 could be run in only two DBMS products and 27 in only three DBMS products.

Table 5 contains the results of the first study. The structure of the table is as follows. Each grey column lists the results produced when the bugs reported for a certain DBMS product were run on that DBMS product. For example, we collected 55 known IB bugs, of which, when run on our installation of IB, 8 did not cause failures (Unreproduced). The 47 bugs that caused failures are further classified in the part of the column below the double horizontal line, after the "Failure observed" row. Performance failures and engine crashes are self-evident. Incorrect Result failures and "Other" failures can be self-evident or non-self-evident, depending on whether the DBMS product gives an error message.

To the right of the grey column, three columns present the results of running the IB bugs on the other three DBMS products. For example, we can see that out of 55 IB bugs, 24 cannot be run in PG 7.0 (dialect-specific bugs). Of the other 31, which we ran in PG 7.0: 3 are classified as "Undecided Performance" meaning that the bug report indicated a "performance failure" but we could not decide, from the query plan and observed response time, whether a performance failure also occurs in PG 7.0; 27 did not cause a failure in PG 7.0; only 1 caused a failure in both IB and PG 7.0. The table shows that this particular failure was a non-self-evident incorrect result. Details about the bugs causing coincident failures were given in (Gashi, Popov et al. 2004b).

As for the failure types, we can see that most of the bugs for each DBMS product cause Incorrect Result failures. The percentage of non-self-evident failures is also high: they range from 44% for MS to 66% for IB. Engine crashes are less frequent: they range from 13% for MS to 21.5% for OR.

Table 5 - Study 1: Results of running the bug scripts on all four DBMS products. Abbreviations: IB – Interbase 6.0; PG 7.0 - PostgreSQL 7.0.0; OR – Oracle 8.0.5; MS – Microsoft SQL Server 7.

			IB	PG 7.0	OR	MS	PG 7.0	IB	OR	MS	OR	IB	MS	PG 7.0	MS	IB	OR	PG 7.0
Total bug scripts			55	55	55	55	57	57	57	57	18	18	18	18	51	51	51	51
Bug script cannot be run (Functionality Missing)			n/a	24	21	17	n/a	33	27	24	n/a	14	14	13	n/a	36	35	31
Total bug scripts run			55	31	34	38	57	24	30	33	18	4	4	5	51	15	16	20
Undecided performance			0	3	3	3	0	0	0	0	0	0	0	1	0	3	4	2
No failure observed			8	27	31	33	5	24	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

3.2.3 Implications for fault tolerance: two-version combinations

We now look more closely at the two-version combinations of the four different DBMS products. We want first to find out how many of the coincident failures are *detectable*

(i.e. *divergent* or *self-evident*) in the two-version systems. Table 6 contains a summary of the results on each of the six possible two-version combinations[12].

Only twelve coincident failures occurred (note that there were thirteen bugs that caused failures in a different DBMS product than the one for which they were reported (as detailed in Table 5); one bug (MS bug report 56775) (Gashi 2006), although reported for MS, did not cause failure in MS (Unreproduced) but did cause failure in PG 7.0); only four of these twelve are non-detectable. We can see that diversity allows detection of failures for at least 95% of these bugs (41 out of 43, for the IB+MS pair). Moreover, it would support masking and forward recovery (following the self-evident failure of a single channel) for a fraction of bugs varying between 11/32 (34%) for the IB+OR pair) and 11/18 (61%) for the OR+MS pair. More details on these bugs are in (Gashi, Popov et al. 2004b) and (Gashi 2006).

Table 6 - Study 1: summary of results for the two-version combinations.
abbreviations: *s.e. – self-evident failure*; *n.s.e. - non-self-evident failure.*

Pairs of DBMS Products	Total number of bug scripts run	Bugs scripts causing failure (in at least one DBMS product)	One out of two DBMS products failing		Both DBMS products failing						
					Non – Divergent				Divergent		
					All Demands		Some Demands		1 s.e. & 1 n.s.e.	2 s.e.	2 n.s.e.
			s.e.	n.s.e.	s.e.	n.s.e.	s.e.	n.s.e.			
IB + PG 7.0	71	49	22	26	0	1	0	0	0	0	0
IB + OR	69	32	11	21	0	0	0	0	0	0	0
IB + MS	78	43	17	23	1	2	0	0	0	0	0
PG 7.0 + OR	72	33	16	16	0	0	0	0	0	0	1
PG 7.0 + MS	85	48	20	21	0	1	0	0	3	3	0
OR + MS	80	18	11	7	0	0	0	0	0	0	0

3.3 The second study

3.3.1 Description of the study

To repeat the study on later releases of DBMS products, we collected 92 new bug reports for the later releases of the open-source DBMS products: PostgreSQL 7.2 and Firebird 1.0 (abbreviated as PG 7.2 and FB respectively; Firebird is the open-source descendant of Interbase 6.0. The later releases of Interbase are issued as closed-development by Borland). We excluded the closed-development DBMS products as most of their bug

[12] Here we only include bugs (reported for any of the four DBMS products) that could be run on both DBMS products, i.e. we exclude dialect-specific bugs. For instance, Table 6 shows that a total of 71 bugs could be run on both IB and PG 7.0. In detail, 31 of these were reported for IB and 24 for PG; these two numbers can be deduced from Table 5. The remaining 16 were bugs of either OR or MS which could be run on both IB and PG 7.0 – these numbers are not directly deducible from Table 5 due to some bugs being dialect-specific for one DBMS product but not another; they can however be obtained from (Gashi 2006) ***or Appendix A (Table A2)***.

reports lacked the bug scripts needed to trigger the faults. But we still translated the new bug scripts of bugs reported for the open-source DBMS products into the dialects of the closed-development ones, and ran them in the releases used in our first study (Oracle 8.0.5 and MSSQL 7.0). The results of the second study are shown in Table 7 (for full details see (Gashi 2006)). The classifications of faults and failures are as defined in Section 3.1.

Incorrect results are still the most frequent failures. Engine crashes are slightly more frequent than in the first study but still no more than 22.2%. The number of non-self-evident failures is lower than in the first study: 35% for PG 7.2 and 53% in FB. The number of bugs causing coincident failures was again low: in the second study we observed a total of 5 coincident failures. None of the bugs caused failures in more than two DBMS products. The coincident failures are detailed in Section 3.3.3.

Table 7 - Study 2: results of running the bug scripts of FB and PG on all four DBMS products. Abbreviations: FB – Firebird 1.0; PG 7.2 - PostgreSQL 7.2; OR – Oracle 8.0.5; MS – Microsoft SQL Server 7

			FB	PG 7.2	OR	MS	PG 7.2	FB	OR	MS
Total bug scripts			43	43	43	43	49	49	49	49
Bug script cannot be run (Functionality Missing)			n/a	12	15	13	n/a	29	29	30
Total bug scripts run			43	31	28	30	49	20	20	19
Undecided performance			0	1	2	1	0	2	2	0
No failure observed			4	29	26	27	4	17	18	18
Failure observed			39	1	0	2	45	1	0	1
Types of failures	Poor Performance		4	0	0	0	5	0	0	0
	Engine Crash		6	0	0	1	10	0	0	1
	Incorrect Result	Self-evident	7	0	0	0	13	1	0	0
		Non-self-evident	20	1	0	1	15	0	0	0
	Other	Self-evident	1	0	0	0	1	0	0	0
		Non-self-evident	1	0	0	0	1	0	0	0

3.3.2 Implications for fault tolerance: two-version combinations

Table 8 shows the results of the two-version combinations of the 4 DBMS products used in the second study. None of the bugs caused non-detectable failures for all demands. Here there are some bugs that are "non-divergent" for "some demands" only. One caused non-detectable failure only for a few demands in the common failure region, but detectable failure on the others. Three bugs caused self-evident failures in both DBMS products and one caused non-self-evident failure in one and self-evident failure in the other.

So, diversity allows detection of failures for all these bugs. It would allow masking and forward recovery (following the self-evident failure of a single channel) for a fraction of bugs varying between 11/28 (39%) for the FB+MS pair and 12/20 (60%) for the PG 7.2+MS pair.

Table 8 - Study 2: summary of results for the two-version combinations. abbreviations: *s.e. – self-evident failure*; *n.s.e. - non-self-evident failure.*

Pairs of DBMS Products	Total number of bug scripts run	Bugs scripts causing failure (in at least one DBMS product)	One out of two DBMS products failing		Both DBMS products failing						
					Non – Divergent				Divergent		
					All Demands		Some Demands		1 s.e. & 1 n.s.e.	2 s.e.	2 n.s.e.
			s.e.	n.s.e.	s.e.	n.s.e.	s.e.	n.s.e.			
FB + PG 7.2	51	47	26	19	1	0	0	1	0	0	0
FB + OR	46	25	10	15	0	0	0	0	0	0	0
FB + MS	46	28	11	15	0	0	1	0	1	0	0
PG 7.2 + OR	47	21	13	8	0	0	0	0	0	0	0
PG 7.2 + MS	47	20	12	7	0	0	1	0	0	0	0

3.3.3 Common bugs

It is interesting to describe in some more detail some of the bugs that caused coincident failures, listed in Table 9, and speculate about the probable frequency and severity of the failure observed (similar accounts for bugs in Study 1 are in our preliminary report (Gashi, Popov et al. 2004b)).

Table 9 - Bugs that cause coincident failures

For which DBMS product was the bug reported?	On which additional DBMS product was failure observed?			
	FB	PG	OR	MS
FB	N/A	1 - (Bug ID 926001)	0	2– (BugIDs 910423, 926624)
PG	1 (Bug report date 16/05/2003)	N/A	0	1 – (BugID 847)

Arithmetic-related bugs

Firebird bug 926001 (Gashi 2006) causes non-self-evident failure in both FB and PG 7.2 when the DBMS product is asked to add two values of type Timestamp (a timestamp value contains both date and time information). Due to rounding errors, FB always gives a result that is 1 second less than the correct result, whereas PG 7.2 adds the dates but not the time of the second timestamp value (i.e. it treats the operation as $Timestamp_1 + Date_2$). The failure rate for this bug would be highest in applications that require high precision arithmetic computations with timestamp datatypes. On most demands the erroneous results of the two DBMS products would be different: the failure is non-divergent only for some (probably rare) demands.

FB bug 926624 (Gashi 2006) causes a crash in both FB and MS. The crash is due to a stack overflow from attempting to use in the column part of the SELECT statement an arithmetic expression longer than: 8000 characters in FB; 2834 characters in MS. Therefore FB fails for a smaller set of demands than MS. The expected correct behaviour is for the DBMS product to process the statements, or to give an error message that warns the user of the maximum allowed expression length. The failure rate for this bug would probably be low for most installations, as SELECT statements would seldom contain such long arithmetic expressions.

Miscellaneous bugs

FB bug 910423 (Gashi 2006) causes failure in both FB and MS. Fig. 2 shows the demands for which they fail. The failure consists in allowing the datatype of a table column to be changed from integer to string even when the string type is specified to be shorter than needed to hold the data already stored in the column. The expected correct behaviour is for the DBMS product to refuse (with an error message) to change the datatype of either any column that already contains data, or at least those containing data that wouldn't fit in the new length specified. If a client later tries to read the column affected, the two DBMS products react differently: FB responds with an error message (self-evident failure), while MS returns a '*' symbol. We have therefore classified the failure as divergent. As shown, MS actually fails on a superset of the demands on which FB does. It is difficult to conjecture how often applications change the datatypes of columns and hence the likely failure rates for this bug. The severity of this failure is different in the two DBMS products. FB does not lose the data stored in the column: if you just change the type again to a long enough string (>=10 in the example above) then the data can again be read. MS instead truncates the data item to the new length, so that it is irremediably lost.

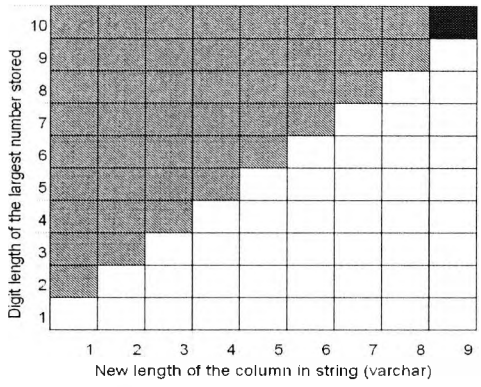

Fig. 2 - FB bug 910423: demands on which MS fails (light grey shaded boxes) and demands for which both FB and MS fail (dark grey).

PG 7.2 bug 847 (Gashi 2006) causes failure in both PG 7.2 and MS. PG 7.2 allows the creation of exceptions that return a message longer than 4000 characters, but then crashes if the exception is raised. The correct behaviour is for a DBMS product to give an error message once its maximum length for exception messages is reached: either when the exception is defined or when attempting to raise the exception. The same problem occurs in MS, but the threshold message length is even smaller (440 characters), and thus failures would be more frequent.

The PG 7.2 bug reported on 16 May 2003 (with no ID in the PG 7.2 mailing list (Gashi 2006)) causes an error message in PG 7.2 and FB, although no error exists. The bug script is given below. The UPDATE statement causes the contents of the database to violate the UNIQUE CONSTRAINT (a constraint over a set of columns requiring that no two values for different rows be equal) at some intermediate stage, although the final state does not violate it:

```
CREATE TABLE TEST2 (V1   INT, V2   INT, CONSTRAINT UQ_TEST UNIQUE (V1,V2));
INSERT INTO TEST2 VALUES (0,0);
INSERT INTO TEST2 VALUES (0,1);
INSERT INTO TEST2 VALUES (0,2);
UPDATE TEST2 SET V2=V2+2;
Violation of UNIQUE KEY constraint "UQ_TEST" on table "TEST2"
```

OR and MS correctly execute the script without error messages; PG 7.2 and FB perform the UNIQUE CONSTRAINT checks at intermediate states (in this case after each row is updated), which causes the exception to be raised. The failure is not specific to this bug script. It can be triggered with any UNIQUE CONSTRAINT on integer, real or float datatypes affecting single or multiple columns, whenever an update is attempted that will

(at an intermediate step during the execution) set a value of a row to that of an existing row in the table, although at the end of the execution of the statement no violations would be present. On every set of parameter values that we tried, either both DBMS products failed or neither did. The failure rate for this bug is expected to be relatively high in update-intensive applications if UNIQUE CONSTRAINT is used.

3.4 Newer vs. older releases (open-source DBMS products)

We now look more closely at those DBMS products that were used in both studies (i.e. the two open-source products). We ran all the new bugs reported for the newer releases on the older releases, to check how many already existed there. The results are in the leftmost eight columns of Table 10 (full details are in (Gashi 2006)).

The structure of Table 10 is the same as that of Table 5 and Table 7. We can see that 33 bugs reported for FB also cause failure in the older release IB. Of the six that do not cause failures in IB, four were Unreproduced in FB. So only 2 bugs that caused failure in FB (the new release) appear to be new bugs, introduced in functionalities that used to work correctly. The reason might be that FB 1.0 was mainly a bug fix release, with no major enhancements, which probably also reduced the number of new problems that could be introduced.

The situation is different for PG 7.2, which featured many more enhancements, for example the support for OUTER JOINS in SELECT statements. We can see that 13 of the bugs reported for PG 7.2 cannot be run at all in the older release (they affect newly added functionality) and, more importantly, 17 of the other 36 bugs do not cause failures in the older release (2 of them are Unreproduced in both releases). This means that the development of the newer release introduced many bugs in functionality that used to work correctly in the old release.

We also ran the old bugs in the new releases of the DBMS products to see how many had been fixed. The results are in the rightmost eight columns of Table 10 (full details are in (Gashi 2006)).

More PG 7.0 bugs were fixed in PG 7.2 than the IB bugs fixed in FB. This high number of fixes, with the attendant risk of new bugs, may be one cause of the relatively many PG 7.2 bugs affecting pre-existing PG 7.0 functionalities(*cf.* the first half of Table 10).

Table 10 - The results of running the new scripts reported for FB and PG 7.2 on the older releases (IB and PG 7.0 respectively) And the bugs reported for the old releases on the new ones Abbreviations: FB – Firebird 1.0; IB – Interbase 6.0; PG 7.0 - PostgreSQL 7.0.0; PG 7.2 - PostgreSQL 7.2.

			FB	IB	PG 7.2	PG 7.0	PG 7.2	PG 7.0	FB	IB	IB	FB	PG 7.0	PG 7.2	PG 7.0	PG 7.2	IB	FB
Total bug scripts			43	43	43	43	49	49	49	49	55	55	55	55	57	57	57	57
Bug script cannot be run (Functionality Missing)			n/a	4	12	26	n/a	13	29	29	n/a	n/a	24	21	n/a	n/a	33	33
Total bug scripts run			43	39	31	17	49	36	20	20	55	55	31	34	57	57	24	24
Undecided performance			0	0	1	1	0	0	2	2	0	0	3	3	0	0	0	0
No failure observed			4	6	29	16	4	17	17	17	8	33	27	31	5	40	24	24
Failure observed			39	33	1	0	45	19	1	1	47	22	1	0	52	17	0	0
Types of failures	Poor Performance		4	3	0	0	5	1	0	0	3	2	0	0	0	0	0	0
	Engine Crash		6	6	0	0	10	3	0	0	7	2	0	0	11	2	0	0
	Incorrect Result	Self-evident	7	6	0	0	13	8	1	1	4	2	0	0	14	6	0	0
		Non-self-evident	20	16	1	0	15	5	0	0	23	10	1	0	20	5	0	0
	Other	Self-evident	1	1	0	0	1	1	0	0	2	2	0	0	2	0	0	0
		Non-self-evident	1	1	0	0	1	1	0	0	8	4	0	0	5	4	0	0

3.4.1 Implications for fault tolerance: the open-source two-version combinations

Table 11 shows the results for all the bugs, from both studies, that could be run on the various open-source combinations.

The first two rows concern the pairs of different releases of the same DBMS product. For PostgreSQL, we see that out of 93 bugs that caused failure in at least one of the releases, 7.0 or 7.2, only 35 cause failures in both; 58 bugs cause failures in only one of the releases. So, using diverse releases of the same DBMS product in a fault-tolerant configuration, as discussed in Section 2, does provide some protection against upgrade problems and can help to assure higher dependability. However there are still many bugs causing failures in both releases of the same DBMS product:

- 57 in Interbase/Firebird
- 35 in PostgreSQL.

This can be compared with the four DBMS product pairs using different DBMS products (last four rows in Table 11), where we get at most 2 bugs that cause coincident failures. This is because:

- The IB 6.0 bug 223512(2) which caused non-divergent coincident failure in IB 6.0 and PG 7.0, has been fixed in the newer releases of both DBMS products.

- The FB 1.0 bug 926001 (Gashi 2006), which causes coincident failure in the new releases FB 1.0 and PG 7.2, did not cause a failure in IB 6.0 and cannot be run in PG 7.0 (dialect-specific).

The main conclusion is to confirm the high level of fault diversity between these DBMS products, and thus potential advantages of a diverse redundant fault-tolerant server. Using different releases of the same DBMS product would also yield dependability gains, but these seem nowhere near as high as the gains that can be achieved by using diverse DBMS products.

Table 11 - Summary of the results of both studies for open-source two-version combinations (abbreviations: *s.e. – self-evident failure*; *n.s.e.- non-self-evident failure*)

Pairs of DBMS Products	Total number of bug scripts run	Bugs scripts causing failure (in at least one DBMS product)	One out of two DBMS products failing		Both DBMS products failing						
					Non – Divergent				Divergent		
					All Demands		Some Demands		1 s.e. & 1 n.s.e.	2 s.e.	2 n.s.e.
			s.e.	n.s.e.	s.e.	n.s.e.	s.e.	n.s.e.			
FB 1.0 + IB 6.0	157	84	8	19	24	33	0	0	0	0	0
PG 7.2 + PG 7.0	164	93	33	25	20	15	0	0	0	0	0
FB 1.0 + PG 7.2	127	65	33	30	1	0	0	1	0	0	0
FB 1.0 + PG 7.0	106	65	34	30	1	0	0	0	0	0	0
IB 6.0 + PG 7.2	127	79	37	41	1	0	0	0	0	0	0
IB 6.0 + PG 7.0	106	77	39	37	0	1	0	0	0	0	0

4. Discussion

The results presented in Section 3 are intriguing and suggest that assembling a fault-tolerant database server from two or more of these OTS DBMS products could yield large dependability gains. But they are not definitive evidence. Apart from the sampling difficulties caused e.g. by lack of certain bug scripts, it is important to clarify to what extent our observations allow us to predict such gains. We gave a detailed discussion of the difficulties in (Gashi, Popov et al. 2004b). In summary:

- the reports available concern *bugs*, not how many *failures* each caused. They do not tell us whether a bug has a large or a small effect on reliability, although the unknown faults – those that have not yet caused failures – would tend to have stochastically lower effect on reliability than those that did cause failures. A better analysis would be obtained from the actual failure reports (including failure counts), if available to the vendors. However, vendors are often wary of sharing such detailed dependability information with their customers;

- less than 100% of the failures that occur, and thus also of the bugs causing them, are reported. However, blatant failures are more likely to be reported than subtle (arguably more dangerous) failures. Therefore failure *under*reporting probably causes a bias towards *under*estimating the frequency of these subtler failures for which diversity would help;
- an organization needs to predict the dependability of its specific installation[s] of a diverse server, compared to a single DBMS product, which depends on the organization's (or each specific installation's) *usage profile*, which differs – perhaps markedly – from the aggregate profile of the user population which generated the bug reports.

How can then individual user organizations decide whether diversity is a suitable option for them, with their specific requirements and usage profiles? As usual for dependability-enhancing measures, the cost is reasonably easy to assess: costs of the DBMS products, the required middleware, difficulties with client applications that require vendor-specific features, hardware costs, run-time cost of the synchronization and consistency-enforcing mechanisms, and possibly more complex recovery after some failures. The gains in improved reliability and availability (fewer system failures and easier recovery from some failures, to be set against possible extra failures due to the added middleware), and possibly less frequent upgrades, are difficult to predict except empirically. Using ballpark figures may provide useful guidelines: there are studies that suggest that the "Total Cost of Ownership" may exceed the initial investment by more than one order of magnitude, and the cost of recovery from failures is a major part of this (Patterson, Brown et al. 2002). This uncertainty will be compounded, for many user organizations, by the lack of trustworthy estimates of their baseline reliability with respect to subtle failures: databases are used with implicit confidence that failures will be self-evident.

Despite all these uncertainties, for some users our evidence already means that a diverse server is a reasonable and relatively cheap precautionary choice, even without good predictions of its effects. These are users who have: serious concerns about dependability (e.g., high costs for interruptions of service or for undetected incorrect data being stored); applications which use mostly the core features common to multiple off-the-shelf DBMS products (recommended by practitioners to improve portability of the applications);

modest throughput requirements for write statements, which make it easy to accept the synchronization delays of a fault-tolerant diverse server.

5. Related work

5.1 Fault tolerance in databases

Fault tolerance in databases has been thoroughly studied and successfully applied in established products. We already mentioned standard database mechanisms such as transaction rollback and retry and checkpointing, which can be used to tolerate faults due to transient conditions. These techniques can be used with or without data replication (discussed in Section 2) in the databases.

5.2 Interoperability between databases

Due to the incompatibilities between the SQL "dialects" of different DBMS products we emphasized the need for SQL translators in the middleware of a diverse fault-tolerant server. Similar ideas have been applied for increasing interoperability between DBMS products (EnterpriseDB 2006), (Janus-Software 2006): the grammar of a DBMS product is re-defined to make it compatible with that of another DBMS product, while keeping the core DBMS product engine unchanged.

5.3 Design diversity

Fault tolerance through design diversity has been studied for over 30 years. The literature is vast: the interested reader can refer to survey papers about the effectiveness of design diversity (Littlewood, Popov et al. 2001), and about design aspects (Strigini 2005). More recent results on the effectiveness of design diversity include measurements with very large populations of amateur programmers (van der Meulen, Bishop et al. 2004), and more detailed probabilistic models on how development affects the reliability of fault-tolerant software (Popov and Littlewood 2004). The literature points at substantial reliability gains from diversity, although it cautions on the difficulty of predicting them, since independence of failures between diverse versions should not be expected.

Our study differs from the earlier experimental studies in three main ways:

- we study large software products - DBMS products - rather than the small programs used in the earlier experiments;
- we study samples of known bug reports, not failures observed during testing;
- we study coincident failure points or regions rather than defects in source code; this is different, for instance, from the analysis by Brilliant et al (Brilliant, Knight et al. 1990) of the causes of coincident failures in the Knight and Leveson experiment (Knight and Leveson 1986).

5.4 Empirical studies of faults and failures

The usefulness of diversity depends on the frequency of those failures that cannot be tolerated without it. There have been comparatively few studies.

Gray studied the TANDEM NonStop system (with non-diverse replication) (Gray 1986). Over the (unspecified) measurement period, 131 out of 132 faults were "Heisenbugs" and thus tolerated. A later study of field software failures for the Tandem Guardian90 operating system (Lee and Iyer 1995) found that 82 % of the reported failures were tolerated. However, the others caused failure of both non-diverse processes in a Tandem process, and thus system failure.

Other related studies concern the determinism and fail-stop properties of database failures, but, like our study, they concern faults rather than failure measurements. A study (Chandra and Chen 2000) examined fault reports of three applications (Apache Web server, GNOME and MySQL DBMS product). Only a small fraction of the faults (5-14%) were Heisenbugs triggered by transient conditions, that would be tolerated by simple rollback and retry. However, as the authors point out, the reason why they, like us, found few Heisenbugs, might be that people are less likely to report faults that they cannot reproduce. Using instead fault injection, the same authors also found (Chandra and Chen 1998) that a significant number of their injected faults (7%) violated the fail-stop model by writing incorrect data to stable storage. Although this fell to 2% when using the Postgres95 transaction mechanism, 2% is still high for applications with stringent reliability requirements.

5.5 Diversity with off-the-shelf applications

Several research projects have addressed architectures supporting software fault tolerance for OTS software. Some have as their main aim intrusion tolerance, e.g.: HACQIT (Reynolds, Just et al. 2002), which demonstrated diverse replication (with two OTS web servers - Microsoft's IIS and Apache) to detect failures (especially maliciously caused ones) and initiate recovery; SITAR (Wang, Gong et al. 2001), an intrusion-tolerant architecture for distributed services and especially COTS servers; the Cactus architecture (Hiltunen, Schlichting et al. 2000), intended to enhance survivability of applications which support diversity among application modules; DIT (Valdes, M. Almgren et al. 2003), an intrusion-tolerant architecture using diversity at several levels (hardware platform, operating system platform, and web servers); the MAFTIA (Dacier (Editor) 2002) project, which delivered a reference architecture and supporting mechanisms. Others target fault tolerance against mainly accidental faults, e.g.: the BASE approach (Castro, Rodrigues et al. 2003) focuses on supporting state recovery for diverse replicas of components via a common abstract specification of a common abstract state, the initial state value and the behaviour of each component; the GUARDS (Powell, Arlat et al. 1999) and Chameleon (Kalbarczyk, Iyer et al. 1999) architectures aim at supporting multiple application-transparent fault tolerance strategies using COTS hardware and software components.

6. Conclusions

We have reported two studies of samples of bug reports for four popular off-the-shelf SQL DBMS products, plus later releases of two of them. We checked for bugs that would cause common-mode failures if the products were used in a diverse redundant (fault-tolerant) architecture: such common bugs are rare. For most bugs, failures would be detected (and may be masked) by a simple two-diverse configuration using different DBMS products. In summary:

- out of the 273 bug scripts run in both our studies, we found very few bug scripts that affected two DBMS products, and none that affected more than two;

- only five of these bug scripts caused identical, non-detectable failures in two DBMS products; of these five, one caused non-detectable failures on only a few among the demands affected.

The results of the second study, on later releases of the same products, substantially confirmed the general conclusions of the first study: the factors that make diversity useful do not seem to disappear as the DBMS products evolve.

Using successive releases of the *same* product for fault tolerance also appeared useful, although less so. We found a high level of fault diversity between successive releases of PostgreSQL: most of the old bugs had been fixed in the new release; many of the newly reported bugs did not cause failure (or could not be run at all) in the old release. This special form of design diversity is attractive for users who need the SQL "dialectal" features of a specific DBMS product, but gives less dependability benefits than using different products. With data diversity also a possibility, users have various trade-offs available between the wishes to exploit dialectal features and to get effective diversity.

These results must be taken with caution, as discussed in Section 4, and their immediate implications vary between users. Our evidence suggests that the forms of redundancy and diversity discussed here will improve the dependability of DBMS products, perhaps dramatically. For some classes of DBMS installations, diversity could already be recommended as a prudent and cost-effective strategy. Yet, users with "ultra-high-dependability" requirements (Littlewood and Strigini 1993) would still have great difficulty achieving confidence that their requirements are satisfied. Our finding some common faults, however rare, certainly suggests caution. Such users might adopt our proposals, but still retain the database- or client-specific solutions mentioned in Section 2.1. The topic of diversity with OTS software certainly deserves further study.

The need for middleware is an obstacle for users wishing to try out diversity in their applications. But our results provide a good business case for implementing the required middleware software.

The performance penalty due to controlling concurrency via the middleware would be a problem with write-intensive loads, but not if concurrent updates are rare (Stankovic and Popov 2006).

Some other interesting observations include:

- there is strong evidence against the fail-stop failure assumption for DBMS products. The majority of bugs reported, for all products, led to "incorrect result" failures rather than crashes (64.5% vs 17.1% in our first study; 65.5% vs 19% in the second), despite crashes being more obvious to the user. Even though these are bug reports and not failure reports, this evidence goes against the common assumption that the majority of failures are engine crashes, and warrants more attention by users to fault-tolerant solutions, and by designers of fault-tolerant solutions to tolerating subtle and non fail-silent failures;
- it may be worthwhile for vendors to test their DBMS products using the known bug reports for other DBMS products. For example, in the first study we observed 4 MSSQL bugs that had not been reported in the MSSQL service packs (previous to our observation period). Oracle was the only DBMS product that never failed when running on it the reported bugs of the other DBMS products;

Future work that is desirable includes:

- statistical testing of the DBMS products to assess the actual reliability gains from diversity. We have run a few million queries with various loads, including ones based on the TPC-C benchmark, observing no failures (however, significant performance gains appear to be possible from using diverse servers (Gashi, Popov et al. 2004a), (Stankovic and Popov 2006)). These results may not be particularly surprising, since these benchmarks use a limited set of well-exercised features of SQL servers. It would be interesting to repeat the tests with more varied test loads. However, these studies are likely to be most useful with reference to specific application environments, for which the usage profile can be approximated reasonably well;
- developing the necessary middleware components for users to be able to try out data replication with diverse servers in their own installations. Lack of these components is the main practical obstacle to the adoption and practical evaluation of these solutions. There are signs that some DBMS product vendors may also help with this problem: EnterpriseDB (EnterpriseDB 2006) and Fyracle (Janus-Software 2006) are Oracle-mode implementations based on PostgreSQL and

Firebird DBMS engines, respectively. With these solutions the problem with SQL dialects is significantly reduced.

Acknowledgment

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council (EPSRC) via projects DOTS (Diversity with Off-The-Shelf components, grant GR/N23912/01) and DIRC (Interdisciplinary Research Collaboration in Dependability, grant GR/N13999/01) and by the European Union Framework Programme 6 via the ReSIST Network of Excellence (Resilience for Survivability in Information Society Technologies, contract IST-4-026764-NOE). We thank Bev Littlewood, Peter Bishop, David Wright and the anonymous TDSC reviewers (in particular the one who suggested further analyses with one of the bug reports) for their comments on earlier versions of this paper.

References

Ammann, P. E. and J. C. Knight (1988), "*Data Diversity: An Approach to Software Fault Tolerance*", IEEE Transactions on Computers 37(4), pp: 418-425.

Anderson, T. and P. A. Lee (1990), "*Fault Tolerance: Principles and Practice (Dependable Computing and Fault Tolerant Systems, Vol 3)*", Springer Verlag.

Bao, Y., X. Sun and K. S. Trivedi (2005), "*A Workload-based Analysis of Software Aging and Rejuvenation*", IEEE Transactions on Reliability R-54(3), pp: 541-548.

Berenson, H., P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil (1995), "*A Critique of ANSI SQL Isolation Levels*", *in proc.* Int. Conf. on Management of Data (SIGMOD '95).

Bernstein, P. A., V. Hadzilacos and N. Goodman (1987), "*Concurrency Control and Recovery in Database Systems*", Reading, Mass., Addison-Wesley.

Blough, D. M. and G. F. Sullivan (1994), "*Voting Using Predispositions*", IEEE Transactions on Reliability R-43(4), pp: 604-616.

Brilliant, S. S., J. C. Knight and N. G. Leveson (1990), "*Analysis of Faults in an N-Version Software Experiment*", IEEE Transactions on Software Engineering 16(2), pp: 238-247.

Castro, M. and B. Liskov (1999), "*Practical Byzantine Fault Tolerance*", *in proc.* Third Symp. on Operating Systems Design and Implementation, New Orleans, LA, USA, pp: 173-186.

Castro, M., R. Rodrigues and B. Liskov (2003), "*BASE: Using Abstraction to Improve Fault Tolerance*", ACM Transactions on Computer Systems (TOCS) 21(3), pp: 236-269.

Chandra, S. and P. M. Chen (1998), "*How Fail-Stop are Programs*", *in proc.* Int. Symp. on Fault-Tolerant Computing (FTCS '98), IEEE Computer Society Press, pp: 240-249.

Chandra, S. and P. M. Chen (2000), "*Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '00), NY, USA, IEEE Computer Society Press, pp: 97-106.

Cook, J. E. and J. A. Dage (1999), "*Highly Reliable Upgrading of Components*", *in proc.* Int. Conf. on Software Engineering (ICSE '99), IEEE-ACM, pp: 203-212.

Dacier (Editor), M. (2002), "*Design of an Intrusion-Tolerant Intrusion Detection System*", MAFTIA deliverable D10, http://www.maftia.org/deliverables/D10.pdf.

Di Giandomenico, F. and L. Strigini (1990), "*Adjudicators for Diverse-Redundant Components*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '90), Huntsville, Alabama, IEEE, pp: 114-123.

EnterpriseDB (2006), "*EnterpriseDB*", http://www.enterprisedb.com/.

Fekete, A., D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha (2005), "*Making Snapshots Isolation Serialisable*", ACM Transactions on Database Systems (TODS) 30(2), pp: 492 - 528.

Frankl, P., D. Hamlet, B. Littlewood and L. Strigini (1998), "*Evaluating Testing Methods by Delivered Reliability*", IEEE Transactions on Software Engineering 24(8), pp: 586-601.

Gashi, I. (2006), "*Fault Diversity Among Off-The-Shelf SQL Database Servers: Complete Results From Two Studies*", http://www.csr.city.ac.uk/people/ilir.gashi/DBMSBugReports/.

Gashi, I. and P. Popov (2006), "*Rephrasing Rules for Off-The-Shelf SQL Database Servers*", *in proc.* 6th European Dependable Computing Conf. (EDCC-6), Coimbra, Portugal, IEEE Computer Society Press, pp: 139-148.

Gashi, I., P. Popov, V. Stankovic and L. Strigini (2004a), "*On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*", *in* Architecting Dependable Systems II, R. de Lemos, Gacek, C., Romanovsky, A. (Eds.), Springer-Verlag, 3069, pp: 191-214.

Gashi, I., P. Popov and L. Strigini (2004b), "*Fault Diversity Among Off-The-Shelf SQL Database Servers*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '04), Florence, Italy, IEEE Computer Society Press, pp: 389-398.

Gray, J. (1986), "*Why Do Computers Stop and What Can be Done About it?*" *in proc.* Int. Symp. on Reliability in Distributed Software and Database Systems (SRDSDS '86), Los Angeles, CA, USA, IEEE Computer Society Press, pp: 3-12.

Gray, J. and A. Reuter (1993), "*Transaction Processing : Concepts and Techniques*", Morgan Kaufmann.

Hiltunen, M. A., R. D. Schlichting, C. A. Ugarte and G. T. Wong (2000), "*Survivability Through Customization and Adaptability: The Cactus Approach*", *in proc.* DARPA Information Survivability Conference & Exposition.

Janus-Software (2006), "*Fyracle*", http://www.janus-software.com/fb_fyracle.html.

Jimenez-Peris, R., M. Patino-Martinez and G. Alonso (2002), "*An Algorithm for Non-Intrusive, Parallel Recovery of Replicated Data and its Correctness*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '02), Osaka, Japan, IEEE Computer Society Press, pp: 150-159.

Jimenez-Peris, R. and M. Patino-Martinez (2003), "*D5: Transaction Support*", ADAPT Middleware Technologies for Adaptive and Composable Distributed Components, Deliverable IST-2001-37126.

Jimenez-Peris, R., M. Patino-Martinez, G. Alonso and B. Kemme (2002), "*Scalable Database Replication Middleware*", *in proc.* 22nd Int. Conf. on Distributed Computing Systems, Vienna, Austria, IEEE Computer Society Press, pp: 477-484.

Kalbarczyk, Z. T., R. K. Iyer, S. Bagchi and K. Whisnant (1999), "*Chameleon: A Software Infrastructure for Adaptive Fault Tolerance*", IEEE Transactions on Parallel Distributed Systems 10(6), pp: 560-579.

Kemme, B. and G. Alonso (2000), "*Don't be Lazy, be Consistent: Postgres-R, a New Way to Implement Database Replication*", *in proc.* Int. Conf. on Very Large Databases (VLDB), Cairo, Egypt.

Knight, J. C. and N. G. Leveson (1986), "*An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming*", IEEE Transactions on Software Engineering 12(1), pp: 96-109.

Lee, I. and R. K. Iyer (1995), "*Software Dependability in the Tandem GUARDIAN System*", IEEE Transactions on Software Engineering 21(5), pp: 455-467.

Lin, Y., B. Kemme, M. Patino-Martínez and R. Jiménez-Peris (2005), "*Middleware based Data Replication providing Snapshot Isolation*", *in proc.* Int. Conf. on Management of Data (SIGMOD '05), Baltimore, Maryland, USA, ACM Press, pp: 419-430.

Littlewood, B., P. Popov and L. Strigini (2001), "*Modelling software design diversity - a review*", ACM Computing Surveys 33(2), pp: 177-208.

Littlewood, B. and L. Strigini (1993), "*Validation of Ultra-High Dependability for Software-based Systems*", Communications of the ACM 36(11), pp: 69-80.

Microsoft, "*List of Bugs Fixed by SQL Server 7.0 Service Packs*", http://support.microsoft.com/default.aspx?scid=kb;EN=US;313980.

Oracle, "*Oracle Metalink*", http://metalink.oracle.com/metalink/plsql/ml2_gui.startup.

Parhami, B. (2005), "*Voting: A Paradigm for Adjudication and Data Fusion in Dependable Systems*", *in* Dependable Computing Systems: Paradigms, Performance Issues, and Applications, H. B. Diab and A. Y. Zomaya (Eds.).

Patino-Martinez, M., R. Jiménez-Peris, B. Kemme and G. Alonso (2005), "*MIDDLE-R: Consistent Database Replication at the Middleware Level*", ACM Transactions on Computer Systems 23(4), pp: 375-423.

Patterson, D., A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kýcýman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman and N. Treuhaft (2002), "*Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques and Case Studies*", UC Berkeley Computer Science, CSD-02-1175.

Pedone, F. and S. Frolund (2000), "*Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '00), Nurnberg, Germany, IEEE Computer Society, pp: 176-85.

Popov, P. and B. Littlewood (2004), "*The effect of testing on the reliability of fault-tolerant software*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '04), Florence, Italy, IEEE Computer Society Press, pp: 265-274.

Popov, P., L. Strigini, A. Kostov, V. Mollov and D. Selensky (2004), "*Software Fault-Tolerance with Off-the-Shelf SQL Servers*", *in proc.* Int. Conf. on COTS-based Software Systems (ICCBSS '04), Redondo Beach, CA USA, Springer, pp: 117-126.

Popov, P., L. Strigini, S. Riddle and A. Romanovsky (2001), "*Protective Wrapping of OTS Components*", *in proc.* 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction, Toronto.

Popov, P., L. Strigini and A. Romanovsky (2000), "*Diversity for Off-The-Shelf Components*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '00) - Fast Abstracts supplement, New York, NY, USA, IEEE Computer Society Press, pp: B60-B61.

PostgreSQL, "*PostgreSQL Bugs Mailing List Archives*", http://archives.postgresql.org/pgsql-bugs/.

Powell, D., J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac and A. Wellings (1999), "*GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems*", IEEE Transactions on Parallel and Distributed Systems 10(6), pp: 580-599.

Reynolds, J., J. Just, E. Lawson, L. Clough, R. Maglich and K. Levitt (2002), "*The Design and Implementation of an Intrusion Tolerant System*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '02), Washington, D.C., USA, IEEE Computer Society Press, pp: 285-292.

Schneider, F. (1984), "*Byzantine Generals in Action: Implementing Fail-Stop Processors*", ACM Transactions on Computer Systems 2(2), pp: 145-154.

SourceForge, "*Interbase (Firebird) Bug tracker*", http://sourceforge.net/tracker/?atid=109028&group_id=9028&func=browse.

Stankovic, V. and P. Popov (2006), "*Improving DBMS Performance through Diverse Redundancy*", *in proc.* Int. Symposium on Reliable Distributed Systems (SRDS '06), Leeds, UK, IEEE Computer Society, pp: 391-400.

Strigini, L. (2005), "*Fault Tolerance Against Design Faults*", *in* Dependable Computing Systems: Paradigms, Performance Issues, and Applications, H. Diab and A. Zomaya (Eds.), J. Wiley & Sons, pp: 213-241.

Sutter, H. (2000), "*SQL/Replication Scope and Requirements Document*", ISO/IEC JTC 1/SC 32 Data Management and Interchange WG3 Database Languages, H2-2000-568.

Tai, A. T., K. S. Tso, L. Alkalai, S. N. Chau and W. H. Sanders (2002), "*Low-Cost Error Containment and Recovery for Onboard Guarded Software Upgrading and Beyond*", IEEE Transactions on Computers 51(2), pp: 121-137.

Tso, K. S. and A. Avizienis (1987), "*Community Error Recovery in N-Version Software: A Design Study with Experimentation*", *in proc.* Int. Symp. on Fault-Tolerant Computing (FTCS '87), Pittsburgh, PA, USA, pp: 127-133.

Valdes, A., M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saidi, V. Stavridou and T. E. Uribe (2003), "*An Architecture for an Adaptive Intrusion-Tolerant Server*", *in* LNCS 2845 - Selected Papers from 10th Int. Workshop on Security Protocols '02, B. Christianson, Crispo, B., Malcolm, J. A., Roe, M. (Eds.), Springer, pp: 158-178.

van der Meulen, M. J. P., P. G. Bishop and M. Revilla (2004), "*An Exploration of Software Faults and Failure Behaviour in a Large Population of Programs*", *in proc.* Int. Symp. on Software Reliability Engineering (ISSRE '04), Rennes, France, Springer-Verlag, pp: 101-112.

Wang, F., F. Gong, C. Sargor, K. Goseva-Popstojanova, K. Trivedi and F. Jou (2001), "*SITAR: A Scalable Intrusion-Tolerant Architecture for Distributed Services*", *in proc.* 2001 IEEE Workshop on Information Assurance and Security, West Point, New York, U.S.A.

Weismann, M., F. Pedone and A. Schiper (2000), "*Database Replication Techniques: a Three Parameter Classification*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '00), Nurnberg, Germany, IEEE Computer Society Press, pp: 206-217.

V. Architectural Aspects of a Fault-Tolerant Diverse SQL Server

Paper-3. On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers

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.*

Co-authors: Dr. Peter Popov, Mr. Vladimir Stankovic, Prof. Lorenzo Strigini

Book: Architecting Dependable Systems II

Series: Lecture Notes in Computer Science

Date of submission: December-2003

Status: *Published*

Number of reviewers: 2

Publication date: October 2004

Full citation: Gashi I., Popov P., Stankovic V., Strigini L., "*On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*", in "*Architecting Dependable Systems II*", Lecture Notes in Computer Science, (R. de Lemos, C. Gacek and A. Romanovsky, Eds.), vol. 3069, pp. 191-214, Springer-Verlag, 2004

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 (Babbage 1974).

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[13]. 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 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 (Traverse 1988). 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

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

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 (Randell 1975) and is a thoroughly studied subject (Lyu 1995). Many implementations of the idea exist, for instance recovery blocks (Randell 1975), N-version programming (Avizienis and Kelly 1984) and self-checking modular redundancy (Laprie, Arlat et al. 1990).

Over the years, diverse redundancy has found its way to various industrial applications (Voges 1988). 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 (Avizienis, Gunningberg et al. 1985), (Lyu 1995), (Pullum 2001).

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.[14] 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[15]. We investigate whether design diversity is useful in this domain from the perspectives of dependability and performance.

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

[14] 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.

[15] 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.

history of database replication (Bernstein, Hadzilacos et al. 1987), effort on standardisation in the area has only started recently (Sutter 2000). Fail-over delivers some improvement over non-replicated servers although limited effectiveness has been observed in some cases (Kalyanakrishnam, Kalbarczyk et al. 1999). Fail-over can be used as a recovery strategy irrespective of the type of failure (not necessarily "fail-stop" (Schneider 1984)). 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 (Gashi, Popov et al. 2004) (***the preceding reference forms part of this thesis as Paper-1***) 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 (Chandra and Chen 1998) 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 (Gray 1986) 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) (Chandra and Chen 2000) 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[16]) and two commercial SQL servers (Microsoft SQL 7.0 and Oracle 8.0.5). The results of the study are described in detail in (Gashi, Popov et al. 2004). Here we concentrate on the aspects relevant to our discussion.

2.1 SQL servers cannot be assumed to 'Fail-Stop'

Table 12 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.

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.

[16] 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".

- *Non-Self-Evident* failures: incorrect result failures, without server exceptions within an accepted time delay.

(Gashi, Popov et al. 2004) 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 (Gashi, Popov et al. 2004). 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 (Gashi, Popov et al. 2004) seems overwhelming against assuming (until actual failure counts are available) that 'fail-stop' failures are the main concern to be resolved by replication.

Table 12 - 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

2.2 Potential of design diversity for detecting/diagnosing failures

Table 13 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. 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.

Table 13 - 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

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 (Bernstein, Hadzilacos et al. 1987). A recent study compared various replication protocols in terms of their performance and the feasibility of their implementation (Jimenez-Peris, M. Patino-Martinez et al. 2003). One of the oldest

replication protocols, 'Read once write all available (ROWAA)' (Bernstein, Hadzilacos et al. 1987) 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 (Bernstein, Hadzilacos et al. 1987). This may lead to a very significant improvement; with up to a fivefold reduction in execution time of the update statements (Jimenez-Peris, Patino-Martinez et al. 2001), (Kemme and Alonso 2000). 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. 3.

Fig. 3 - 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 (Anderson and Lee 1990).

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 (Gray and Reuter 1993) 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 (Tso and Avizienis 1987)). 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 (Sutter 2000). 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 (Bernstein, Hadzilacos et al. 1987). 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 (Bernstein, Hadzilacos et al. 1987). 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" (Jimenez-Peris, M. Patino-Martinez et al. 2002) 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 (Poledna 1996) and there are well known architectural solutions for dealing with it (Powell, Arlat et al. 1999). Empirically (Popov, Strigini et al. 2004), we repeatedly observed data inconsistency even with replication of the same SQL server.

To achieve data consistency, i.e. a 1-copy serialisable history (Bernstein, Hadzilacos et al. 1987) 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 (Popov, Strigini et al. 2004). 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 (Jimenez-Peris, M. Patino-Martinez et al. 2002).

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 (Gruber 2000). Triggers and stored procedures (Melton 2002) 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 (Microsoft 2003).

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 (Popov, Strigini et al. 2004) (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" (Melton 2002) (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. 3). 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 Section 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 (Popov, Strigini et al. 2004), or by the client applications by not using auto fields at all.

This is just one case of diversity causing violations of *replica determinism* (Poledna 1996); 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 (Ammann and Knight 1988) (who studied it in a different context). The simplest example of the idea in (Ammann and Knight 1988) 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 (Ammann and Knight 1988) 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 re-phrases 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.[17]

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 (Ammann and Knight 1988) 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

[17] 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.

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. 4 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.

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

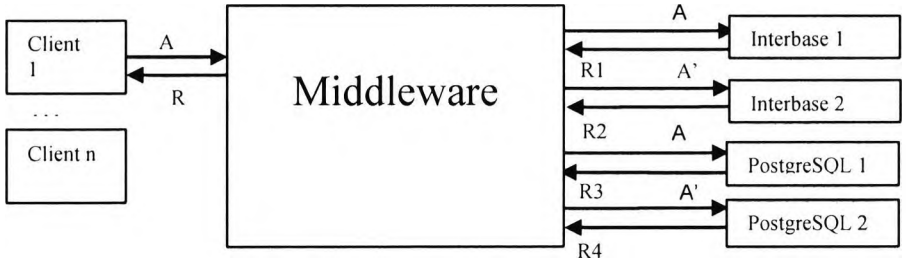

Fig. 4 - A possible design for a fault-tolerant server using diverse SQL servers and data diversity. The original query (*A*) is sent to the pair {*Interbase 1*, *PostgreSQL 1*}, the re-phrased query (*A'*) is sent to the pair {*Interbase 2*, *PostgreSQL 2*}. The *middleware* compares/votes the results in one of the ways described in Section 3.2 for solutions without data diversity

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. 5 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.

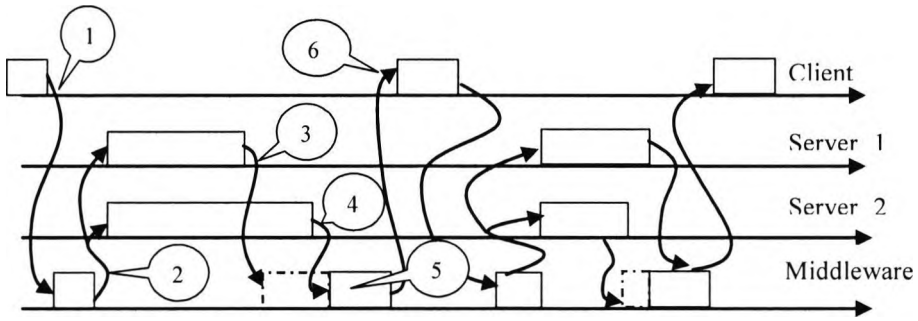

Fig. 5 - Timing diagram with two diverse servers and middleware running in pessimistic regime. The meaning of the arrows is: 1 – the client sends a query to the middleware; 2 – the middleware translates the request to the dialects of the servers and sends the resulting queries, or sequences of queries, 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. 6 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.

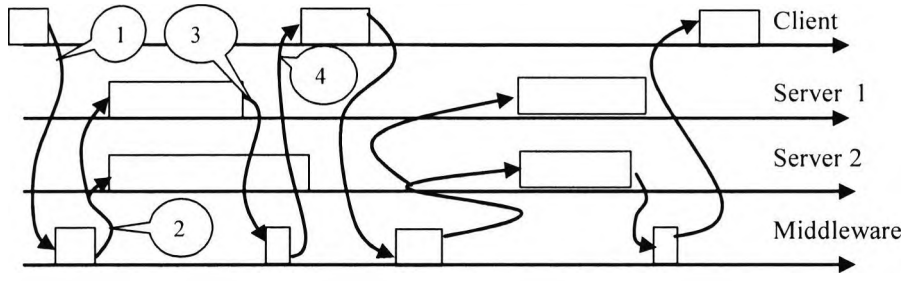

Fig. 6 - Timing diagram with two diverse servers and middleware running in optimistic regime. The meaning of the *arrows* is: *1* – the client sends a query to the middleware, *2* – the middleware translates the request to the dialects of the servers and sends the resulting queries, or sequences of queries, 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. 6). 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 (Chen, Lee et al. 1994). 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 (TPC 2002). 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. 3) 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. 5) and optimistic (Fig. 6). 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. 7)
- mean response times per transaction of each type (Fig. 8).

With *two identical SQL servers* (last two server pairs in Fig. 7), 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.

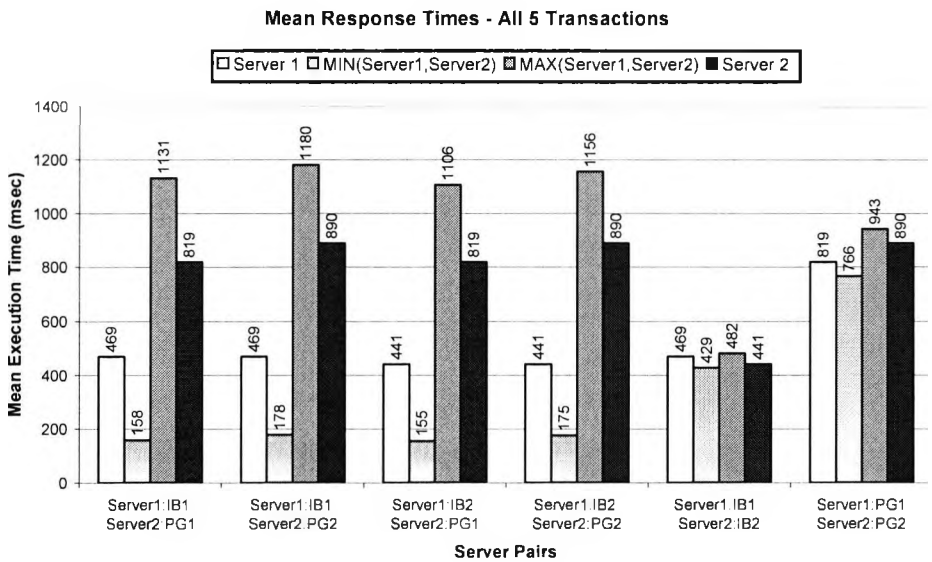

Fig. 7 - Mean response time for all five transaction types over 10,000 transactions for two replicas of Interbase 6.0 and two of PostgreSQL 7.2.4. The X-axis lists the servers grouped as pairs (Server 1 and Server 2). Each server may be of type Interbase (IB) or PostgreSQL (PG). For each of the 6 server 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. 8. 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. 7).

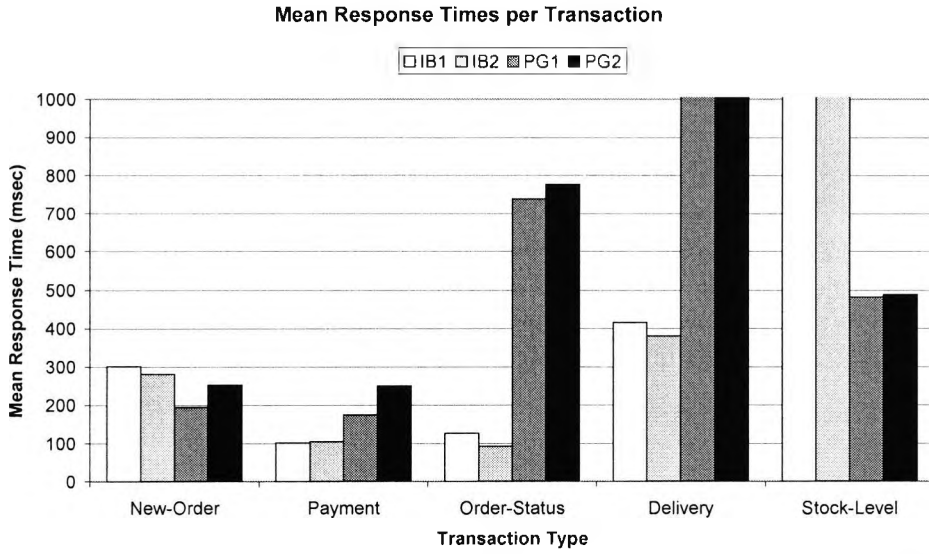

Fig. 8 - Mean response times by two replicas of Interbase 6.0 and PostgreSQL 7.2.4 for all five transactions. The X-axis lists the transaction types (New-Order, Payment, Order-Status, Delivery and Stock-Level). The Y-axis gives the values of the mean response time in milliseconds for each of the servers (IB1, IB2, PG1 and PG2) for a particular transaction type

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. 9. 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.

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).

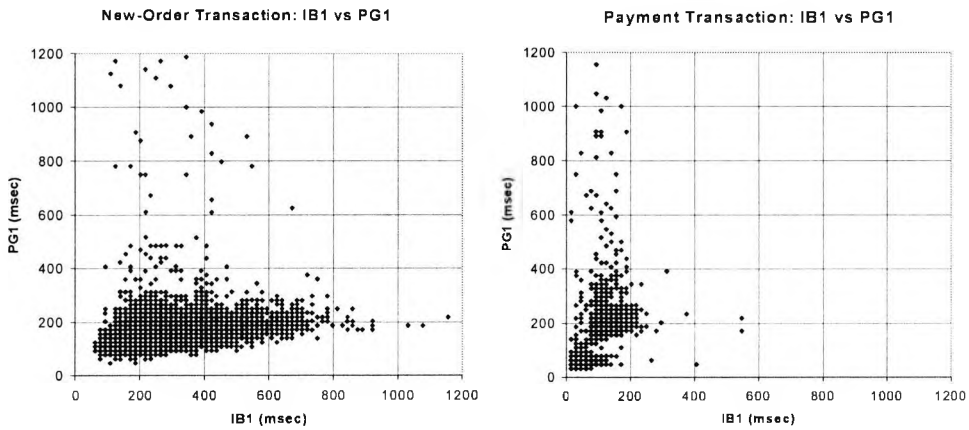

Fig. 9 - Response times for the *New-Order* and *Payment* transactions. Every *dot* in the plots represents the response times of two servers for an *instance* of the respective transaction type. If the times were close to each other most of the dots would be concentrated around the unit slope (observed 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.[18] If the slowest server is still processing the previous query, there are two options:

[18] 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

- 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 (Bernstein, Hadzilacos et al. 1987), (Sutter 2000), (Jimenez-Peris, Patino-Martinez et al. 2001), (Gray and Reuter 1993) and (Kemme and Alonso 2000).

Recent surveys exist of the mechanisms for eager replication of databases (Weismann, Pedone et al. 2000), and for the replication mechanisms (mainly lazy replication)

slower server to 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 various SQL servers (Vaysburd 1999). The Pronto protocol (Pedone and Frolund 2000) 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 (Jimenez-Peris and Patino-Martinez 2003). 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, implemented in (Patino-Martinez, Jimenez-Peris et al. 2000) and (Jimenez-Peris, Patino-Martinez et al. 2002), 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 (Jimenez-Peris, M. Patino-Martinez et al. 2003), (Jimenez-Peris, Patino-Martinez et al. 2001).

The problem of on-line recovery is scrutinised in (Kemme and Alonso 2000) and (Jimenez-Peris, M. Patino-Martinez et al. 2002) 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 (Voas 2000). 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 (Gashi, Popov et al. 2004).

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

Ammann, P. E. and J. C. Knight (1988), "*Data Diversity: An Approach to Software Fault Tolerance*", IEEE Transactions on Computers 37(4), pp: 418-425.

Anderson, T. and P. A. Lee (1990), "*Fault Tolerance: Principles and Practice (Dependable Computing and Fault Tolerant Systems, Vol 3)*", Springer Verlag.

Avizienis, A., P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso and U. Voges (1985), "*The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software*", *in proc.* Int. Symp. on Fault-Tolerant Computing (FTCS '85), Ann Arbor, Michigan, USA, IEEE Computer Society Press, pp: 126-134.

Avizienis, A. and J. P. J. Kelly (1984), "*Fault Tolerance by Design Diversity: Concepts and Experiments*", IEEE Computer 17(8), pp: 67-80.

Babbage, C. (1974), "*On the Mathematical Powers of the Calculating Engine (Unpublished manuscript, December 1837)*", *in* The Origins of Digital Computers: Selected Papers, B. Randell (Eds.), Springer, pp: 17-52.

Bernstein, P. A., V. Hadzilacos and N. Goodman (1987), "*Concurrency Control and Recovery in Database Systems*", Reading, Mass., Addison-Wesley.

Chandra, S. and P. M. Chen (1998), "*How Fail-Stop are Programs*", *in proc.* Int. Symp. on Fault-Tolerant Computing (FTCS '98), IEEE Computer Society Press, pp: 240-249.

Chandra, S. and P. M. Chen (2000), "*Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '00), NY, USA, IEEE Computer Society Press, pp: 97-106.

Chen, P. M., E. K. Lee, G. A. Gibson, R. H. Katz and D. A. Patterson (1994), "*Raid: High-Performance, Reliable Secondary Storage*", ACM Computing Surveys 26(2), pp: 145-185.

Gashi, I., P. Popov and L. Strigini (2004), "*Fault Diversity Among Off-The-Shelf SQL Database Servers*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '04), Florence, Italy, IEEE Computer Society Press, pp: 389-398.

Gray, J. (1986), "*Why Do Computers Stop and What Can be Done About it?*" *in proc.* Int. Symp. on Reliability in Distributed Software and Database Systems (SRDSDS '86), Los Angeles, CA, USA, IEEE Computer Society Press, pp: 3-12.

Gray, J. and A. Reuter (1993), "*Transaction Processing : Concepts and Techniques*", Morgan Kaufmann.

Gruber, M. (2000), "*Mastering SQL*", SYBEX.

Jimenez-Peris, R., M. Patino-Martinez and G. Alonso (2002), "*An Algorithm for Non-Intrusive, Parallel Recovery of Replicated Data and its Correctness*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '02), Osaka, Japan, IEEE Computer Society Press, pp: 150-159.

Jimenez-Peris, R., M. Patino-Martinez, G. Alonso and B. Kemme (2003), "*Are Quorums an Alternative for Data Replication?*" ACM Transactions on Database Systems 28(3), pp: 257-294.

Jimenez-Peris, R. and M. Patino-Martinez (2003), "*D5: Transaction Support*", ADAPT Middleware Technologies for Adaptive and Composable Distributed Components, Deliverable IST-2001-37126.

Jimenez-Peris, R., M. Patino-Martinez, G. Alonso and B. Kemme (2001), "*How to Select a Replication Protocol According to Scalability, Availability and Communication Overhead*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '01), New Orleans, Louisiana, IEEE Computer Society Press, pp: 24 -33.

Jimenez-Peris, R., M. Patino-Martinez, G. Alonso and B. Kemme (2002), "*Scalable Database Replication Middleware*", *in proc.* 22nd Int. Conf. on Distributed Computing Systems, Vienna, Austria, IEEE Computer Society Press, pp: 477-484.

Kalyanakrishnam, M., Z. Kalbarczyk and R. Iyer (1999), "*Failure Data Analysis of LAN of Windows NT Based Computers*", *in proc.* Int. Symp. on Reliable and Distributed Systems (SRDS '99), Lausanne, Switzerland, IEEE Computer Society Press, pp: 178-187.

Kemme, B. and G. Alonso (2000), "*Don't be Lazy, be Consistent: Postgres-R, a New Way to Implement Database Replication*", *in proc.* Int. Conf. on Very Large Databases (VLDB), Cairo, Egypt.

Laprie, J. C., J. Arlat, C. Beounes and K. Kanoun (1990), "*Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures*", IEEE Computer 23(7), pp: 39-51.

Lyu, M. R., Ed. (1995), "*Software Fault Tolerance*", Trends in Software, Wiley.

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

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

Patino-Martinez, M., R. Jimenez-Peris and G. Alonso (2000), "*Scalable Replication in Database Clusters*", *in proc.* Int. Conf. on Distributed Computing, DISC'00, Springer, pp: 315-329.

Pedone, F. and S. Frolund (2000), "*Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '00), Nurnberg, Germany, IEEE Computer Society, pp: 176-85.

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

Popov, P., L. Strigini, A. Kostov, V. Mollov and D. Selensky (2004), "*Software Fault-Tolerance with Off-the-Shelf SQL Servers*", *in proc.* Int. Conf. on COTS-based Software Systems (ICCBSS '04), Redondo Beach, CA USA, Springer, pp: 117-126.

Powell, D., J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac and A. Wellings (1999), "*GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems*", IEEE Transactions on Parallel and Distributed Systems 10(6), pp: 580-599.

Pullum, L. (2001), "*Software Fault Tolerance Techniques and Implementation*", Artech House.

Randell, B. (1975), "*System Structure for Software Fault Tolerance*", IEEE Transactions on Software Engineering 1(2), pp: 220-232.

Schneider, F. (1984), "*Byzantine Generals in Action: Implementing Fail-Stop Processors*", ACM Transactions on Computer Systems 2(2), pp: 145-154.

Sutter, H. (2000), "*SQL/Replication Scope and Requirements Document*", ISO/IEC JTC 1/SC 32 Data Management and Interchange WG3 Database Languages, H2-2000-568.

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

Traverse, P. J. (1988), "*AIRBUS and ATR System Architecture and Specification*", *in* Software diversity in computerized control systems, U. Voges (Eds.), Springer-Verlag, 2, pp: 95-104.

Tso, K. S. and A. Avizienis (1987), "*Community Error Recovery in N-Version Software: A Design Study with Experimentation*", *in proc.* Int. Symp. on Fault-Tolerant Computing (FTCS '87), Pittsburgh, PA, USA, pp: 127-133.

Vaysburd, A. (1999), "*Fault Tolerance in Three-Tier Applications: Focusing on the Database Tier*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '99), Lausanne, Switzerland, IEEE Computer Society Press, pp: 322-327.

Voas, J. (2000), "*Deriving Accurate Operational Profiles for Mass-Marketed Software*", http://www.cigital.com/papers/download/profile.pdf.

Voges, U., Ed. (1988), "*Software diversity in computerized control systems*", Dependable Computing and Fault-Tolerance series, A. Avizienis, H. Kopetz, J.C. Laprie (series Editors), Wien, Springer-Verlag.

Weismann, M., F. Pedone and A. Schiper (2000), "*Database Replication Techniques: a Three Parameter Classification*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '00), Nurnberg, Germany, IEEE Computer Society Press, pp: 206-217.

Paper-4. Rephrasing Rules for Off-The-Shelf SQL Database Servers

Abstract: *We have reported previously (Gashi, Popov et al. 2004b) results of a study with a sample of bug reports from four off-the-shelf SQL servers. We checked whether these bugs caused failures in more than one server. We found that very few bugs caused failures in two servers and none caused failures in more than two. This would suggest a fault-tolerant server built with diverse off-the-shelf servers would be a prudent choice for improving failure detection. To study other aspects of fault tolerance, namely failure diagnosis and state recovery, we have studied the "data diversity" mechanism and we defined a number of SQL rephrasing rules. These rules transform a client sent statement to an additional logically equivalent statement, leading to more results being returned to an adjudicator. These rules therefore help to increase the probability of a correct response being returned to a client and maintain a correct state in the database.*

Co-author: Dr. Peter Popov

Conference: European Dependable Computing Conference 2006 (EDCC-06)

Date of submission: April-2006

Status: *Published*

Number of reviewers: 6

Publication date: October 2006

Full citation: Gashi, I., Popov, P., "*Rephrasing Rules for Off-The-Shelf SQL Database Servers*", in Proc. *6th European Dependable Computing Conference (EDCC-6)*, 18-20 October, Coimbra, Portugal, IEEE Computer Society, pp. 139-148, 2006

1. Introduction

Fault tolerance is frequently the only viable approach of obtaining the required system dependability from systems built out of "off-the-shelf" (OTS) products (Popov, Strigini et al. 2004). There are various methods in which this fault tolerance can be achieved ranging from simple error detection and recovery add-ons (e.g. wrappers (Popov, Strigini et al. 2001)) to diverse redundancy replication using diverse versions of the components. These design solutions are well known. Questions remain, however, about the dependability gains and implementation difficulties for a specific system.

We have studied some of these issues in SQL database servers, a very complex category of off-the-shelf products. We have previously reported (Gashi, Popov et al. 2004b) (***the preceding reference forms part of this thesis as Paper-1***) results from a study with a sample of bug reports from four off-the-shelf SQL servers so as to assess the possible advantages of software fault tolerance - in the form of modular redundancy with diversity - in complex off-the-shelf software. We found that very few bugs cause failures in two servers and none cause failures in more than two, which would indicate that significant dependability improvements can be expected from the deployment of a fault-tolerant server built out of diverse off-the-shelf servers in comparison with individual servers or the non-diverse replicated configurations.

Although we found that using multiple diverse SQL servers can dramatically improve error detection rates it does not make them 100%, e.g. our study (Gashi, Popov et al. 2004b) found four bugs causing identical non-self-evident failures in two servers. Thus there is room for improving failure detection further. Many of the cases, in which a failure was detected did not allow for immediate diagnosis of the failed server. Fault tolerance requires also diagnosing the faulty server and maintaining data consistency among the databases in addition to failure detection. To improve the situation, we studied the mechanism called "data diversity" by Ammann and Knight (Ammann and Knight 1988) (who studied it in a different context). The simplest example of the idea in (Ammann and Knight 1988) refers 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 (Ammann and Knight 1988) to be an effective way of detecting as well as masking failures, i.e. delivering fault-tolerance. Data diversity, thus, can help with failure detection and state recovery, and thus complement fault-tolerance solutions which employ diverse modular redundancy, as well as helping achieve a certain degree of fault tolerance without employing diverse modular redundancy.

Data diversity is applicable to SQL servers because of the inherent redundancy that exists in the SQL language: statements can be "rephrased" into different, but logically equivalent [sequences of] statements. While working with the bug reports we found examples where a particular statement causes a failure in a server but a rephrased version of the same statement does not. Examples of such statements often appear in bug reports as "workarounds".

In this paper we provide details of how SQL rephrasing can be employed systematically in a fault-tolerant server and provide examples of useful rephrasing rules. We also report on performance measurements using the TPC-C (TPC 2002) benchmark client implementation to get some initial estimates of the delays introduced by rephrasing.

The paper is structured as follows: in Section 2 we give details of the architecture of a fault-tolerant server employing rephrasing. In Section 3 we give details of the data diversity study we have conducted for defining SQL rephrasing rules and illustrate how one of these rules has been used as a workaround for two known bugs of two SQL servers. In Section 4 we give some empirical results of experiments we have conducted to measure the performance penalty due to rephrasing. In Section 5 we discuss some general implications of our results and finally in Section 6 some conclusions are presented with possibilities for further work.

2. Architecture of a fault-tolerant server

2.1 General scheme

Data replication is a well-understood subject (Patino-Martinez, Jiménez-Peris et al. 2005), (Lin, Kemme et al. 2005), (Bernstein, Hadzilacos et al. 1987). The main problem

replication protocols deal with is guaranteeing consistency between copies of a database without imposing a strict synchronisation regime between them. A study which compared various replication protocols in terms of their performance and the feasibility of their implementation can be found in (Jimenez-Peris, M. Patino-Martinez et al. 2003). Existing protocols implement efficient solutions for this problem, but depend on running copies of the *same* (non-diverse) server. These schemes would not tolerate non-self-evident[19] failures that cause incorrect writes to the database or that return incorrect results from read statements. For the former, incorrect writes 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. An illustration of this architecture with two diverse Off-The-Shelf servers ("O-servers") is shown in Fig. 10. A brief explanation of the figure follows. Several nodes (computers) are depicted which run client applications (Client node 1, Client node 2 and Client node 3) or server applications (Middleware node, RDBMS 1 node and RDBMS 2 node). The bottom three nodes together form the FT-server. Components may share a node: e.g. Replication Middleware, and the two SQL connectors for dialects 1 and 2 are deployed on the Middleware node. The SQL connectors additionally contain the SQL rephrasing rules. The diagram assumes that the Off-The-Shelf servers (O-servers) run on separate nodes, RDBMS 1 node and RDBMS 2 node. The circles represent the interfaces through which the components interact. Each SQL connector, implements the SQL Connector API interface used by the Replication Middleware component. This, in turn implements the Middleware API interface via which the client applications access the FT-server, either directly or via a driver for the FT-server in a specific run-time environment, e.g. JDBC driver or .NET Provider.

Further improvements to this architecture would be to also run diverse replicas of the middleware component. We have described elsewhere (Gashi, Popov et al. 2004a) (***the preceding reference forms part of this thesis as Paper-3***), (Popov, Strigini et al. 2004) in

[19] In (Gashi, Popov et al. 2004b) we classified the failures according to their detectability by a client of the database servers into: *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.

more detail the FT-node architecture. Here we will only elaborate on the parts relevant to the discussion of rephrasing.

2.2 SQL connectors

The O-servers are not fully compatible: they "speak different dialects" of SQL, despite being compliant at various levels with SQL standards. Therefore the FT-server includes a translator between these dialects, defined for a subset of SQL (e.g. "SQL-92 entry level") plus some more advanced features important for enterprise applications (such as TRIGGERs and STORED PROCEDUREs). The translators are depicted as "SQL Dialect Connector's" in Fig 10.

A similar idea (implemented in (EnterpriseDB 2006), (Janus-Software 2006)) is to redefine the grammar of one database server to make it compatible with that of another while keeping the core database engine unchanged.

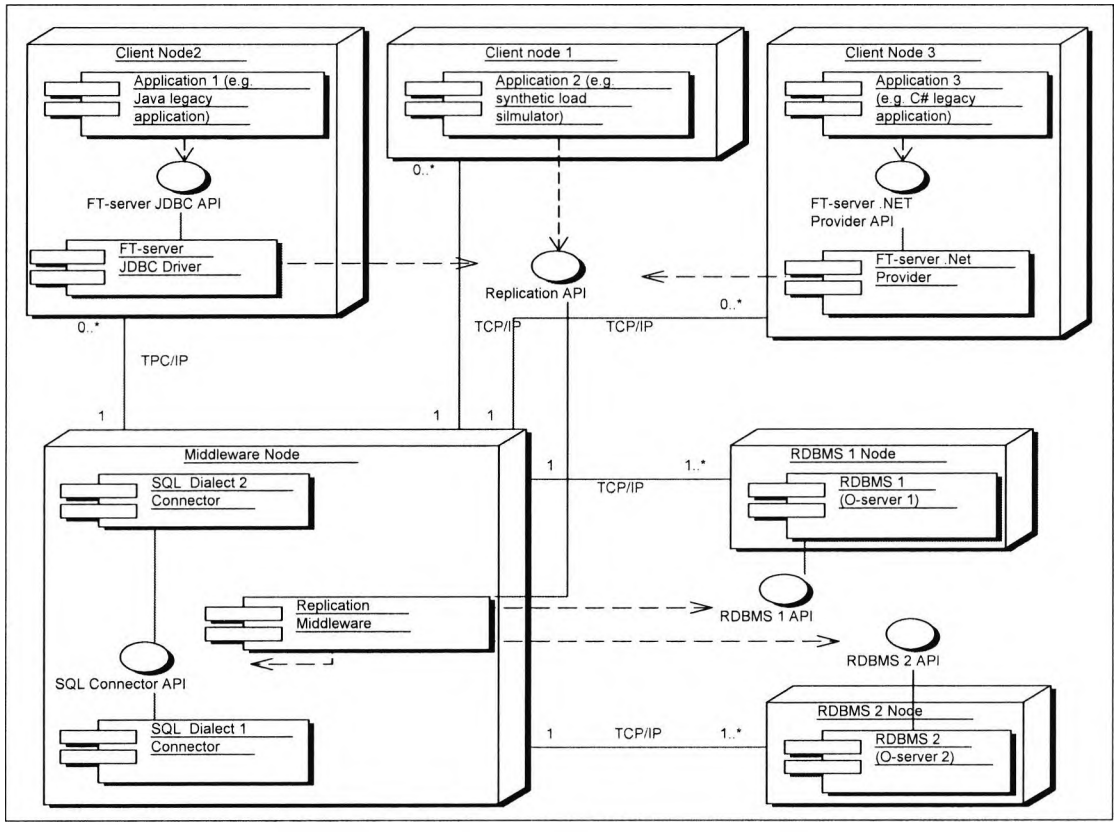

Fig. 10 - UML Deployment diagram of the FT-server

2.3 Failure detection, masking, recovery

The middleware of the FT-server includes extensive functionality for failure detection, masking and state recovery. 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[20] of the different O-servers. In a FT-node with 3 or more diverse O-servers, majority voting can be used to choose a result and thus mask the failure to the clients, and identify the failed O-server which may need a recovery action to correct its state. With a 2-diverse FT-node, if the two O-servers give different results, the middleware cannot decide which O-server is in error. This is where "data diversity" can help by providing additional results to break the tie (more in the next subsection). 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 transient failures (caused by so called "Heisenbugs" (Gray 1986)). To command backward error recovery, the middleware may use the standard database transaction mechanisms: aborting the failed transaction and replaying its statements may produce a correct execution. With "data diversity" a finer granularity level of recovery is possible using SAVEPOINTs and ROLLBACKs;
- additionally, diversity offers ways of recovering from non-transient failures (caused by so called "Bohrbugs" (Gray 1986)), by essentially copying the database state of a correct server into the failed one (similarly to (Tso and Avizienis 1987)). 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 (Sutter 2000). This would be expensive, perhaps to be completed off-line, but a designer can use

[20] An "output" may be the results from a SELECT statement or the number of rows affected for a write (INSERT, UPDATE and DELETE) statement. For INSERT and UPADTE statements a more refined way would be to read back the affected rows and use those for comparison.

multi-level recovery, in which the first step is to correct only those records that have been found erroneous on read statements.

2.4 Data diversity extensions

Even with just two diverse O-servers, many of the O-server failures may be masked by using "data diversity" (rephrasing an SQL statement into a different, but semantically equivalent one) to solicit "second opinions" from the O-servers and if possible outvote the incorrect response.

Data diversity could be implemented via an algorithm in the "Middleware Node" that rephrases statements according to predefined rules. We can define these rules for each type of SQL statement defined by the SQL grammar implemented by the server. These rules therefore may form part of the "SQL Dialect Connectors". Upon receiving a statement from a client application the middleware can look up a rule from the list of available rules and rephrase the statement. The middleware must allow for new rules to be defined as and when necessary. If the middleware exhausts the list of rules that it can apply to a certain statement but no "correct result"[21] can be established by applying the closed adjudication mechanism then an error message is returned to the client.

Data diversity can be used *with* or *without* design diversity. Architectural schemes using data diversity are similar to those using design diversity. For instance, Amman and Knight in (Ammann and Knight 1988) 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 statement is only rephrased if either the server "fail-stops" or its output fails an acceptance test. In "n-copy programming", a copy of the statement as issued by the client is sent to one of the O-servers and rephrased variant(s) are sent to the others; their results are voted to mask failures.

Data diversity allows for a finer-granularity of state recovery, which is facilitated by the implementation of SAVEPOINT and ROLLBACK within transactions. The procedure (written in pseudocode), for a statement within a transaction, is given at the end of this sub-section.

[21] Depending on the setup used a correct result could be either the majority result or one that passes an acceptance test.

A performance optimisation could be to perform adjudication at an intermediate step of the WHILE loop execution rather than at the end (e.g. for a "majority voting" adjudication, if there are five rules for a particular statement then could check after the execution of the first three rephrased versions of the statement whether results returned by each of them are identical; if yes then majority result is already obtained and therefore no need for the last two rephrased versions of the statement to be executed).

The SAVEPOINT and ROLLBACK approach is the correct way of ensuring the "isolation" property of an ACID transaction.[22] Otherwise, if we "ABORTed" the transaction and started a new one to perform the rephrased version of the statement, a concurrent transaction may have updated rows in the target table. This would lead to different results being returned by the O-server for the rephrased statement even though the behaviour is not faulty.

```
WHILE more rephrasing rules available for the statement DO
        IF WRITE (i.e. DML (INSERT, UPDATE or DELETE) or DDL (e.g. CREATE VIEW etc.)) statement THEN
                SAVEPOINT;
                Execute WRITE statement[s] produced by the current rephrasing rule;
                READ the rows amended by the WRITE statement;
                Store the results produced by the preceding READ statement;
                ROLLBACK TO last SAVEPOINT;
        ELSE IF READ (i.e. SELECT) statement THEN
                Execute READ statement[s] produced by the current rephrasing rule;
                Store the results produced by the READ statement;
        END IF
END WHILE
Adjudicate from the stored results produced by each rephrased version of the statement;
IF adjudication succeeds (e.g. "majority voting" produced a result) THEN
        Execute the statement which was adjudicated to be correct;
ELSE
        ABORT current Transaction
        Raise an exception;
END IF
```

3. SQL rephrasing rules

As explained in Section 2, the support for data diversity can be implemented in the middleware in the form of rephrasing rules. The initial step is defining the rules that are to be implemented. The rules can be defined by studying in depth the SQL language itself

[22] This is under the assumption that the ACID property of the transaction is failure-free.

to identify the parts of the language which are synonymous and therefore enable the definition of logically equivalent rephrasing rules. We took a different more direct approach to defining these rules: we studied the known bugs reported for 4 open-source servers, namely Interbase 6.0, Firebird 1.0[23], PostgreSQL 7.0 and PostgreSQL 7.2 (abbreviated IB 6.0, PG 7.0, FB 1.0 and PG 7.2 respectively). However our intention was not to simply define workaround rules which are highly bug specific, but instead to define generic rephrasing rules, which can be used in a broader setting. As a result we found that some of the generic rules that we defined could be applied to multiple bugs in our study. We provide examples next.

3.1 Generic rules

The "generic rules" are rephrasing rules, which can be applied to a range of 'similar' statements, be it DML (data manipulation language: SELECT, INSERT, UPDATE and DELETE) or DDL (data definition language e.g. CREATE TABLE etc.) statements. We have defined a total of 14 generic rephrasing rules. Full details of these rules are in (Gashi 2006) (***and also provided as Appendix A of this thesis***). We will provide details of Rule 8 and how it proved to be a useful *workaround* for two different bugs reported for two different servers.

Rule 8: *An SQL* VIEW *can be rephrased as an SQL* STORED PROCEDURE *or SQL* TEMPORARY TABLE

This rule proved to be a useful workaround for FB 1.0 Bug 488343 (Gashi 2005) (***and also provided as Appendix A of this thesis***). To observe the failure the bug report details the following setup:

```
CREATE TABLE CUSTOMERS (ID INT, NAME VARCHAR(10) );
CREATE TABLE INVOICES (ID INT, CUST_ID INT, CODE VARCHAR(10), QUANTITY INT);
INSERT INTO CUSTOMERS VALUES (1, 'ME');
INSERT INTO INVOICES VALUES (1, 1, 'INV.1', 5);
INSERT INTO INVOICES VALUES (2, 1, 'INV.2', 10);
INSERT INTO INVOICES VALUES (3, 1, 'INV.3', 15);
INSERT INTO INVOICES VALUES (4, 1, 'INV.4', 20);
```

The following VIEW is faulty (specifically, the use of the SQL DISTINCT keyword to filter the results of a SELECT statement is faulty in SQL VIEWs of the FB 1.0 server):

[23] Firebird is the open-source descendant of Interbase 6.0. The later releases of Interbase are issued as closed-development by Borland.

```
CREATE VIEW V_CUSTOMERS AS SELECT DISTINCT ID, NAME FROM CUSTOMERS;
```

The failure can be observed by issuing the following statement:

```
SELECT SUM(INV.QUANTITY) FROM INVOICES INV INNER JOIN
        V_CUSTOMERS CUST ON INV.CUST_ID = CUST.ID;
SUM
20
```

The expected result is 50 not 20. If we use a STORED PROCEDURE instead of the VIEW then the correct results is returned[24]:

```
SET TERM !!;
CREATE PROCEDURE V_CUSTOMERS RETURNS  (ID INT, NAME VARCHAR(10)) AS
        BEGIN
                FOR SELECT DISTINCT ID, NAME FROM CUSTOMERS INTO :ID, :NAME DO
                BEGIN
                        SUSPEND;
                END
        END!!
SET TERM; !!
```

Issuing the same SELECT statement as before we obtain the expected result (50):

```
SELECT SUM(INV.QUANTITY) FROM INVOICES INV INNER JOIN V_CUSTOMERS CUST ON
        INV.CUST_ID = CUST.ID;
SUM
50
```

The same rule was a useful workaround for another bug, this time the PG 7.0 bug 23 (Gashi 2003). To observe the failure the bug report details the following setup:

```
CREATE TABLE L (PID INT NOT NULL, SEARCH BOOL, SERVICE BOOL);
INSERT INTO L VALUES (1,'T','F'); INSERT INTO L VALUES (1,'T','F');
INSERT INTO L VALUES (1,'T','F'); INSERT INTO L VALUES (1,'T','F');
INSERT INTO L VALUES (1,'T','F'); INSERT INTO L VALUES (1,'F','F');
INSERT INTO L VALUES (1,'F','F'); INSERT INTO L VALUES (2,'F','F');
INSERT INTO L VALUES (3,'F','F'); INSERT INTO L VALUES (3,'T','F');
```

The following VIEWs are then defined (notice the use of the GROUP BY clause):

```
CREATE VIEW CURRENT AS SELECT PID, COUNT(PID), SEARCH, SERVICE FROM L GROUP BY PID,
SEARCH, SERVICE;
CREATE VIEW CURRENT2 AS SELECT PID, COUNT (PID), SEARCH, SERVICE FROM L GROUP BY PID,
SEARCH, SERVICE;
```

[24] The syntax used is specific for Firebird.

By issuing the following SELECT statement incorrect results are obtained (this is due to the GROUP BY clause used in the VIEWs and the COUNT used on a column from a VIEW):

```
SELECT CURRENT.PID, CURRENT.COUNT AS SEARCHTRUE, CURRENT2.COUNT AS
SEARCHFALSE FROM CURRENT,CURRENT2 WHERE CURRENT.PID =CURRENT2.PID AND
CURRENT.SEARCH='T' AND CURRENT2.SEARCH='F' AND CURRENT.SERVICE='F' AND
CURRENT2.SERVICE='F';
-- pid | searchtrue | searchfalse
--   1 |       10   |       10
--   3 |       1    |       1
```

The expected results are:

```
-- pid | searchtrue | searchfalse
--   1 |       5    |       2
--   3 |       1    |       1
```

By using TEMPORARY TABLEs instead of VIEWs the correct result is obtained:

```
SELECT PID, COUNT(PID), SEARCH, SERVICE INTO TEMP
        CURRENT FROM L GROUP BY PID, SEARCH, SERVICE;
SELECT PID, COUNT(PID), SEARCH, SERVICE INTO TEMP
CURRENT2 FROM L GROUP BY PID, SEARCH, SERVICE;
SELECT CURRENT.PID,CURRENT.COUNT AS SEARCHTRUE, CURRENT2.COUNT AS SEARCHFALSE
FROMCURRENT, CURRENT2 WHERE CURRENT.PID=CURRENT2.PID AND CURRENT.SEARCH='T' AND
CURRENT2.SEARCH='F' AND CURRENT.SERVICE='F' AND CURRENT2.SERVICE='F';
 -- pid | searchtrue | searchfalse
--   1 |       5    |       2
--   3 |       1    |       1
```

We used TEMPORARY TABLEs in PG 7.0 and not STORED PROCEDUREs since PG 7.0 does not support functions (procedures) that return multiple rows.

Details of the other generic rephrasing rules and how they can be used as workarounds for other reported bugs are given in (Gashi 2006).

We looked at how many of the generic rules can be applied to the bugs reported for the open-source servers in our bugs study. The results are shown in Table 14. The leftmost three columns of the table show the results for the non-self-evident failures caused by read (i.e. SELECT) statements. Clearly, a number of these are also classified as a "user error", i.e. the user issues an incorrect statement, which the server incorrectly executes without raising an exception. For example IB 6.0 incorrectly executes a statement such as

SELECT X FROM A, B even though the column X is defined in both tables A and B, which can lead to ambiguous results. PG 7.0 / PG 7.2, correctly, raise an exception.

If we take away the "user error" bugs then we can see that in all the server pairs the generic rules can be used as workarounds for at least 80% of the non-self-evident failures caused by read statements.

Table 14 - A summary of applying the generic rephrasing rules for non-self evident and state-changing bugs of IB 6.0 and PG 7.0 and the later releases FB 1.0 and PG 7.2

Server pair	Non-self evident non-state-changing failures (SELECT statements)			State-changing failures			
				DDL statement failures		Write statement failures	
	Total	Total covered by generic rules	Total user errors *	Total	Total covered by generic rules	Total	Total covered by generic rules
IB 6.0 + PG 7.0	21	12	6	21	13	9	7
IB 6.0 + PG 7.2	26	18	6	19	13	7	5
FB 1.0 + PG 7.0	16	11	2	19	13	8	6
FB 1.0 + PG 7.2	19	15	2	17	13	6	4

The right-most 4 columns of the table are for the bugs that cause state-changing failures, which have been further subdivided into bugs in DDL and write statements. We can see that generic rules can be used as workarounds for at least 60% of failures caused by the state-changing statements.

3.2 Specific rules

The generic rephrasing rules that we have defined do not provide workarounds for all the failures caused by the bugs collected in our study. For these failures specific workaround rules need to be defined. For example recursive BEFORE UPDATE TRIGGERs can return error messages in FB 1.0/IB 6.0 which means the table for which the trigger is defined becomes unusable (FB 1.0 bug 625899 (Gashi 2005)). A generic rule could not be defined for this bug. A specific workaround (and a generic recovery procedure) upon encountering this error message would be to:

- disable the trigger in FB 1.0 / IB 6.0
- read the log of the other server to check the sequence of the write statements that have been issued as a result of the trigger
- send this sequence of statements explicitly to the FB 1.0 / IB 6.0 server

The workaround above would work in a diverse server-type configuration if the other server[s] works correctly (the other server[s] in our study do not contain this bug) while without design diversity a fault, clearly, cannot be dealt with this way.

We have found that a large number of bugs, if server diversity is not employed, would require very specific rules to be defined to workaround the failures that they cause. In many cases these rules require substantial new implementation in the form of "wrapping" of the results returned to the client (or for write statements before they are stored in the database) or re-implementing parts of the functionality of the database that are found to be faulty and no workaround exists in SQL. Although possible such an approach is clearly limited because the newly developed code can itself be faulty which may diminish the gains in reliability that can be obtained from its use. This reiterates that design diversity is desirable.

4. Performance implications of rephrasing

To measure the performance implications of rephrasing, we conducted a number of experiments based on the industry standard benchmark for databases - TPC-C (TPC 2002)[25]. The factors which degrade performance when rephrasing is employed are:

1. delays enforced by the middleware for comparison of results
2. delays from using the following mechanisms within transactions:
   - Transaction SAVEPOINTs
   - Transaction ROLLBACKs
   - Execution of SELECT statements after WRITE statements (INSERT, UPDATE, DELETE)
   - Rephrasing

The additional delay introduced by the use of rephrasing is delay 2. We have performed an experimental study to estimate delay 2. Delay 1 would exist also in a diverse setup with or without rephrasing. Studies that have reported measures of other delays which are not specific to rephrasing (such as enforcing 1-copy serialisability) can be found in (Lin, Kemme et al. 2005), (Patino-Martinez, Jiménez-Peris et al. 2005)[26]. There are other

[25] The TPC-C experiments were carried out with 1 emulated client and 1 warehouse with client think times set to 0.

[26] These studies also provide some optimisation procedures for 1-copy serialisability.

factors that can influence the degradation of performance that we have not measured in our experimental setup (e.g. rephrasing delays when more than one rephrasing rule is used etc.). The experiments that we have conducted aim to provide an initial estimate of the delays due to rephrasing. A more thorough performance evaluation should also take into account concurrent execution of transactions. As was also noted by one of the anonymous reviewers, for some concurrency control mechanisms, the increase in transaction execution times due to the use of rephrasing, the probability of conflicts due to concurrency may also increase which may further degrade performance.

The experimental setup consisted of three computers. All three computers ran on Microsoft's Windows 2000 operating system, they had 384 MB RAM, and Intel Pentium 4 1.5GHz processors. One machine hosted the client implementation of the TPC-C benchmark. The other two machines hosted the servers (PostgreSQL 8.0 and Firebird 1.5). We used later releases of the servers than the ones used in our bugs study since these earlier releases do not support SAVEPOINTs and ROLLBACKs within transactions. We have not used any commercial servers in our experiments since the license agreements are very restrictive with regard to publishing performance data.

We ran experiments on both diverse and non-diverse setups. In the diverse experiments we always wait for the slowest server response before we can start the next transaction. Therefore the diverse setups here are always slower (other configurations are possible and we have discussed some of these in (Gashi, Popov et al. 2004a)).

Fig. 11 illustrates the sequence of executions within a transaction for the different non-diverse setups. The grey boxes represent the fault tolerance mechanism used whereas the dotted lines represent the added delay from the use of the respective mechanism. Setup a) is the baseline, against which we will measure the added delays. Setups b), c), and d) measure the delays of using the fault tolerance mechanisms when no failures are observed (i.e. the cost of being cautious)[27]. Setups e) and f), measure the cost of re-execution of a statement[28]. These experiments measure delays for a number of situations:

---

[27] b) detection of erroneous writes; c) SAVEPOINT are used before write statements for finer grained recovery; d) both SAVEPOINTs are used and the modified rows are read back (combination of b) and c));

[28] e) optimistic (on writes) rephrasing: each statement is executed twice; to ensure that the state of the database remains unchanged during the second execution of the write statement we use SAVEPOINTs and ROLLBACKs; f) pessimistic rephrasing: same as e) but the written rows are also read to protect against erroneous writes.

- re-execution of an unchanged statement as a possible protection against transient failures (caused by the so called "Heisenbugs" (Gray 1986))
- re-execution of a logically equivalent rephrased statement in case the first one has failed self-evidently (i.e. a crash or other exceptional failures)
- re-execution of a logically equivalent rephrased statement to get additional results for comparison on the middleware to increase the likelihood of failure detection for non-self-evident failures

In our experiments we did not use rephrased statements. Instead, the same statement was executed twice. This is a simplification due to the absence of a proper implementation of rephrasing. In the absence of any other data, we wanted to get an initial estimate of the delays that the various fault tolerance mechanisms will produce with the database servers. The diverse setups have a similar structure. The only difference is that in diverse setups we only use 1 SAVEPOINT (at the beginning of the transaction) rather than before each write statement and therefore we may also have only one ROLLBACK (at the end of transaction). For setups e) and f), this means that we first execute every statement once then we ROLLBACK to the beginning and execute all the statement again. So the difference between the diverse and non-diverse setups is a different level of granularity of using SAVEPOINTs/ROLLBACKs.

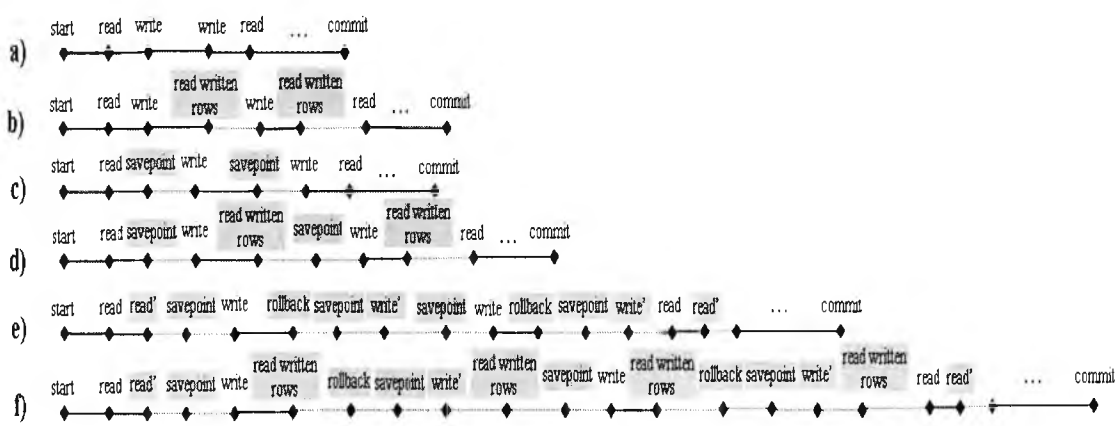

Fig. 11 - A transaction execution sequence in the experimental setups. The shaded boxes represent the fault tolerant mechanism used and the dotted lines represent the additional delays from their use. The second executions of the statements are proxies for rephrased versions of statements.

The full results of these experiments are given in Table 15. The first column explains the setup under which the experiment was run. The following 4 columns spell out which fault tolerance mechanisms were used (if the cell is blank then the respective mechanism was not used). The following 3 columns show the average execution time of a transaction, and the last 3 columns show the added delay (in percentages) proportional to the baseline of each setup. The first six rows contain the results for each of the setups we explained earlier (and illustrated in Fig. 11).

The last two rows are structurally the same as setups (e) and (f) respectively. However in these experiments we have tried to simulated the effect of a simple learning rule: if after 1000 executions a statement has been found to be correct then we stop rephrasing (in our simulation it means we stop executing the statement twice for both setups and additionally stop executing the SELECT statement that read the modifications of the write statements for setup (h)).

Table 15 - Performance effects of the various fault-tolerance schemes. Each experiment is run with loads of 10,000 transactions

Setup					Average Transaction Execution time (milliseconds)			Delays proportional to the baseline (%)		
Setup description (with reference to Fig. 11)	SAVEPOINTs	ROLLBACKs	2 executions for each statement	SELECT after Write statements	PG 8.0	FB 1.5	Diverse PG 8.0 & FB 1.5	PG 8.0	FB 1.5	Diverse PG 8.0 & FB 1.5
Baseline (a)					228	306	343			
Detection of erroneous writes (b)				√	292	356	434	28.3	16.3	26.5
Finer granularity of recovery (c)	√				240	308	350	5.3	0.4	1.8
Combination of b and c (d)	√			√	305	364	433	33.9	18.6	26.0
Optimistic (on writes) Rephrasing (e)	√	√	√		353	450	489	54.9	46.9	42.3
Pessimistic Rephrasing (f)	√	√	√	√	496	601	699	118.	96.2	105.5
Learning Optimisation (g)	√	√	√		256	325	402	12.6	6.2	17.3
Learning Optimisation (h)	√	√	√	√	278	341	524	22.5	11.4	52.6

The delays seem to be higher proportionally in PostgreSQL than in Firebird. This is because the execution time of COMMITs is smaller in Firebird for the experiments with larger number of SELECT statements. The number of write statements to be COMMITed always remains the same in all experiments (even in the ones with 2 executions of statements, since the first execution of a write statement is always ROLLBACKed). Comparing the setups a) with e) we can see that even though in setup e) every statement

is being executed twice the average execution times of the transactions are not simply twice the execution time of transactions in setup a). This is explained by the fact that the number of transactions remains the same (i.e. we still have the same number of COMMITs) and also the data may be stored already in the RAM which reduces the execution time of the second statement. The same holds when comparing results of setups b) with f).

Since the numbers in Table 15 represent point estimates (i.e. they are single runs of an experiment per setup) we have repeated the experiments for setup a) and f) to measure the non-deterministic variation that may exist between the different runs. We observed a very small difference (less than 1% for 5 out of six of the experiments and less than 3% for all). Hence we can trust with a higher degree of confidence that the observations documented in Table 15 represent closely the 'true' differences between different setups.

5. Discussion

We presented in Section 2 the architecture we propose for a fault-tolerant server employing rephrasing. The middleware used would make use of a rephrasing algorithm. Any fault-tolerant solution, which makes use of server diversity would need to have "connectors" developed as part of the middleware to translate a client sent statement to the dialect of the respective server. This is because each server 'speaks' its own dialect of SQL. The rephrasing algorithms can also be part of these connectors. A related point is that database servers offer features that are extensions to the SQL standard, and these features may differ between the servers. Therefore for applications which require a richer set of functionality data diversity would be attractive alone as it would for instance allow applications to use the full set of features. A complex statement, which can be directly executed with some servers but not others, may need to be rephrased as a logically equivalent sequence of simpler statements for the latter. For example, the TRUNCATE command is a PostgreSQL specific feature (and is buggy in version 7.0; see bug 20 (Gashi 2003) for details). In its stead the DELETE command can be used to workaround the problem. The DELETE command is also implemented in Firebird and all the other SQL compliant servers.

Since most of these rules are transformations of the SQL grammar, they are amenable to formal analysis. Thus, despite the additional implementation, high reliability can be achieved with a combination of formal analysis and testing of the new code.

The results presented in Section 3 demonstrate that a small number of rephrasing rules can help with server diagnosis and state recovery. We observed that a limited set of generic rephrasing rules that we have defined (14 in total) can be used as workarounds for at least 80% of the non-self-evident failures caused by read statements and at least 60 % of failures caused by write or DDL statements in any of the open-source 2-diverse setups in our study. We have also observed that using data diversity without design diversity would lead to a large number of *specific rephrasing rules* to workaround certain failures. Implementing such rules might require a substantial amount of new implementation, which itself may be faulty, thus, reducing the possible reliability gains that can be obtained from their use.

Rephrasing has been proposed as a possibility to detect failures that would otherwise be un-detectable in some replication settings. The possible benefits of this approach could be its relatively low cost in comparison with design diversity, and also that it can be used with or without design diversity allowing for various cost-dependability trade-offs. Possible setups include:

- In non-diverse redundant replication settings, if high dependability assurances are required, the only option available would be to rephrase all the statements sent to the server. This can lead to high performance penalties. To reduce the performance penalty some form of learning strategy can be applied, e.g. keep track of all the statements that have been rephrased. If the rephrased statement keeps giving the same results as the original statement then confidence is gained that the original statement is giving the correct result and the statement does not have to be rephrased in future occurrences (what we did in setups g) and h) of the TPC-C experiments). The other dimension is to stop sending the client-version of the statement to a server if it always gives an incorrect result. In this case the middleware can flag each occurrence of this statement and use the rephrased version of it without sending the original statement to the server (Popov, Strigini et al. 2004). This reduces the time taken to respond to the client.

- In a diverse server configuration a less rephrasing-intensive approach may be used where only the read statements (i.e. SELECTs) that return different results are rephrased (assuming that at least two servers are running in parallel so that a mismatch is detected). The rephrasing is also done for all the write statements (to ensure that the state of the database is not corrupted). Since a smaller set of statements needs to be rephrased the performance is enhanced. The non-self-evident identical failures, however, (we observed 4 of these in the study with known bugs of SQL servers (Gashi, Popov et al. 2004b)) will not be detected. To further enhance the performance the same learning strategies can be used as in the previous setup.

6. Conclusions

We have reported previously (Gashi, Popov et al. 2004b) on the dependability gains that can potentially be achieved from deploying a fault-tolerant SQL server, which makes use of diverse off-the-shelf SQL servers. From studying bugs reported for four off-the-shelf servers we reported that failure detection rates in 1-out-of-2 configurations was at least 94% and this increased to 100% in configurations which employed more than two servers. However fault tolerance is more than just failure detection. In this paper we reported on the mechanism of data diversity and its application with SQL servers in aiding with failure diagnosis and state recovery. We have defined 14 generic 'workaround rules' to be implemented in a 'rephrasing' algorithm which when applied to a certain SQL statement will generate logically equivalent statements. We have also argued that since these rules are transformations of the SQL language syntax, they are amenable to formal analysis and dependability gains from employing rephrasing are achievable despite the development of a bespoke new code.

We also outlined a possible architecture of a fault tolerant server employing diverse SQL servers and detailed how the middleware used in it can be extended to also handle rephrasing of SQL statements.

We also presented some performance measurements from experiments we have run with an implementation of the TPC-C benchmark (TPC 2002), which gave initial estimates of the likely delays due to employing rephrasing.

Further work that is desirable includes:

- demonstrating the feasibility of automatic translation of SQL statements from, say ANSI/ISO SQL syntax to the SQL dialect implemented by the deployed SQL servers. We have completed some preliminary work on implementing translators between MSSQL and Oracle dialects for SELECTs, and between Oracle and PostgreSQL dialects for SELECT, INSERT and DELETE statements;
- developing the necessary components so that users can try out diversity in their own installations, since the main obstacle now is the lack of popular off-the-shelf "middleware" packages for data replication with diverse SQL servers. This would also include implementing a mechanism of maintaining (adding/removing) rephrasing rules as add-on components in the middleware.

Acknowledgment

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). Authors would like to acknowledge the anonymous reviewers for the thoughtful comments and useful suggestions.

References

Ammann, P. E. and J. C. Knight (1988), "*Data Diversity: An Approach to Software Fault Tolerance*", IEEE Transactions on Computers 37(4), pp: 418-425.

Bernstein, P. A., V. Hadzilacos and N. Goodman (1987), "*Concurrency Control and Recovery in Database Systems*", Reading, Mass., Addison-Wesley.

EnterpriseDB (2006), "*EnterpriseDB*", http://www.enterprisedb.com/.

Gashi, I. (2003), "*Tables containing known bug scripts of Interbase, PostgreSQL, Oracle and MSSQL.*" http://www.csr.city.ac.uk/people/ilir.gashi/DBMSBugReports/.

Gashi, I. (2005), "*Tables containing known bug scripts of Firebird 1.0 and PostgreSQL 7.2*", http://www.csr.city.ac.uk/people/ilir.gashi/DBMSBugReports/.

Gashi, I. (2006), "*Rephrasing Rules for SQL servers*", http://www.csr.city.ac.uk/people/ilir.gashi/DBMSBugReports/.

Gashi, I., P. Popov, V. Stankovic and L. Strigini (2004a), "*On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*", *in* Architecting Dependable Systems II, R. de Lemos, Gacek, C., Romanovsky, A. (Eds.), Springer-Verlag, 3069, pp: 191-214.

Gashi, I., P. Popov and L. Strigini (2004b), "*Fault Diversity Among Off-The-Shelf SQL Database Servers*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '04), Florence, Italy, IEEE Computer Society Press, pp: 389-398.

Gray, J. (1986), "*Why Do Computers Stop and What Can be Done About it?*" *in proc.* Int. Symp. on Reliability in Distributed Software and Database Systems (SRDSDS '86), Los Angeles, CA, USA, IEEE Computer Society Press, pp: 3-12.

Janus-Software (2006), "*Fyracle*", http://www.janus-software.com/fb_fyracle.html.

Jimenez-Peris, R., M. Patino-Martinez, G. Alonso and B. Kemme (2003), "*Are Quorums an Alternative for Data Replication?*" ACM Transactions on Database Systems 28(3), pp: 257-294.

Lin, Y., B. Kemme, M. Patino-Martínez and R. Jiménez-Peris (2005), "*Middleware based Data Replication providing Snapshot Isolation*", *in proc.* Int. Conf. on Management of Data (SIGMOD'05), Baltimore, Maryland, USA, ACM Press, pp: 419-430.

Patino-Martinez, M., R. Jiménez-Peris, B. Kemme and G. Alonso (2005), "*MIDDLE-R: Consistent Database Replication at the Middleware Level*", ACM Transactions on Computer Systems 23(4), pp: 375-423.

Popov, P., L. Strigini, A. Kostov, V. Mollov and D. Selensky (2004), "*Software Fault-Tolerance with Off-the-Shelf SQL Servers*", *in proc.* Int. Conf. on COTS-based Software Systems (ICCBSS '04), Redondo Beach, CA USA, Springer, pp: 117-126.

Popov, P., L. Strigini, S. Riddle and A. Romanovsky (2001), "*Protective Wrapping of OTS Components*", *in proc.* 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction, Toronto.

Sutter, H. (2000), "*SQL/Replication Scope and Requirements Document*", ISO/IEC JTC 1/SC 32 Data Management and Interchange WG3 Database Languages, H2-2000-568.

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

Tso, K. S. and A. Avizienis (1987), "*Community Error Recovery in N-Version Software: A Design Study with Experimentation*", *in proc.* Int. Symp. on Fault-Tolerant Computing (FTCS '87), Pittsburgh, PA, USA, pp: 127-133.

VI. Optimal Selection of COTS Components

Paper-5. Uncertainty Explicit Assessment of Off-the-Shelf Software: Selection of an Optimal Diverse Pair

Abstract: *Assessment of software COTS components is an essential part of component-based software development. Sub-optimal selection of components may lead to solutions with low quality. The assessment is based on incomplete knowledge about the COTS components themselves and other aspects, which may affect the choice such as the vendor's credentials, etc. We argue in favour of assessment methods in which uncertainty is explicitly represented ('uncertainty explicit' methods) using probability distributions. We have adapted a model (developed elsewhere (Littlewood, Popov et al. 2000)) for assessment of a pair of COTS components to take account of the fault (bug) logs that might be available for the COTS components being assessed. We also provide empirical data from a study we have conducted with off-the-shelf database servers, which illustrate the use of the method.*

Co-author: Dr. Peter Popov

Conference: IEEE International Conference on COTS-Based Software Systems 2007 (ICCBSS-07)

Date of submission: July-2006

Status: *Published*

Number of reviewers: 3

Publication date: March 2007

Full citation: Gashi, I., Popov, P., "*Uncertainty Explicit Assessment of Off-the-Shelf Software: Selection of an Optimal Diverse Pair*", in Proc. *ICCBSS-2007, Sixth International Conference on COTS Based Software Systems*, Banff, Alberta, Canada, pp. 93-102, IEEE Computer Society Press, 2007

1. Introduction

Commercial-off-the-shelf (COTS) components often form an essential part in software development. Benefits of their use are wide ranging: from the incentive to cut-down on cost to reducing the development time and improving quality by using tried and tested components. An initial and essential part of component based software development is the assessment of available COTS components. There exist a plethora of available methods for COTS assessment (Ncube and Maiden 1999), (Kontio, Chen et al. 1995), (Jeanrenaud and Romanazzi 1994), (Tran and Liu 1997), (Ochs, Pfahl et al. 2001), (Alves and Castro 2001), (Phillips and Polen 2002), (Boehm, Port et al. 2003), (Dean 2000), (Kunda and Brooks 1999), (Gregor, Hutson et al. 2002), (Burgués, Estay et al. 2002), (Comella-Dorda, Dean et al. 2002), (Ruhe 2003). An often overlooked aspect in the existing assessment techniques is the inherent *uncertainty* in the values of the parameters being assessed. This is because the assessment is carried out with limited resources of time and budget. Therefore the true values of the assessed attributes will rarely be known with certainty.

For solutions with very stringent dependability requirements a single component may rarely be able to meet the required dependability target. It has been argued (Littlewood and Strigini 1993) that employing fault-tolerance in the form of software design diversity (i.e. using more than one component to perform the same function) is usually the best guarantee of achieving higher levels of dependability than what the available COTS components can offer. But, employing software diversity was seen in the past as an expensive method for increasing dependability due to the need of building more than one component. With off-the-shelf components this problem is overcome: there may be many different components that will have the required functionality therefore bespoke development may not be required[29]. Moreover many of these components are free and open-source, thus the cost of procurement may be non-existent. The problem of assessment though still exists. If we were interested in building a 1-out-of-2[30] system, simply choosing the two best components that exist in the market may not be enough.

[29] Apart from *'glue code'* (usually referred to as *middleware)* which may be needed to ensure the components can be deployed for a given system in a coordinated manner as required by the particular system context.

[30] In this configuration the system performs correctly as long as 1 of the 2 components works correctly.

What is of interest is how well the pair works together. The optimal pair will be the one with the lowest probability of coincident failures of both components of the pair. The components that form the best pair may not necessarily be the ones which are the best individually. For further details on the subtleties of this problem the interested reader is referred to a recent survey (Littlewood, Popov et al. 2001).

In this paper we will provide details of an adaptation of the model in (Littlewood, Popov et al. 2000) which allows for an optimal selection of a pair of components to be used in a fault-tolerant system. In this model the assessment results are subject to uncertainty and we discuss how this may impact the decisions about which pair of components we choose. The model also enables representing the dependencies that exist between uncertainties associated with the values of each COTS component in the pair.

The paper is structured as follows: Section 2 contains a brief review of related work on COTS assessment; in Section 3 we describe the model of assessment, in which model parameters are not known with certainty and argue in favour of using probability distributions as an adequate mechanism to capture this uncertainty; in Section 4 we provide details of an empirical study with off-the-shelf database servers and illustrate how our approach can be used to select the optimal diverse pair; in Section 5 we provide a discussion of the method and finally Section 6 contains conclusions and provisions for further work.

2. Related work

There are a wide variety of COTS assessment approaches available. All of them start with an initial statement of requirements, which defines what is being sought. It has been proposed that the requirements initially should not be too stringent, since this would discard potentially appropriate COTS candidates at a very early stage (Dean 2000), (Lewis, Hyle et al. 2000). It has even been suggested (Lewis, Hyle et al. 2000) that if the requirements are not flexible then the COTS based development may not be appropriate and bespoke development could be more cost-effective. So initially (Lewis, Hyle et al. 2000) suggests distinguishing between essential requirement and those that are negotiable. The selection criteria are then based on the essential requirements.

Off-the-shelf-option (OTSO) (Kontio, Chen et al. 1995) is a multi-phase approach to COTS selection. The phases are: the search phase, the screening and evaluation phase and the analysis phase. In the first phase COTS products are identified. In the screening and evaluation phase the products are further filtered using a set of evaluation criteria (established from a number of sources, including the requirements specification, the high level design specification etc). In the analysis phase results of the evaluation are analysed, which lead to the final selection of COTS products for inclusion in the system.

Procurement-oriented requirements engineering (PORE) (Ncube and Maiden 1999) is a process in which requirements are defined in parallel with COTS component evaluation and selection. (Ncube and Maiden 1999) propose using prototypes to develop knowledge concerning COTS products and their use within the wider system.

Other assessment methods include: CISD (COTS-based Integrated System Development) (Tran and Liu 1997), STACE (Socio Technical Approach to COTS Evaluation) (Kunda and Brooks 1999), CDSEM (Checklist Driven Software Evaluation Methodology) (Jeanrenaud and Romanazzi 1994), CRE-COTS-Based Requirements Engineering Method (Alves and Castro 2001), CEP (Comparative Evaluation Process Activities) (Phillips and Polen 2002), CBA Process Decision Framework (Boehm, Port et al. 2003), A Proactive Evaluation Technique (Dean and Vidger 2000), CAP-COTS Acquisition Process method (Ochs, Pfahl et al. 2001), Storyboard Process (Gregor, Hutson et al. 2002), Combined Selection of COTS Components (Burgués, Estay et al. 2002), PECA Process (Comella-Dorda, Dean et al. 2002) or COTS-DSS (Ruhe 2003).

3. Assessment of diverse COTS solutions: Bayesian approach

3.1 Uncertainty in the assessment

Any assessment is conducted with limited resources and under various assumptions, which may not hold true in real operation. Therefore the outcome of the assessment is subject to uncertainty. For example, deciding to rate a COTS component exactly 7 out of 10 according to a chosen scale may be difficult to justify. The assessor may be certain that the values of the attribute outside the range {6,7} are unreasonable but be *indifferent*

between the possible values inside this interval. Software reliability is a typical example of an attribute which is never known with certainty. Probability of failure on a randomly chosen demand (*pfd*) is unknown, but the assessor may be prepared to state, with confidence 99%, that it is less than, say 10^{-3}. The assessor may be even more specific of their doubts about the COTS *pfd* and state that the most likely range of the *pfd* is between 10^{-4} and 10^{-3}.

There are various methods for representing uncertainty (Wright and Cai 1994). Bayesian approach to probabilistic modelling is one of the best-known ones and used with some success in reliability assessment (Littlewood and Wright 1997). It allows one to combine, in a mathematically sound way, the prior belief (which is 'subjective' and possibly inaccurate) about the values of a parameter with the ('objective') evidence from seeing the modelled artefact (in this case a COTS component) in operation. Combining the prior belief and the evidence from the observations in a mathematically correct way leads to a posterior belief about the values of the assessed attribute. If the prior belief is represented as a probability *distribution* rather than a single value, then after seeing the observations we get a posterior distribution (quantification of uncertainty) which takes into account both the prior knowledge and the empirical evidence.

3.2 Model for assessment of 1 COTS component with one attribute

In this section we illustrate how the Bayesian approach to assessment is normally applied to assessing a single attribute of a single COTS component. Assume that the attribute of interest is the component's probability of failure on demand (*pfd*).

If the system is treated as a black box, i.e. we can only distinguish between *COTS component's* failures or successes (Fig.12), the Bayesian assessment proceeds as follows.

Fig. 12 - Black-box model of a COTS component

Let us denote the system *pfd* as p, with prior distribution $f_p(\bullet)$, which characterizes the assessor's knowledge about the system *pfd prior* to observing the COTS component in operation. Assume further that the COTS component is subjected to n demands,

independently drawn from a 'realistic' operational environment (profile), and r failures are observed. The posterior distribution, $f_p(x \mid r,n)$, of p after the observations will be:

$$f_p(x \mid r,n) \propto L(n,r \mid x) f_p(x) \quad (1)$$

where $L(n,r \mid x)$ is the *likelihood* of observing r failures in n demands *if* the *pfd* were exactly x, which in this case of independent demands is given by the *binomial* distribution:

$$L(n,r \mid x) = \binom{n}{r} x^r (1-x)^{n-r} \quad (2)$$

For any prior and any observation (r, n), including (r=0), the posterior can be calculated. Thus it can be applied to all the COTS components included in the assessment. Now, the selection can be based on the posterior distribution derived for the COTS components using different criteria:

- for a given reliability target the COTS component chosen will be the one which has the highest probability of having a *pfd* lower than the given target;
- for a predefined 'mission' of say, 1000 demands, the COTS component chosen will be the one which is most likely to survive the mission without a failure.

3.3 Model for assessment of a fault-tolerant system consisting of 2 COTS components

The Bayesian assessment can also be applied to choosing a pair of components. In what follows we will describe how the assessment can be performed for a system made up of two components. The mathematical details can be found in (Littlewood, Popov et al. 2000) and ***Appendix VI-1A at the end of this paper***.

Let us assume that the attribute of interest is again the *pfd* of the system: that is of simultaneous failure of both components. Now assume that the system is subjected to a series of independently selected demands. On each demand the response received from each of the COTS components is characterized as correct/incorrect. Since we have two COTS components clearly 4 combinations exist, which can be observed on a demand, as shown in Table 16.

The four probabilities given in the last column of Table 16 sum to unity (i.e. they sum to 1). So if the last three probabilities are 0.2, 0.4 and 0.3 respectively then the first one p_{10} = 1- (0.2 + 0.4 + 0.3) = 0.1. Thus, the joint distribution of any three of these probabilities, will give an exhaustive description of the COTS pair behaviour. In statistical terms, the model of the COTS component pair has three degrees of freedom. Since we have a three variate distribution we need to define three prior distributions (not a single one as in the previous section): the prior distributions for the *pfd* of each of the components, and then the conditional prior distribution for the *pfd* of both components simultaneously. The details of this joint distribution are given in (Littlewood, Popov et al. 2000) and ***Appendix VI-1A at the end of this paper***. From this distribution we can then derive the marginal distribution of common failures which will be used to choose the best pair of components in a 1-out-of-2 setup.

Table 16 - The outcomes and their frequency and probabilities for each demand

Event	COTS A Correct	COTS B Correct	Observations in n demands	Probability
A	No	Yes	r_1	p_{10}
B	Yes	No	r_2	p_{01}
Γ	No	No	r_3	p_{11}
Δ	Yes	Yes	r_4	p_{00}

3.4 Utilizing multiple sources of data in the assessment

In some areas of software engineering, especially in testing, the usefulness of *partitioning the demand space* has been recognized (Jeng and Weyuker 1991), (Hamlet and Taylor 1990), (Musa 1993). The demand space partitions typically represent different *types* of demands, which may have different likelihoods of occurring in realistic environment. Realistic testing, thus, would require generating mixes of demands, which take into account the likelihood of the types of demands.

In our context, operating in a partitioned demand space may imply that the uncertainty associated with the attribute of interest may differ among the partitions, e.g. as a result of different number of observations being made for the different partitions.

If the demand space is partitioned into M partitions $\{S_1, S_2, \ldots S_M\}$, then for each of these the assessment will be performed as described above, e.g. with two COTS components the description provided in Section 3.3 will apply. As a result M conditional distributions will be associated with each pair of COTS components from which the conditional

uncertainty of interest will be expressed, that characterizes the behaviour of the particular pair of COTS components in the specific partition. Finally, in order to compare the competing pairs of COTS components the unconditional distribution of the probability of joint failure should be derived for the particular profile defined *over the set of partitions*, which represents the targeted operational environment. In (Gashi 2006) (***the preceding reference forms part of this thesis as Paper-7***) we describe an approach of combining the assessment in partitions under the assumption of independence of uncertainties across the sub-domains. Mathematical details can be found in (Gashi 2006) and ***Appendix VI-1B at the end of this paper***.

4. Empirical results from a study with off-the-shelf databases

We have reported previously results of a study on dependability of off-the-shelf database servers (Gashi, Popov et al. 2004) (***the preceding reference forms part of this thesis as Paper-1***). In this paper we will use the data collected in that study to demonstrate how the model explained in Section 3.3 can be utilized to perform the selection of the best pair of 2 servers. We note that the ideal selection of the best pair is to perform statistical testing using the COTS products. This, however, is problematic in practice due to the lack of suitable middleware[31] for diverse database replication. Database replication is non-trivial as it requires synchronizing the operation of the copies while serving concurrent clients. Additionally the software vendor of the middleware may like to make a 'strategic' choice of an SQL server pair for use in the foreseeable future. The application(s), which may be developed by the users of the middleware in the future, will be clearly unknown at the time of making the selection, therefore performing statistical testing (which is crucially dependent on knowing the operational profile in the targeted environment) will be impossible.

Given these difficulties we can use alternative options. We will describe in this paper one such option: using stressful environments which increase the likelihood of failures occurring. After all the fault-tolerant solution with a pair of servers is intended to cope

[31] Some rudimentary solutions such as C-JDBC (ObjectWeb 2006) only allow for the use of a minimal subset of SQL with diverse SQL servers.

with the difficult situations (demands) where the individual channels are deficient. The set of bugs of a particular COTS product (in our case SQL server) defines one such stressful environment for a server. We have collected known bug reports for four SQL servers, namely PostgreSQL 7.0, Interbase 6.0, Oracle 8.0.5 and Microsoft SQL server 7 (Gashi, Popov et al. 2004) (for the sake of brevity we will use the abbreviations PG, IB, OR and MS respectively throughout the rest of the text when referring to these servers). The union of the bugs reported for all the compared COTS products will form a demand space, in which there will be a partition stressing each of the products. The logs of the known bugs are treated as a sample (without replacement[32]) from the corresponding partition (representing the server, for which the bug has been reported). We label the partitions $S_{Server\ name}$. Partition S_X is called an 'own' partition for server X and a 'foreign' partition for any other server Y≠X. The servers are then compared using the methodology described in Sections 3.3 and 3.4.

4.1 Prior distributions

The prior distributions used in this study are explained next. The joint prior distribution was constructed under the assumptions that the respective *pfd*s of a server A and a server B are independently distributed; in the general case of the failures being non-independent events, the conditional distributions of the probability of coincident failure are specified for every pair of values of the *pfd*s of servers A and B.

The distributions were assumed to be identical for each of the four servers across both their 'own' and 'foreign' partitions respectively. This assumption was made because we did not have reasons to believe otherwise. We discuss other options of deriving more accurate priors in the Conclusions section. A summary of the distributions used is given in Table 17.

For 'own' partitions the prior distributions of *pfd*s of both A and B were defined as uniform in the range [L, 1], where L < 1 accounts for the chance that some of the reported

[32] Strictly, there might be a difference between sampling with and without replacement. Our model is based on sampling without replacement while the inference procedure described in Section 3.3 implies sampling with replacement. This is a simplification, which in many cases is acceptable (e.g. sampling from a large population of units, none of which dominates the sampling process, which seems a plausible assumption in our case of SQL servers being very complex products and likely to contain many unknown bugs).

bugs might be Heisenbugs[33] (Gray 1986), i.e. we expect most of the bugs that have been reported for a particular server to cause failures when they are run on that server (hence the probability of observing an incorrect results failure is very close to 1) but, due to Heisenbugs, not always so. As a source for L we used the study by Chandra and Chen (Chandra and Chen 2000). These authors studied the fault reports for three off-the-shelf products: MySQL database server, GNOME desktop environment and the Apache web-server and reported that 5%, 7% and 14%, respectively, of the reported bugs were Heisenbugs. Given the variation between the products we interpreted these findings by setting L = 1-(2*0.14), that is twice the highest value of Heisenbugs reported, i.e. allowing for even higher proportion of the Heisenbugs than recorded in (Chandra and Chen 2000). The prior, thus, is expected to be within the range [0.72, 1]. Notice that here the prior distribution for incorrect results is being defined at a range close to 1 (i.e. high unreliability). This is because of the unusual profile of the demands: since we are using known bug reports as demands we expect most of the bugs to cause failures when we run them on the server for which they were reported.

Table 17 - The Prior distributions (identical for all four servers)

Partition	Range	Distribution
pfd of server A *or* B on 'Own' partitions	0.72 – 1	Uniform
pfd of server A *or* B on 'Foreign' partitions	0 – 1	Uniform
Conditional Distribution of 'Coincident failures' in both A *and* B on either partition	0 – min (value of *pfd* of A, value of *pfd* of B)	Uniform

For 'foreign' partitions, however, the prior distributions for both *pfd*s of A or B were defined as uniform in the range [0, 1]. This is due to the absence of any comparative study to guide our expectation about the likely value. In passing we note that theoretical work such as (Littlewood and Miller 1989), (Eckhardt and Lee 1985) suggest that diverse software versions will tend to fail coincidentally on 'difficult' demands. Since all the bugs are 'difficult' – they are known to be problematic at least for one of the servers – we may consider them genuinely difficult, hence assume as plausible that the other servers too, are likely to fail on them. On the other hand, empirical studies such as (Knight and Leveson 1986), (Eckhardt, Caglayan et al. 1991), have shown that significant gains can

[33] Gray defines two types of bugs (Gray 1987): "Bohrbugs" for bugs that appear to be deterministic (they manifest themselves each time the bug script is executed); and "Heisenbugs" for those that are difficult to reproduce as they only cause failures under special conditions (e.g., created by usage pattern, other software and internal state)

be had via design diversity – hence low chances that a particular server will fail on bugs reported for other servers are also plausible. In summary, we are indifferent between the values of the probability that a server will fail from a 'foreign' bug.

All conditional prior distributions for coincident failures of the two servers for given values of the components' *pfd*s were defined in the range [0, min (value of *pfd* of A, value of *pfd* of B)] (since it cannot be greater than the probability of *either* of the two individually). This is again due to the rather unique profile, under which we apply the inference and the lack of comparable studies that would enable us to define a more accurate prior, thus 'indifference'.

For the comparison we use a distribution defined on the partitions which does not favour any of the servers, i.e. we assumed that probability of each partition is 0.25 in the study with 4 servers[34].

4.2 Observations

The observations using the known bugs of four off-the-shelf servers are given in Table 18 (Gashi, Popov et al. 2004). Since we included 4 servers in our study and we are interested in diverse pairs of servers, then we have a total of 6 different server pairs.

Table 18 - The observations for the 6 diverse server pairs on the bug reports of the different partitions. In the partition column the subscript indicates for which server these bugs have been reported. N is the total number of bugs run and r_1, r_2 and r_3 are as defined in Table 16.

Server Pair	Partition	N	r_1	r_2	r_3	Server Pair	Partition	N	r_1	r_2	r_3	Server Pair	Partition	N	r_1	r_2	r_3
PG & IB	S_{PG}	24	21	0	0	IB & OR	S_{PG}	18	0	0	0	IB & MS	S_{PG}	21	0	1	0
	S_{IB}	28	0	23	1		S_{IB}	31	25	0	0		S_{IB}	35	27	0	2
	S_{OR}	3	0	0	0		S_{OR}	4	0	3	0		S_{OR}	4	0	0	0
	S_{MS}	9	0	0	0		S_{MS}	10	1	0	0		S_{MS}	12	0	6	1
PG & OR	S_{PG}	30	27	0	0	PG & MS	S_{PG}	33	28	0	2	OR & MS	S_{PG}	27	0	2	0
	S_{IB}	24	1	0	0		S_{IB}	25	1	2	0		S_{IB}	30	0	2	0
	S_{OR}	4	0	2	1		S_{OR}	3	0	0	0		S_{OR}	4	3	0	0
	S_{MS}	7	0	0	0		S_{MS}	18	1	7	5		S_{MS}	12	0	7	0

We can see that the number of bugs collected for each server was different, which indicated that the empirical evidence differs between the partitions. The reason for this

[34] We could use the number of known bugs for each of the partition to construct a profile consistent with the observations. This is not acceptable for two reasons: i) it will favour poor bug reporting practices, an ii) we would have used the bugs twice – once in the inference procedure and another time in defining the profile, which is theoretically unsound.

was merely differences in the reporting practices operated by the vendors of the servers, e.g. unavailability in the public domain of fully reproducible bug scripts for the commercial servers (especially OR). Therefore, the sizes of the samples from the partitions on each server are different[35]. Additionally, these servers are not functionally identical: they offer different degrees of compliance with the SQL standard(s) and even proprietary extension to SQL. Bugs affecting one of these extensions, thus, literally cannot exist in a server that lacks the extension. We called these "dialect-specific" bugs. Due to this, not all the bugs reported for a server can be run on the other servers. Therefore the number of 'foreign' bug reports varies between the servers.

4.3 Posteriors

Table 19 shows the percentiles of the priors and posteriors of the probability of a failure of a pair of components assuming a 1-out-of-2 setup. The values in the cells represent the confidence that the probability of a coincident failure of both components of a pair on the same randomly chosen demand is no greater than the respective confidence level, e.g. for PG & IB the value of 0.02 at the 50th percentile can be interpreted as "we are 50% confident that the probability of a coincident failure of both PG & IB on a randomly chosen demand is no greater than 0.02".

Table 19 - The percentiles of the probability of system failure for each server pair.

Server Pair	50th percentile		99th percentile	
	Prior	Posterior	Prior	Posterior
PG & IB	0.3	0.02	0.61	0.12
PG & OR		0.07		0.19
PG & MS		0.09		0.20
IB & OR		0.02		0.14
IB & MS		0.04		0.14
OR & MS		0.02		0.14

We can see that universally the best pair across the percentiles is the open-source server pair PG & IB. There are some interesting remarks to note from the results on Table 19, which highlight the value of handling the uncertainty explicitly using probability

[35] It may seem desirable to have a similar amount of data for the different servers, but in reality there are different reporting practices for each server. Such differences simply translate into different amounts of empirical evidence available for the servers, with which our method can cope easily.

distributions, rather than using point estimates of attribute values and the value of exploiting the dependence in the failure behaviour of the servers:

- It may seem surprising that the best server pair is PG & IB given that results in Table 18 show that one coincident failure (i.e. r_3) was observed for this pair and none for the commercial server pair OR & MS. But, in Table 18 we also saw that there is a much larger number of single channel failures (i.e. r_1 and r_2) observed for the open-source server pair than for the commercial server pair which increases our confidence of a strong *negative correlation* in the failure behaviour of the open-source pair, i.e. we see extensive evidence that diversity does work: when one of the servers fails the other works correctly. No such evidence is available for the commercial servers.
- We cannot make a selection purely on the 50th percentile of the posterior distribution of the system *pfd* since 3 of the server pairs give identical results. Most of the conventional assessment techniques, which rely on median values of the assessment attributes would have also been unable to provide a clear choice. However we can make a selection from the 99th percentile of the same setup.

We have also used the model described in Section 3.1 to calculate the posteriors of single servers (using the same prior definitions as for the pairs, the observations for each individual server and utilizing the partitions theory described in Section 3.4). The posteriors for each server are shown in Table 20. We can see that even the worst pair from Table 19 on all percentiles performs better than the best single server in Table 20. This is hardly surprising given the fact that coincident failures are very rare despite the choice of a stressful demand profile (known bug reports). We can also see that the differences in the *pfd* values of a single server vs. a diverse pair of servers are quite significant.

Table 20 - The percentiles of the probability of failure on demand for each single server.

Posteriors	PG	IB	OR	MS
50th percentile	0.41	0.30	0.26	0.30
99th percentile	0.54	0.43	0.32	0.42

The worst performing server pair has a *pfd* of no worse than 0.20 with confidence 99% whereas the best performing single server has a *pfd* of no worse than 0.32 with the same

confidence level. These results indicate that the use of a diverse server pair would bring significant dependability gains: the best single server may fail up to once in 3 demands while the worst pair – up to once in 5 demands.

5. Discussion

The Bayesian model explained in Sections 3.3 and 3.4 can be used for selection of an optimal pair of COTS components, as was illustrated in Section 4, when the attribute of interest is the probability of failure on demand. It is a common practice that COTS components are assessed in terms of more than 2 attributes, usually many more. The obvious question, therefore, is whether the proposed 'uncertainty explicit' assessment 'scales up' to:

- more than one attribute
- fault-tolerant configurations in which more than two COTS components are used (for example, three COTS components to enable majority voting on the results)

In both of these cases, the question is how the method applies if we have to define multivariate distributions. Even though mathematically possible, Bayesian inference with multivariate distributions is difficult. The difficulty has two aspects:

- specifying a multivariate prior distribution becomes problematic because many non-intuitive dependencies between the attributes must be defined and *justified.*
- manipulating a multivariate distribution is non-trivial even using the most advanced math/statistical tools. Calculating the posterior distribution is impracticable with more than 3 variates and without simplifying assumptions about the dependencies between them.

For scenarios where the COTS components need to be assessed in terms of more than one attribute, to partially overcome these difficulties, a "divide-and-conquer" approach can be employed: first the attributes can be grouped into smaller groups so that there are dependencies within the groups, which the assessment can capture, but the groups are assumed independent (i.e. knowing the values of the attributes in one group does not change the assessor's knowledge (belief) about the values of the attributes included in the other group); then, due to the independence assumption, the final distribution is the

product of the distributions of the individual groups. More details on this approach can be found in (Gashi 2006).

The limitations we outlined in this section are *not specific* to our assessment method; in fact they are more serious for the conventional methods in which the individual attributes are assessed separately. We have shown in (Gashi 2006) that even when the assessment of single COTS components is done using just two attributes, ignoring the dependence between the values of the attributes may lead to wrong decisions: a sub-optimal component may wrongfully be chosen as the best one. If this could be observed with only two attributes, then it is bound to be much more pronounced with more than two attributes, where many more dependencies may exist between the values of the attributes.

The "divide and conquer" approach to attributes also has its problems. It can only be applied if the assessor can justify that assuming a set of independent pairs is plausible. Despite this problem, however, using small independent groups is still an improvement compared with the extreme assumption used implicitly in the existing assessment methods surveyed, that all of the attributes are independent.

It is worth pointing out that many of the attributes, such as 'has the required functions', various forms of compliance, e.g. 'complies with certain standards', "Backward Compatibility", etc. (Bertoa and Vallecillo 2002), do not require any uncertainty attached to their values. Assessment with respect to such attributes normally leads to a reduction of the number of the COTS components (which satisfy all these 'binary' attributes), for which the more thorough assessment with respect to the remaining 'non-binary' attributes can proceed (Ncube and Maiden 1998).

6. Conclusions

Software diversity is a well known and well studied subject in the literature (Anderson and Lee 1990). It is recognized that often the only way of obtaining dependability assurances is to employ software diversity (Littlewood and Strigini 1993). With the plethora of off-the-shelf components available fault tolerance through software diversity becomes a much more achievable and affordable solution especially since many of the components are open-source and free. The important questions for a given project is how

much dependability gains there will actually be from employing diversity, or at least given a set of diverse software alternatives which is the best for a given application.

We applied methods of Bayesian assessment developed elsewhere (Littlewood, Popov et al. 2000), (Gashi 2006). We illustrated how our model can be used with the collected evidence to perform the assessment and choose the best server pair. We then compared the results of the posteriors of server pairs with those of single servers and we saw that even the worst server pair still performs much better than the best single server. This indicates that significant dependability gains may be obtained from using diverse off-the-shelf database servers. It is also interesting to note that in our assessment the best single server is a commercial server, namely Oracle, whereas the best pair of components is the pair PostgreSQL & Interbase both of which are free and open-source components.

The prior definition in Bayesian assessment is crucial. In our study we have assumed that prior distributions for each component are the same. This was due to the unavailability of other known evidence that we could use to define more accurate priors. However this problem can be remedied by utilizing evidence from *earlier versions* of the servers and then doing multiple steps of inference, i.e. if we want to perform the assessment with later versions of the servers in our study we can use the posteriors from this step as priors for the later versions, collect the new evidence for the later versions and then use the model to derive the posteriors for each.

Future work that is desirable would be to enable effective assessment with a higher number of COTS components in a diverse setup (more than two components may be desirable in a diverse setup to enable majority voting on the results from the components).

Acknowledgement

This work was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under the 'Interdisciplinary Research Collaboration in Dependability of Computer-Based Systems' (DIRC) project.

References

Alves, C. and J. Castro (2001), "*CRE: A Systematic Method for COTS Components Selection*", *in proc.* XV Brazilian Symposium on Software Engineering (SBES), Rio de Janeiro, Brazil.

Anderson, T. and P. A. Lee (1990), "*Fault Tolerance: Principles and Practice (Dependable Computing and Fault Tolerant Systems, Vol 3)*", Springer Verlag.

Bertoa, M. F. and A. Vallecillo (2002), "*Quality Attributes for COTS Components*", *in proc.* 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002), Málaga, Spain, pp: 54-66.

Boehm, B., D. Port, Y. Yang, J. Bhuta and C. Abts (2003), "*Composable Process Elements for Developing COTS-Based Applications*", *in proc.* Int. Symp. on Empirical Software Engineering. (ISESE '03), ACM-IEEE, pp: 8-17.

Burgués, X., C. Estav, X. Franch, J. A. Pastor and C. Quer (2002), "*Combined Selection of COTS Components*", *in proc.* Int. Conf. on COTS-Based Software Systems (ICCBSS '02), Florida, USA, Springer-Verlag, pp: 54-64.

Chandra, S. and P. M. Chen (2000), "*Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '00), NY, USA, IEEE Computer Society Press, pp: 97-106.

Comella-Dorda, S., J. Dean, E. Morris and P. Oberndorf (2002), "*A Process for COTS Software Product Evaluation*", *in proc.* Int. Conf. on COTS-Based Software Systems (ICCBSS '02), Florida, USA, Springer-Verlag, pp: 86-92.

Dean, J. (2000), "*An Evaluation Method for COTS Software Products*", http://www.stc-online.org/cd-rom/cdrom2000/webpages/johndean/paper.pdf.

Dean, J. and M. Vidger (2000), "*COTS Software Evaluation Techniques*", *in proc.* The NATO Information Systems Technology: Symposium on Commercial Off-the-shelf Products in Defence Applications, Brussels, Belgium.

Eckhardt, D. E., A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk and J. P. J. Kelly (1991), "*An experimental evaluation of software redundancy as a strategy for improving reliability*", IEEE Transactions on Software Engineering 17(7), pp: 692-702.

Eckhardt, D. E. and L. D. Lee (1985), "*A theoretical basis for the analysis of multiversion software subject to coincident errors*", IEEE Transactions on Software Engineering 11(12), pp: 1511-1517.

Gashi, I., P. Popov and L. Strigini (2004), "*Fault Diversity Among Off-The-Shelf SQL Database Servers*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '04), Florence, Italy, IEEE Computer Society Press, pp: 389-398.

Gashi, I., P. Popov and V. Stankovic (2006), "*Uncertainty Conscious Assessment of Off-The-Shelf Software*", Submitted for publication, http://www.csr.city.ac.uk/people/ilir.gashi/COTS/.

Gray, J. (1986), "*Why Do Computers Stop and What Can be Done About it?*" *in proc.* Int. Symp. on Reliability in Distributed Software and Database Systems (SRDSDS '86), Los Angeles, CA, USA, IEEE Computer Society Press, pp: 3-12.

Gregor, S., J. Hutson and C. Oresky (2002), "*Storyboard Process to Assist in Requirements Verification and Adaptation to Capabilities Inherent in COTS*", *in proc.* Int. Conf. on COTS-Based Software Systems (ICCBSS '02), Florida, USA, Springer-Verlag, pp: 132-141.

Hamlet, D. and R. Taylor (1990), "*Partition testing does not inspire confidence*", IEEE Transactions on Software Engineering 16(12), pp: 1402-1411.

Jeanrenaud, J. and P. Romanazzi (1994), "*Software Product Evaluation: A Methodological Approach*", *in proc.* Software Quality Management II: Building Software into Quality, pp: 55-69.

Jeng, B. and E. J. Weyuker (1991), "*Analyzing partition testing strategies*", IEEE Transactions on Software Engineering 17(7), pp: 703-711.

Knight, J. C. and N. G. Leveson (1986), "*An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming*", IEEE Transactions on Software Engineering 12(1), pp: 96-109.

Kontio, J., S. Y. Chen, K. Limperos, R. Tesoriero, G. Caldiera and M. Deutsch (1995), "*A COTS Selection Method and Experiences of Its Use*", *in proc.* Twentieth Annual Software Engineering Workshop,NASA Goddard Space Flight Center, Greenbelt, Maryland.

Kunda, D. and L. Brooks (1999), "*Applying Social-Technical Approach for COTS Selection*", *in proc.* UK Academy for Information Systems (UKAIS'99), University of York, England.

Lewis, P., P. Hyle, M. Parrington, E. Clark, B. Boehm, A. Abts and R. Manners (2000), "*Lessons Learned in Developing Commercial Off-The-Shelf (COTS) Intensive Software Systems*", http://www.cebase.org/www/researchActivities/COTS/LessonsLearned.pdf.

Littlewood, B. and D. R. Miller (1989), "*Conceptual Modelling of Coincident Failures in Multi-Version Software*", IEEE Transactions on Software Engineering 15(12), pp: 1596-1614.

Littlewood, B., P. Popov and L. Strigini (2000), "*Assessment of the Reliability of Fault-Tolerant Software: a Bayesian Approach*", *in proc.* Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP '00), Rotterdam, the Netherlands, Springer, pp: 294-308.

Littlewood, B., P. Popov and L. Strigini (2001), "*Modelling software design diversity - a review*", ACM Computing Surveys 33(2), pp: 177-208.

Littlewood, B. and L. Strigini (1993), "*Validation of Ultra-High Dependability for Software-based Systems*", Communications of the ACM 36(11), pp: 69-80.

Littlewood, B. and D. Wright (1997), "*Some conservative stopping rules for the operational testing of safety-critical software*", IEEE Transactions on Software Engineering 23(11), pp: 673-683.

Musa, J. D. (1993), "*Operational Profiles in Software-Reliability Engineering*", IEEE Software (March), pp: 14-32.

Ncube, C. and N. Maiden (1998), "*Acquiring COTS Software Selection Requirements*", IEEE Software 15(2), pp: 46-56.

Ncube, C. and N. Maiden (1999), "*PORE:Procurement Oriented Requirements Engineering Method for the Component-Based Systems Engineering Development Paradigm*", *in proc.* International Workshop on Component-Based Software Engineering.

ObjectWeb (2006), "*C-JDBC*", http://c-jdbc.objectweb.org/.

Ochs, M., D. Pfahl, G. Chrobok-Diening and B. Nothhelfer-Kolb (2001), "*A Method for Efficient Measurement-based COTS Assessment and Selection -Method Description*

and Evaluation Results", *in proc.* 7th Symposium on Software Metrics, London, England, IEEE Computer Society, pp: 285-294.

Phillips, B. C. and S. M. Polen (2002), "*Add Decision Analysis to Your COTS Selection Process*", http://www.stsc.hill.af.mil/crosstalk/2002/04/phillips.html.

Ruhe, G. (2003), "*Intelligent Support for Selection of COTS Products*", *in proc.* Web, Web-Services, and Database Systems, Springer, pp: 34-45.

Tran, V. and D.-B. Liu (1997), "*A Risk Mitigating Model for the Development of Reliable and Maintainable Large-Scale Commercial-Off-The-Shelf Integrated Software Systems*", *in proc.* Reliability and Maintainability Symp. (RAMS'97), IEEE Print, pp: 361-367.

Wright, D. and K.-Y. Cai (1994), "*Representing Uncertainty for Safety Critical Systems*", PDCS2 Tech. Rep. 135. Center for Software Reliability, City University, London.

Appendix VI-1A – Component-Pair Assessment

Assume that the attribute of interest is the probability of failure on demand (*pfd*). Now assume that the system is subjected to a series of independently selected demands. On each demand the response received from the COTS components is characterized as correct/incorrect. But since we have two COTS components clearly 4 combinations exist, which can be observed on a randomly chosen demand, as shown in Table 16 of Section 3. The four probabilities given in the last column of Table 16 sum to unity (i.e. they sum to 1). This constraint remains even if we treat the probabilities in Table 16 as random variables: their sum will always be 1. Thus, the joint distribution of any three of these probabilities, e.g. $f_{p_{01},p_{10},p_{11}}(\bullet,\bullet,\bullet)$, gives an exhaustive description of the COTS component's behaviour. In statistical terms, the model has three degrees of freedom.

The probabilities of getting an incorrect response on a random demand from COTS A, let's denote it p_A, or COTS B, p_B, respectively, can be expressed as:

$p_A = p_{10} + p_{11}$ and $p_B = p_{01} + p_{11}$.

p_{11} represents the probability of receiving an incorrect response from both the COTS components. Hence, a notation $p_{AB} \equiv p_{11}$ will capture better the intuitive meaning of the

event it is assigned to. Instead of using $f_{p_{01},p_{10},p_{11}}(\bullet,\bullet,\bullet)$ another distribution, which can be derived from it through functional transformation, can be used. We use $f_{p_A,p_B,p_{AB}}(\bullet,\bullet,\bullet)$. We define the joint prior distribution as:

$$f_{p_A,p_B,p_{AB}}(\bullet,\bullet,\bullet) = f_{p_{AB}|p_A,p_B}(\bullet \mid p_A,p_B)\ f_{p_A,p_B}(\bullet,\bullet) \quad (3)$$

under the assumption that p_A and p_B are independently distributed, i.e.

$$f_{p_A,p_B}(\bullet,\bullet) = f_{p_A}(\bullet)\ f_{p_B}(\bullet) \quad (4)$$

It can be shown that for a given observation (r_1, r_2, and r_3 in N demands) the posterior joint distribution can be calculated as:

$$f_{p_A,p_B,p_{AB}}(x,y,z \mid N,r_1,r_2,r_3) = \frac{f_{p_A,p_B,p_{AB}}(x,y,z)L(N,r_1,r_2,r_3 \mid p_A,p_B,p_{AB})}{\iiint_{p_A,p_B,p_{AB}} f_{p_A,p_B,p_{AB}}(x,y,z)L(N,r_1,r_2,r_3 \mid p_A,p_B,p_{AB})dxdydz} \quad (5)$$

where

$$L(N,r_1,r_2,r_3 \mid p_A,p_B,p_{AB}) = \frac{N!}{r_1!r_2!r_3!(N-r_1-r_2-r_3)!}(p_A-p_{AB})^{r_1}(p_B-p_{AB})^{r_2}p_{AB}^{r_3}(1+p_{AB}-p_A-p_B)^{N-r_1-r_2-r_3} \quad (6)$$

is the multinomial likelihood of the observation (N, r_1, r_2, r_3).

The marginal distribution $f_{p_{AB}}(\bullet)$, which is used for comparison of the COTS component pairs, can be derived from $f_{p_A,p_B,p_{AB}}(\bullet,\bullet,\bullet)$ by integrating out p_A and p_B, i.e.

$$f_{p_{AB}}(\bullet) = \iint_{p_A p_B} f_{p_A,p_B,p_{AB}}(\bullet,\bullet,\bullet)dp_A dp_B \quad (7)$$

Appendix VI-1B – Partitions Theory

If the demand space is partitioned into M partitions {S_1, S_2, … S_M}, then for each of these the assessment will be performed as described in Section 3.3, e.g. with two COTS

components the description provided in Section 3.3 (with details given in Appendix A) will apply. As a result M conditional distributions will be associated with each pair of COTS components, e.g. using two components these can be denoted as $f_{p_A, p_B, p_{AB}}(\bullet,\bullet,\bullet \mid S_i)$, from which the conditional uncertainty $f_{p_{AB}}(\bullet \mid S_i)$ will be expressed. This distribution characterizes the probability of failure, $P_{AB} \mid S_i$, of both components in the specific partition. Finally, in order to compare the competing COTS pairs the unconditional distribution $f_{p_{AB}}(\bullet)$ should be derived for the particular profile defined over the set of partitions, which represents the targeted operational environment.

Let us denote the profile of the targeted environment as $\{P(S_1),...,P(S_M)\}$, and assume that these are known with certainty. The marginal probability of failure of a COTS component pair, according to the formula of full probability is:

$$P_{AB} = \sum_{i=1}^{M} P_{AB} \mid S_i \times P(S_i) \tag{8}$$

The distribution of this random variable, P_{AB}, depends on the joint distribution, $f_{P_{AB}|S_1,\ldots,P_{AB}|S_M}(\bullet,\ldots,\bullet)$, i.e. of the conditional probabilities of failure in sub-domains. In some setups it may be plausible to assume that the conditional probabilities of failure (in the partitions that is) are independently distributed, i.e.:

$$f_{P_{AB}|S_1,\ldots,P_{AB}|S_M}(\bullet,\ldots,\bullet) = \prod_{i=1}^{M} f_{P_{AB}|S_1}(\bullet)\ldots f_{P_{AB}|S_M}(\bullet) \tag{9}$$

Such an assumption represents the assessor's belief that learning something about the probability of failure, $P_{AB} \mid S_i$, of a particular COTS component pair in partition *i* will not change their belief about the probability of failure, $P_{AB} \mid S_j$, of the same COTS component pair in another partition. The assumption is consistent with applying inferences to the individual partitions, i.e. conditional on the demands coming from a particular partition.

Under (9) the unconditional probability of COTS component pair failure (8) can be expressed as a convolution of the distributions of the random variables $P_w(i) = P_{AB} \mid S_i \times P(S_i)$, i.e.:

$$P_{AB}^{w} = \otimes P_w(i) \tag{10}$$

The selection of the best COTS component pair, out of the available alternatives, then will be based on the marginal distributions, $f_{p_{AB}^{w}}(\bullet)$, associated with the available COTS component pairs.

Paper-6. Reliability Growth Modelling of a 1-Out-Of-2 System: Research with Diverse Off-The-Shelf SQL Database Servers

Abstract: *Fault tolerance via design diversity is often the only viable way of achieving sufficient dependability levels when using off-the-shelf components. We have reported previously on studies with bug reports of four open-source and commercial off-the-shelf database servers and later release of two of them. The results are very promising for designers of fault-tolerant solutions that wish to employ diverse servers: very few bugs cause failures in more than one server and none cause failure in more than two. In this paper we offer details of two approaches we have studied to construct reliability growth models for a 1-out-of-2 fault-tolerant server which utilize the bug reports. The models presented are of practical significance to system designers wishing to employ diversity with off-the-shelf components since often the bug reports are the only direct dependability evidence available to them.*

Co-authors: Prof. Peter Bishop, Prof. Bev Littlewood, Dr. David Wright

Conference: IEEE International Symposium on Software Reliability Engineering 2007 (ISSRE-07)

Date of submission: 30-April-2007

Status: *Under Review*

Number of reviewers: TBC

Publication date: TBC

Full citation: TBC

1. Introduction

Off-the-shelf (OTS) components are used ubiquitously in software systems development due to the perceived lower costs from their use (some of the components may be open-source and/or freely available), faster deployment and the multitude of available options. There remain concerns, however, about the dependability levels of the components: they tend to be distributed without any assurances of their dependability, with "use-as-is" labels often attached to them by the vendors. As a result, the only viable way available to users and system integrators of achieving higher dependability is to use software fault tolerance. Fault tolerance may take multiple forms, with examples ranging from simple error detection and recovery add-ons (e.g. "wrappers" (Popov, Strigini et al. 2001)) "diverse modular redundancy" (e.g. "N-version programming": replication with diverse versions of the components) (Strigini 2005).

The design decisions are well known from the literature. Questions remain however about the dependability gains that developers of systems using OTS components can expect, the implementation difficulties and the extra cost expected. We have studied some of these issues with OTS database servers or database management system (DBMS) products: a highly complex category of OTS components. The architectural solutions for implementing a fault tolerant DBMS using diverse OTS database products are given in (Gashi, Popov et al. 2007) (***the preceding reference forms part of this thesis as Paper-2***).

With regard to the dependability of a fault tolerant DBMS, we have reported previously on a study with the publicly available fault reports of four OTS DBMS products (both open-source and closed development) (Gashi, Popov et al. 2004) (***the preceding reference forms part of this thesis as Paper-1***) and later releases of two of them (Gashi, Popov et al. 2007). We found that a high number of these faults would not be tolerated (or even detected) by the existing non-diverse fault-tolerant schemes but did not cause failures in any two diverse DBMS products. We found the number of faults that caused coincident failures to be very low. These results seem to suggest that significant dependability gains may be achieved if diverse modular redundancy is employed with OTS DBMS products. However they are not definitive evidence. The main problem is that the available reports concern faults (bugs) and not how many failures each caused,

which makes their use in reliability predictions difficult. Complete failure logs would be much more useful as statistical evidence, but they are not available. The only direct dependability evidence available for these products often are the fault reports.

It is the absence of failure data and the lack of known approaches that can utilize existing fault reports of OTS components in reliability assessment that has motivated the research detailed in this paper. More precisely, the question we attempt to answer is "how can we incorporate existing evidence for off-the-shelf products to evaluate the possible gains in reliability achievable by a 1-out-of-2 diverse server?" To this end we have studied two approaches which use fault reports for obtaining dependability measures of a fault tolerant server employing two diverse OTS DBMS products For the sake of brevity, we shall refer to this fault tolerant DBMS as a "FT-node".

The two approaches presented in this paper for estimating the reliability of a FT-node are:

1. An extension of a previous software reliability growth model (Littlewood 1981) for use in reliability growth modelling of the FT-node.
2. An alternative "proportions" approach where the observed reliability of a single server is scaled by a factor to derive the expected reliability of the FT-node.

The first method requires information on actual usage time. In closed development environments, it should be feasible to derive usage time from dated fault reports if the total population of the DBMS product is known over time (e.g. from product registration information). However for open source products, information on the product population over time is hard to obtain, and hence the usage time is difficult to estimate.

We have therefore developed a second method where information about usage time is not required and statements about the reliability improvement achievable by an FT-node can be made (under certain assumptions about the underlying failure rate distributions) based only on information derived from reported product faults.

The paper is structured as follows: Section 2 contains background on the studies we have conducted with known fault reports of the DBMS products, software reliability growth modelling and the Littlewood (Littlewood 1981) model; Section 3 details the extensions of the Littlewood model (Littlewood 1981) for the reliability growth modelling of the FT-node; Section 4 contains details of an alternative model in which fault counts alone are used for reliability prediction of the FT-node; in the same section we also provide

empirical data to illustrate the use of the method; Section 5 contains a discussion and verifications of the main modelling assumptions made and finally Section 6 contains a discussions of the two modelling approaches, conclusions and provisions for further work.

2. Background and related work

2.1 Analysis of common faults in OTS DBMS products

We have conducted two studies with fault reports of four OTS DBMS products and later releases of two of them. We have fully described these studies and provided analysis of the results in (Gashi, Popov et al. 2004) and (Gashi, Popov et al. 2007). We will be utilizing the results of those studies in this paper as empirical evidence with one of the models, as well as for verification of the assumptions. Therefore, in what follows we will provide a brief summary of the studies and the main results.

A mixture of free open-source and commercial closed development products were used in the studies. In the first study we collected a total of 181 bugs reported for the following DBMS products (for the sake of brevity, we will use the abbreviations (detailed in brackets next to each product), when referring to these products from this point forward. The first two products in the list are open-source; the last two are commercial closed-development):

- Interbase 6.0 (IB)
- PostgreSQL 7.0 (PG7.0)
- Oracle 8.0.5 (OR)
- Microsoft SQL Server 7 (MS).

We first ran the bug scripts (contained within the bug reports) on the products for which they were reported and then (when possible[36]) on the other products. We found very few bugs that caused coincident failures in more than one DBMS product, and none which caused failure in more than two.

[36] Even though all of these DBMS products are compliant with the SQL language, each of them also implements their own proprietary extensions. Therefore some faults could be run on only one (or a subset of the four) DBMS products.

The results were encouraging, but they only represented one snapshot in the evolution of these products. Therefore we repeated the study for the later releases of the two open-source products (due to difficulties with data collection no new bug reports were collected for the commercial products):

- Firebird 1.0 (FB) (this is the open-source descendant of Interbase 6.0)
- PostgreSQL 7.2 (PG7.2)

We collected 92 new bugs reported for these two products. The results of the second study substantially confirmed those of the first: very few bug reports caused coincident failures. This suggests that factors that make diversity useful do not disappear as the DBMS products evolve and is a further indication that diversity with OTS products certainly deserves further study.

2.2 Software reliability growth modelling

Software reliability growth modelling is a well studied subject over the previous thirty years. A good reference to the subject is (Lyu 1996). Chapter three of (Lyu 1996) provides a comprehensive survey of the well known models. In the next sub-section we will provide details of one of these models which has been extended in this paper.

2.3 Littlewood model

In what follows we will stick to the notation that was first described in the Littlewood model (Littlewood 1981) and the assumptions made there. In the Littlewood model (Littlewood 1981) (and in reliability growth modelling in general), interest centres upon time-to-failure distributions and the data is a sequence of successive execution times between the failures $t_1, t_2, \ldots t_i$. The following assumptions are made:

1. Each of the N (the number of faults that exist in the OTS software product at its release) faults will cause a failure after a time which is distributed exponentially, and independently of other faults, with rate Φ_i,
   where $\Phi_1, \ldots, \Phi_N$ are *independent identically distributed (i.i.d.)* random variables,
2. When a failure occurs, there is an instantaneous removal of the fault which caused the failure,

3. If a total time τ has elapsed and i faults have been removed, the failure rate of the program is:

$$\Lambda = \Phi_1 + \ldots + \Phi_{N-i}$$

4. When debugging starts each Φ_i has the probability density function (*pdf*) $b\,\Gamma(b\phi; a)$

where Γ is the Gamma Distribution with parameters[37] a and b, with ϕ being the realization of the random variable Φ_i.

Following on from these assumptions it is shown in (Littlewood 1981) that the times, T_i, at which the faults show themselves are *i.i.d.* random variables and they are *Pareto* distributed:

$$P(T_i < t) = 1 - (\frac{b}{b+t})^a \qquad (11)$$

The motivation behind these assumption and the full details of the model can be found in (Littlewood 1981).

3. Extending the Littlewood model

In this section we discuss how the Littlewood model can be extended for reliability growth modelling of 1-out-of-2 FT-node (i.e. the FT-node is assumed to fail only if both of its components fail on a particular demand). Let's assume that the FT-node is made up of two OTS DBMS products A and B. Each of these DBMS products has their own log of fault (bug) reports. We collected the faults reported for each DBMS product over specific periods and ran the "bug scripts" (which are contained in the fault reports) on both the DBMS product for which the fault was reported and on the other DBMS product. Even though we have observed faults that *cause* failures in more than one DBMS product, we have not observed any fault which has been *reported* for more than one DBMS product; however double-reporting may happen, therefore our inference method accepts double-reported data. Each fault is therefore characterized from two dimensions: whether it *causes* failure in either or both DBMS products A or B; which DBMS product[s] the fault was *reported* for. Therefore for any two DBMS products A or B we have five types of

[37] We defined the parameters of the gamma Distribution as a (shape) and b (scale) instead of the conventional α and β since we will define β for a different purpose in Section 4.

faults, as detailed in Table 21. Clearly, the faults which will cause a 1-out-of-2 FT-node failure are the ones which make both DBMS products fail (the columns in Table 21 shaded in grey).

Table 21 – The types of failures caused by each fault on the DBMS products A and B of the FT-node

	Fault reported for A only		*Fault reported for B only*		*Fault reported for both*
Failure in A	Yes	Yes	Yes	No	Yes
Failure in B	Yes	No	Yes	Yes	Yes

Deriving the inter-failure times is more complicated than when only a single failure log is used. Since the fault reports come from potentially thousands of installations worldwide, it is virtually impossible to get accurate measures of the operational time and hence inter-failure times. If the vendors maintain detailed information about the date and time when the fault was reported (this is usually available) and date and time as well as frequency of the downloads of the DBMS products, then a (simplified[38]) proxy for the inter-failure time might be the inter-reporting time which can be calculated for each fault report as follows[39]:

Inter-reporting time =
{(calendar time of current fault report) –
(calendar time of previous fault report)} *
(number of DBMS product downloads since the release of the version being used)

Since the fault reports come from two different logs (one for each DBMS product) then we will have two different sets of inter-reporting times; additionally we also need to consider the inter-reporting time that we assign to the faults that have been reported for both DBMS products. These are inference issues and to deal with them the Likelihood equations of the Littlewood model have been extended (see ***Appendix VI-2A at the end of this paper*** for the details). The Littlewood model remains unchanged and the assumptions described in Section 2.3 were retained apart from the following extensions:

1. We assume that the operational profiles (averaged over all users) of the two different DBMS products are the same.

[38] There are limitations with this simplified calculation which will be discussed in Section 4.

[39] Unless additional information exists about the server downloads, the following assumptions are made to derive the expression: there is a single installation for each of the downloaded servers; the installation is running round the clock from the time of the download until the present time; all the server downloads are still in operation.

2. Depending on the failure they cause (DBMS product A-only, DBMS product B-only or both DBMS products A and B) there are three different fault totals N_A, N_B and N_{AB} of faults initially present in the software. The time to discovery of these faults will still be assumed to be conditionally exponentially distributed random variables given the failure rate distribution, but the failure rate may be different for each type of failure.
3. While collecting the fault reports for the various OTS DBMS products, we noted that fault reporters are specifically instructed not to report already known faults. This may be thought of as satisfying assumption 2 of the Littlewood model: a known fault will not be reported again, so subsequent data is as though it had been fixed.
4. The failure rate distributions for each of the Φ_A, Φ_B and Φ_{AB} (for the three different types of failures that the faults cause) are assumed to be drawn from the same gamma distribution.

We will discuss in more detail the assumption 4 above in Section 5.

The DBMS product releases that we have used in our studies with the faults (Gashi, Popov et al. 2004), (Gashi, Popov et al. 2007) are relatively old (the later releases of the open-source DBMS products were released in mid-2002; the DBMS products used in our first study were released in year 2000 or earlier). Therefore no data existed for the download rates of these DBMS products. As a result, proxies for usage time that would allow empirical measurements with the extended Littlewood model, could not be calculated. SourceForge (SourceForge 2006) does keep download numbers for the later releases of Firebird (release 1.0.3 onwards), but does not contain any data for the older releases of the DBMS products that we have used.

Information on the number of installations could, in principle, be derived for closed development DBMS products (such as Oracle and Microsoft SQL server), but further research is needed before the theory can be evaluated using data from such sources.

4. The proportions approach

In this section we will explain a different approach which attempts to get away from the need to quantify actual usage time.

This alternative approach is useful in applications where it may be difficult to quantify the usage time, and hence difficult to use the model described in Section 3. Quantifying the usage time is especially difficult for open-source OTS software products. Some of these difficulties are:

- Faults are reported in calendar time; it is difficult to quantify how much usage time this represents
- Even if "proxies" for usage time may be calculated (as we discussed in Section 3) there are several issues that still remain:
  - Multiple download sites and/or mirrors exist for each product
  - OTS products are often distributed as part of operating systems
  - Even if users can be quantified, the actual usage time of the products by each user will remain difficult to quantify.

The alternative approach to modelling the reliability of a 1-out-of-2 FT node is to use:

- the *counts* of faults which are available from the fault logs of each product. From it we can then calculate the *proportion* of faults in product A that are also found to cause failure in product B, β_{AB}, (from the ratio of common to non-common faults in the fault history of A). Similarly we can also calculate β_{BA} for product B faults that are also found to cause failure in A.
- the *pfd* (probability of failure on demand) of the products A and/or B; estimates of these may exists for a particular application based on actual failures in operation for that application

This approach has the following underlying assumptions:

- Common faults are drawn from the same failure rate distribution as non common faults, i.e. a constant proportion of faults in each failure rate band are common to A and B.
- The failure rate distributions for A and B are the same.

These assumptions are identical to those made for the extension to the Littlewood model described in Section 3. Given these assumptions, we can estimate the *expected* common mode failure rate as:

$E(\lambda_{AB}) = \beta_{AB} E(\lambda_A)$ or

$E(\lambda_{BA}) = \beta_{BA} E(\lambda_B)$

Where $E(\lambda_{AB})$ and $E(\lambda_{BA})$ both represent common mode failure rate estimates that should be, in principle, equivalent. In what follows we will describe in more detail how these two expressions were obtained.

4.1 The underlying theory of the proportions approach

Fault density represents the number of faults within a given failure rate interval that remain in a component. We assume the *fault density functions* of the A, B and AB functions are:

$$h(\phi)_A = N_A \, p(\phi) \quad (12)$$

$$h(\phi)_B = N_B \, p(\phi) \quad (13)$$

$$h(\phi)_{AB} = N_{AB} \, p(\phi) \quad (14)$$

where:

- N_A is the total number of faults in Product A
- N_B is the total number of faults in Product B
- N_{AB} are faults common to Products A and B
- $p(\phi)$ is the probability distribution of failure rate[40] ϕ for a fault in the product (assumed to be the same for A, B and AB faults).

Under these assumptions, the expected number of faults $n_{A,\tau A}$ observed in product A after some usage time τ_A is:

$$E(n_{A,\tau A}) = N_A \left(1 - \int_0^\infty p(\phi) e^{-\phi \tau_A} d\phi\right) \quad (15)$$

The expected number of faults $n_{AB,\tau A}$ observed in product A that are common to product B after some usage time τ_A is:

$$E(n_{AB,\tau A}) = N_{AB} \left(1 - \int_0^\infty p(\phi) e^{-\phi \tau_A} d\phi\right) \quad (16)$$

It can be seen that the assumption of a common failure rate distribution means that the bracketed term (the probability a fault is found after time τ_A) is identical for $E(n_A)$ and

[40] We use λ for the failure rate of an entire *program* (i.e. Product A, Product B or FT-node AB failure rate), and we use ϕ for the failure rate of a randomly chosen *fault*.

$E(n_{AB})$ and will cancel out if we take the ratios. So knowledge of the actual usage time τ_A and the failure rate distribution $p(\lambda)$ is not required.

So we can estimate β_{AB} from the fault sequence observed in product A up to τ_A, where some faults in the sequence are labelled as being common to B (from a knowledge of the B product faults). Given the observed values, $n_{A,\tau A}$ and $n_{AB,\tau A}$:

$$\beta_{AB} = N_{AB}/N_A \sim n_{AB,\tau A}/n_{A,\tau A} \quad (17)$$

Similarly, we can also estimate β_{BA} from the fault sequence observed in product B up to τ_B

$$\beta_{BA} = N_{AB}/N_B \sim n_{BA,\tau B}/n_{B,\tau B} \quad (18)$$

These β values need not necessarily be identical as one product could contain more faults than another.

If we now consider the use of a product for a particular application, the operational profile is likely to differ from the average usage profile for the product which determines the average failure rate distribution $p(\phi)$. For a different usage profile there will be a new failure rate distribution $p(\phi)'$. However if common faults are randomly chosen from the set of available faults, there is no reason to believe that the proportion of common faults will change for any given failure rate ϕ), i.e. we assume that:

$$h(\phi)'_A = N_A\, p(\phi)' \quad (19)$$

$$h(\phi)'_{AB} = N_{AB}\, p(\phi)' \quad (20)$$

So the expected failure rates are:

$$E(\lambda_A) = N_A \int_0^\infty p(\phi)'\phi d\phi \quad (21)$$

$$E(\lambda_{AB}) = N_{A\beta} \int_0^\infty p(\phi)'\phi d\phi \quad (22)$$

and hence:

$$E(\lambda_{AB}) = \beta_{AB}\, E(\lambda_A) \quad (23)$$

and similarly:

$$E(\lambda_{BA}) = \beta_{BA}\, E(\lambda_B) \quad (24)$$

The estimates of the performance of each DBMS product, $E(\lambda_A)$ and $E(\lambda_B)$ are derived from testing or standalone operation for the actual application, and the β values are estimated from the bug history.

In practice, the estimates $E(\lambda_{AB})$ and $E(\lambda_{BA})$ are likely to differ due to uncertainties in the β and λ values and, in this case, the most conservative estimate should be used.

4.2 Empirical derivation of β

In this section we will use the results of our previous studies with the bugs (Gashi, Popov et al. 2004), (Gashi, Popov et al. 2007) (which we summarized in Section 2.1) to derive empirical estimates of β. The results of the first study from running the DBMS product faults that could be run on each pair and the failures that they cause are given in Table 22:

- n_A are faults reported in product A
- n_{AB} are product A faults that also affect B
- n_B are faults reported in product B
- n_{BA} are product B faults that also affect A

The results presented in Table 22 do not distinguished between "Heisenbugs" and "Bohrbugs"[41], i.e. we assume the fault will always cause a failure in the DBMS product for which it was reported even if when we tested it in our setup we did not observe the failure that was detailed in the bug report. Therefore the estimates that we will get for β will be conservative.

Table 22 – The results of running the faults on each DBMS product pair. First DBMS product in the pair is labelled A and the second one B.

Pair: Failure in:	n_A	n_{AB}	n_B	n_{BA}
IB-PG7.0	28	1	24	0
IB-OR	31	0	4	0
IB-MS	35	2	12	1
PG7.0-OR	30	0	4	1
PG7.0-MS	33	2	18	6
OR-MS	4	0	12	0

[41] Terms introduced by Gray (Gray 1986), defining two types of bugs: "Bohrbugs" appear to be deterministic (the failures they cause are easy to reproduce in testing); "Heisenbugs", are difficult to reproduce as they only cause failures under special conditions: "strange hardware conditions (rare or transient device fault), limit conditions (out of storage, counter overflow, lost interrupt, etc.) or race conditions"

The β values obtained for this dataset are given in Table 23. The table also contains 90% upper confidence bounds on the estimates. The confidence bound is computed using:

$$\Pr(\beta < p \mid n,x) = \sum_{r=0,x} C(n,r)p^r(1-p)^{n-r} \quad (25)$$

where x is the number of common faults in a sequence of n faults.

Table 23 – Estimates of β for each DBMS product pair

Pair	β_{AB}	90% bound	β_{BA}	90% bound
IB-PG7.0	0.036	0.132	0	0.092
IB-OR	0	0.072	0	0.436
IB-MS	0.057	0.145	0.083	0.288
PG7.0-OR	0	0.074	0.250	0.679
PG7.0-MS	0.061	0.153	0.333	0.511
OR-MS	0	0.436	0	0.175

Note that the number of faults for a product (like IB) is not constant for different partners (like PG7.0 and MS) as it only includes the subset of faults that can be run on the product pair.

β values vary considerably for the different product pairs but the number of common faults is low (and sometimes zero) so the estimation errors are large. From equations (17) and (18), it can be seen that the β_{BA}, β_{AB} values need not be identical as they depend on the number of residual faults, N_A and N_B, which can vary with the quality of the development process. However many of the β_{BA}, β_{AB} values for the DBMS product pairs seem to be similar given the inherent sampling errors. The main exceptions to this observation are the *PG7.0-MS* and *PG7.0-OR* DBMS product pairs where the β_{BA} value exceeds the 90% confidence bound estimate for the β_{AB} value. The proportions theory indicates that this would occur if PG7.0 had significantly more residual faults than OR and MS.

Taking the data set as a whole, the results suggest that for most diverse DBMS product pairs, β values of 0.1 (and possibly lower) are possible. This means that using a 1-out-of-2 FT-node may reduce the failure rate 10 fold or more compared with a single DBMS product. We also compared β values for successive versions of the same DBMS product pairs using the results from our second study (Gashi, Popov et al. 2007) (described in Section 2.1). These can be compared with the faults found in the earlier releases. The results are given in Table 24.

Table 24 - The results for different releases

Pair: Failure in:	n_A	n_{AB}	n_B	n_{BA}
IB-PG7.0	28	1	24	0
FB-PG7.2	30	1	18	1

The β factor estimates for earlier and later releases of the products are given in Table 25.

Table 25 - β values for different releases

Pair	β_{AB}	90% bound	β_{BA}	90% bound
IB-PG7.0	0.036	0.132	0.000	0.092
FB-PG7.2	0.033	0.124	0.056	0.200

The β values seem relatively consistent between different releases of the same product pair (around 0.035). This might indicate that the relative improvement can be estimated from previous releases of the DBMS product pairs (where more data may be available). However it is difficult to draw any firm conclusions due to the high uncertainty in the estimations.

5. Validity of assumptions

The two underlying assumptions of both the approaches that have been discussed in this paper are that: the failure rate distributions for A and B are the same; the distribution of AB-faults is also the same as those of A and B. The following subsections consider whether these assumptions are credible, and present some statistical tests of the underlying assumptions.

5.1 Similar failure rate distribution assumption

There is some justification for believing the assumption that the failure rate distributions for the DBMS product pairs are the same. The research by Adams (Adams 1984) shows that there is remarkable consistency in the failure rate distributions of *different* operating systems from the same supplier. In addition, in previous work by one of the authors of this paper (Bishop and Bloomfield 2003), it was argued that the failure rate distribution is determined by the complexity of the program structure and the failure rates are likely to have the same (log-normal) distribution. This theory is consistent with the empirical observations in Adams (Adams 1984).

5.2 Conservatism of the common failure rate assumption

It is also assumed that AB faults have the same distribution as the A and B faults. This would be the case if the AB faults are not "special" in any way (i.e. the AB faults are chosen at random from the set of A faults). For the empirical results presented in Section 4.2, the AB faults chosen differed for each DBMS product pair. So no faults were observed that were common to three products (i.e. there are no "special" AB faults that occur very frequently). This gives some credence to the idea of random selection (as a bias towards selecting the same common faults should make triple common faults more likely).

We also note that an assumption of an identical distribution of A and AB failure rates would be conservative if there is a higher proportion of AB faults at higher failure rates. In this case, the β factor calculation based on the higher rate faults would *over*estimate the β value of the remaining faults, and hence *over*estimate the common failure rate using equations (23) or (24).

Some empirical experiments (Meulen, Strigini et al. 2005) suggest that the β factor decreases from a high value down to a "plateau" as the higher failure rate faults are excluded from the fault set. This might be expected if additional coincident failures occur when dissimilar faults occupy a large proportion of the input space (and hence are more likely to overlap with other faults in the input space). If this was generally true for product pairs, the assumption of common failure rate distributions for A, B and AB faults would be *conservative* (as the β factor would be *over*estimated for the low failure rate faults remaining in the two products).

5.3 Statistical tests of the "constant proportion of common faults" assumption

The assumption of constant proportion of common faults can be tested using the data taken from the fault histories. Basically we would expect the sequence of common faults to be scaled to the sequence of all faults, as illustrated in Fig. 13.

We have used two methods to check whether the steps are consistent with the linearity assumption for the fault reports in our studies with the faults:

- We constructed a *u-plot*[42] (Brocklehurst and Littlewood 1996) for the DBMS product pair PG7.0-MS on MS faults and checked whether the Kolmogorov-Smirnov (KS) distance value obtained is statistically significant.
- Divided the sample of fault reports for each DBMS product in two equally sized groups and performed the following tests to check whether there is a difference in the number of common faults observed between the two groups:
  - Fisher's Exact test (Fisher 1922)
  - Binomial proportions test (Institute of Phonetic Sciences 2006)

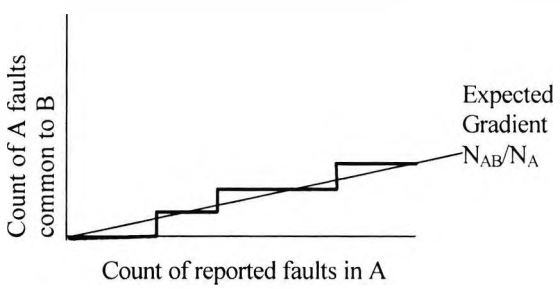

Fig. 13 - Illustration of constant proportions

5.3.1 U-plots

In the earlier work (Brocklehurst and Littlewood 1996) on prediction analysis, *u-plots* were used to check for consistent differences between the sequence of functions $F^\wedge_i(t_i)$ (the predictions) and t_i (the actual values). The sequence of numbers u_i were calculated as

$$u_i = F^\wedge_i(t_i) \quad (26)$$

Each element of the sequence is $P(T_i \leq t_i)$ (previous predictive probability that the failure time will be lower than it's subsequently observed value).

With the fault reports we have actual values and not predictions (here, the u_i represent the relative distance along the chronological fault sequence at which the i^{th} coincident fault is observed). We want to check whether the fault reports of DBMS product A which were also found to cause a failure in DBMS product B are equally likely to occur at any stage in the (ordered) history of fault reports for A. If this is the case then the step function depicted in the *u-plot* should not deviate significantly from the unit-slope (which is the cumulative uniform distribution function), i.e. using the hypothesis testing terminology:

H_0: The u_i are uniformly distributed random variables

[42] *U-plots* can be used (in our case) to test for deviations of the observations from the unit slope.

H_1: The u_i are not uniformly distributed random variables

We can produce u-plots for DBMS product pairs in which coincident failures were observed. However, as we saw in Table 22 and Table 24, the number of coincident failures observed is very small (≤ 2) for all but one pair (the PG7.0-MS pair on MS faults). We will therefore show only one of the u-plots: for the pair PG7.0-MS on the MS faults. This *u-plot* is depicted in Fig 14.

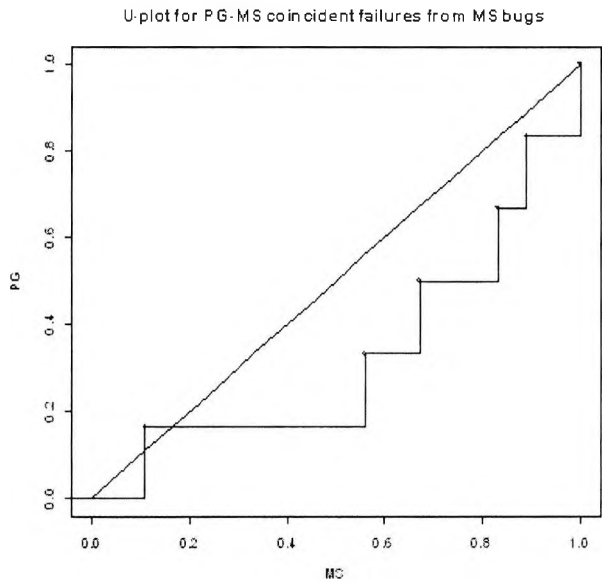

Fig. 14 - The u-plot computed for the coincident failure causing faults in PG7.0 and MS by the faults reported for MS.

The explanation of the u-plot follows: a total of 18 fault reports of MS could be run on the PG7.0 DBMS product (we'll call it n); of those 6 caused a coincident failure in PG (we'll call it r). Since there are 6 coincident failure faults there will be a total of six steps in the *u-plot* function. The size of each step is $1/r$. The u_i values on the x-axis will represent the point i/n, i.e. the sequence in which the fault was reported in MS. Therefore if the second fault of MS was found to cause a coincident failure in PG7.0 this will be shown as the point 2/18 (i.e. 1/9) on the x-axis. The *KS distance* for this pair is 0.3889 with the p-value 0.5041. Since the p-value is so large we do not have enough evidence to reject H_0 for this DBMS product pair: we do not have enough evidence to reject the claim that u_i are uniformly distributed random variables. Therefore on this dataset there is not enough evidence to reject the assumption of constant proportion of common faults.

5.3.2 Tests for equality of proportions

We can also verify the assumption of constant proportions of coincident failures by partitioning the sample of faults observed and checking whether the proportion of coincident faults differs significantly between the partitions. Initially we have done this by partitioning the samples in two. To illustrate how this was done, we will again use the MS faults that could also be run on the PG7.0 DBMS product. We have a total of 18 fault reports. We split the sample in half chronologically and check whether the proportion of MS faults reported earlier that cause coincident failures in PG7.0 differs significantly from the proportion of later MS faults. We therefore have two partitions each with 9 fault reports and the common faults found in each half is 1 and 5 respectively. The full details for each pair of DBMS products on each dataset are given Table 26. The table also contains the results of performing the Fisher's Exact test and the Binomial proportions test. The Fisher's Exact test used is the one for 2 X 2 tables. Fisher's exact test for 2 X 2 tables is used when members of two independent groups can fall into mutually exclusive categories. Quoting from (Preacher 2001): "The test is used to determine whether the proportions of those falling into each category differ by group."

Table 26 - Results from performing the Fisher's exact and Binomial proportions tests on the data sets of the faults study (after the data sets were partitioned into two halves).

DBMS product pair	Faults reported for DBMS product	N_1	AB_1	N_2	AB_2	Fisher's exact test		Binomial Proportions:
						Exact probability	*p*-value	*p*-value
IB-PG7.0	IB	14	1	14	0	0.5	0.5	0.309
IB-PG7.0	PG7.0	12	0	12	0	N/A	N/A	N/A
IB-OR	IB	15	0	16	0	N/A	N/A	N/A
IB-OR	OR	2	0	2	0	N/A	N/A	N/A
IB-MS	IB	17	2	18	0	0.2286	0.2286	0.134
IB-MS	MS	6	0	6	1	0.5	0.5	0.296
PG7.0-OR	PG7.0	15	0	15	0	N/A	N/A	N/A
PG7.0-OR	OR	2	1	2	0	0.5	0.5	0.248
PG7.0-MS	PG7.0	16	1	17	1	0.5152	0.7728	0.965
PG7.0-MS	MS	9	1	9	5	0.0611	0.0656	0.0455
MS-OR	MS	6	0	6	0	N/A	N/A	N/A
MS-OR	OR	2	0	2	0	N/A	N/A	N/A

The Binomial Proportions test (the last column of Table 26 contain the *p*-values of this test), is only an approximation, whereas the Fisher's exact test calculates the exact probability. Note that the problem of low sample sizes for coincident faults remains. Whenever possible (i.e. when the number of coincident failures is not 0) we have also

tried to calculate the p-values, but we warn the readers that, due to the small sample sizes, these values should be taken with caution.

We can see in Table 26 that the p-values for the Fisher's exact test are not statistically significant at the 5% level. This is due to there being little difference between the two partitions, or in the case of PG7.0-MS on MS faults, the sample size being too small for the p-value to be significant. For this latter pair the value is statistically significant at the 10% level (on the MS faults). For the Binomial Proportions test only the p-value of PG-MS on MS faults is statistically significant at the 5% significance level (the p-value is 0.0455); none of the others are significant at the 5% or 10% level.

6. Discussion and Conclusions

Two approaches to predicting the reliability of a 1-out-of-2 FT-node were described in Sections 3 and 4. These methods are based on some strong assumptions about the operational profile and failure rate distributions which may not hold in real operation. Ideally we would like to have detailed information about failure counts and usage time. However vendors discourage users from reporting already known faults and detailed failure data are rarely available even to the software vendors themselves. Also due to the various non-restrictive license agreements of the open-source DBMS products, a DBMS product may be downloaded from a multitude of sources and then installed in many different instances, which makes estimation of the usage time of the DBMS products very difficult. Faced with these difficult problems of data availability, it was necessary to make these strong modelling assumptions in order to derive initial estimates of the potential benefits of fault tolerance with SQL DBMS products.

In Section 5.2 we argued that assuming a common failure rate distribution for A-, B- and AB-faults is conservative. Also we have observed in earlier research (Gashi, Popov et al. 2004), (Gashi, Popov et al. 2007) that AB-faults can fail in different ways in the two DBMS products, and hence can be detected (and potentially corrected (Gashi and Popov 2006) (***the preceding reference forms part of this thesis as Paper-4***)). As a result, the estimates that we get using our models for the reliability benefits of diversity will most probably be *under*estimates: the true benefits may be higher. Despite this conservatism, using the reported faults for the DBMS products in our studies, we would expect an order

of magnitude increase in reliability when switching from a single DBMS product to a 1-out-of-2 FT-node. This result should however be treated with caution, due to the small sample sizes and relatively high estimation errors. There also appear to be variations in dependability improvement between different DBMS product pairs.

We used the reported faults from our studies to test for statistical significance of the "constant proportion of common faults" assumption, using u-plots and two tests for difference between proportions (namely Fisher's exact and the Binomial Proportions tests). We found that these tests are giving reasonably consistent results with regard to whether the hypothesis of constant proportion of common faults should be rejected. Using these tests at the 90% confidence level, we found, at most, one case out of 12 where the null hypothesis was rejected (and typically 1 in 10 cases might be rejected at the 90% confidence level when the hypothesis is true). This would indicate that the assumption of constant proportions does seem to hold for the dataset that we have.

In summary, for users who want to assess the likely dependability gains achievable if they switch from using a single DBMS product to a 1-out-of-2 diverse server then:

- if the only dependability data available for the DBMS products are the fault reports, and reasonable estimates can be obtained for the failure rate of the DBMS product they are using, then the model described in Section 4 can be used to calculate the likely improvements in reliability that they may expect from the changeover to a diverse setup
- if proxies can also be obtained for usage time, then the extended Littlewood model, described in Section 3, may also be used to assess the improvements as well as obtain other estimates such as:
  - the distribution of the common faults
  - predictions about the expected time to next diverse server failure etc.
- the two approaches may also be used sequentially to improve the predictions:
  - the β values using the proportions approach of Section 4 are calculated first
  - these β values are then used as *priors* for the b parameter of the extended Littlewood model

Further research is needed to validate the theory presented in this paper. This research includes:

- Methods for obtaining more accurate proxies for usage time.
- Empirical investigations of the predictive performance of the proportions model for actual DBMS product pairs.
- Empirical investigations of the consistency of β factor estimates in successive releases of the same product pair.
- Applying the method to other types of off-the-shelf components (such as diverse web-servers and application servers).

Acknowledgment

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council (EPSRC) via projects DOTS (Diversity with Off-The-Shelf components, grant GR/N23912/01) and DIRC (Interdisciplinary Research Collaboration in Dependability, grant GR/N13999/01) and by the European Union Framework Program 6 via the ReSIST Network of Excellence (Resilience for Survivability in Information Society Technologies, contract IST-4-026764-NOE).

References

Adams, E. N. (1984), "*Optimizing Preventive Service of Software Products*", IBM Journal of Research and Development 28(1), pp: 2-14.

Bishop, P. G. and R. E. Bloomfield (2003), "*Using a Log-normal Failure Rate Distribution for Worst Case Bound Reliability Prediction*", *in proc.* Int. Symp. on Software Reliability Engineering (ISSRE '03), Denver, Colorado, U.S.A., pp: 237-245.

Brocklehurst, S. and B. Littlewood (1996), "*Techniques for prediction analysis and recalibration*", *in* Handbook of Software Reliability Engineering, M. R. Lyu (Eds.), McGraw-Hill and IEEE Computer Society Press.

Fisher, R. A. (1922), "*On the interpretation of chi-squared from contingency tables, and the calculation of P*", Journal of the Royal Statistical Society 85(1), pp: 87-94.

Gashi, I. and P. Popov (2006), "*Rephrasing Rules for Off-The-Shelf SQL Database Servers*", *in proc.* 6th European Dependable Computing Conf. (EDCC-6), Coimbra, Portugal, IEEE Computer Society Press, pp: 139-148.

Gashi, I., P. Popov and L. Strigini (2004), "*Fault Diversity Among Off-The-Shelf SQL Database Servers*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '04), Florence, Italy, IEEE Computer Society Press, pp: 389-398.

Gashi, I., P. Popov and L. Strigini (2007), "*Fault tolerance via diversity for off-the-shelf products: a study with SQL database servers*", IEEE Transactions on Dependable and Secure Computing, to appear.

Gray, J. (1986), "*Why Do Computers Stop and What Can be Done About it?*" *in proc.* Int. Symp. on Reliability in Distributed Software and Database Systems (SRDSDS '86), Los Angeles, CA, USA, IEEE Computer Society Press, pp: 3-12.

Institute of Phonetic Sciences, A. (2006), "*Binomial proportions*", http://www.fon.hum.uva.nl/Service/Statistics/Binomial_proportions.html.

Littlewood, B. (1981), "*Stochastic Reliability Growth: a Model for Fault-Removal in Computer Programs and Hardware Designs*", IEEE Transactions on Reliability R-30(4), pp: 313-320.

Lyu, M. R., Ed. (1996), "*Handbook of Software Reliability Engineering*", McGraw-Hill and IEEE Computer Society Press.

Meulen, M. J. P. v. d., S. Riddle, L. Strigin and N. Jefferson (2005), "*On the Effectiveness of Run-Time Checks*", *in proc.* Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP '05), Fredrikstad, Norway, Springer-Verlag, pp: 151-164.

Popov, P., L. Strigini, S. Riddle and A. Romanovsky (2001), "*Protective Wrapping of OTS Components*", *in proc.* 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction, Toronto.

Preacher, K. J. and Briggs, N. E. (2001), "*Calculation for Fisher's Exact Test: An interactive calculation tool for Fisher's exact probability test for 2 x 2 tables [Computer software]*", http://www.quantpsy.org.

SourceForge (2006), "*Firebird downloads*", http://sourceforge.net/project/showfiles.php?group_id=9028.

Strigini, L. (2005), "*Fault Tolerance Against Design Faults*", *in* Dependable Computing Systems: Paradigms, Performance Issues, and Applications, H. Diab and A. Zomaya (Eds.), J. Wiley & Sons, pp: 213-241.

Appendix VI-2A –Likelihood equations of the extended Littlewood model

To produce a general likelihood function for data of the kind discussed in Section 3, several different kinds of data need to be considered. For each fault we need to consider:

- whether it is *present* in both DBMS products (i.e. does it *cause* a failure in both DBMS products),
- whether it is *randomly encountered* during testing of one or more DBMS products (i.e. has it been *reported* in more than one DBMS product).

We assume that all three classes of faults (faults that are *present* on DBMS product A only, B only, or both) have rates independently selected from *one common* gamma rate distribution. This produces a 5-parameter model with *parameters* being three unknown fault-count parameters (say N_A, N_B, N_{AB}), and two Γ-distribution parameters a, b.

For the *data* symbols, we will use a convention that observed failure *counts* n, and also observed *times* T of random failure *are all, likewise,* sub*scripted to denote which DBMS product(s)* contain *the faults*, and now, *in addition,* super*scripted* to identify the DBMS product(s) during the testing of which the fault is *randomly encountered.* In the case of Ts only, with multiple *super*scripts (AB or BA), the *first* of these superscripts will indicate which DBMS product's time T is. For all other cases (whether of *sub*scripts or *super*scripts AB) the order has no significance and will be left alphabetic. Finally, parameters l^A and l^B represent the *time* it took to uncover the faults in DBMS product A or B respectively. With these rather complex but necessary conventions, the extended Likelihood function for the Littlewood model is now given by:

$$p(n_{AB}^{A}, n_{AB}^{B}, n_{AB}^{AB}, T_{AB1}^{A} \ldots T_{ABn_{AB}^{A}}^{A}, T_{AB1}^{B} \ldots T_{ABn_{AB}^{B}}^{B},$$

$$T_{AB1}^{AB} \ldots T_{ABn_{AB}^{AB}}^{AB}, T_{AB1}^{BA} \ldots T_{ABn_{AB}^{BA}}^{BA}, n_{A}^{A}, T_{A1}^{A} \ldots T_{An_{A}^{A}}^{A}, \; n_{B}^{B}, T_{B1}^{B} \ldots T_{Bn_{B}^{B}}^{B}; N_A, N_B, N_{AB}, a, b) \;=$$

$$\frac{N_{AB}!}{(N_{AB} - n_{AB}^{A} - n_{AB}^{B} - n_{AB}^{AB})!} \times (a+1)^{n_{AB}^{AB}} \frac{a^{n_{AB}^{A} + n_{AB}^{B} + n_{AB}^{AB}}}{b^{n_{AB}^{A} + n_{AB}^{B} + 2n_{AB}^{AB}}} \times$$

$$\prod_{i=1}^{n_{AB}^{A}} \{(1 + \frac{T_{ABi}^{A} + l^{B}}{b})^{-(a+1)}\} \times$$

$$\prod_{j=1}^{n_{AB}^{B}} \{(1+\frac{T_{ABj}^{B}+l^{A}}{b})^{-(a+1)}\}\times$$

$$\prod_{k=1}^{n_{AB}^{AB}} \{(1+\frac{T_{ABk}^{AB}+T_{ABk}^{BA}}{b})^{-(a+2)}\}\times$$

$$(1+\frac{l^{A}+l^{B}}{b})^{-(N_{AB}-n_{AB}^{A}-n_{AB}^{B}-n_{AB}^{AB})a}\times$$

$$\frac{N_{A}!}{(N_{A}-n_{A}^{A})!}\times\frac{a^{n_{A}^{A}}}{b^{n_{A}^{A}}}\prod_{l=1}^{n_{A}^{A}} \{(1+\frac{T_{Al}^{A}}{b})^{-(a+1)}\}\times$$

$$(1+\frac{l^{A}}{b})^{-(N_{A}-n_{A}^{A})a}\}\times$$

$$\frac{N_{B}!}{(N_{B}-n_{B}^{B})!}\times\frac{a^{n_{B}^{B}}}{b^{n_{B}^{B}}}\prod_{m=1}^{n_{B}^{B}} \{(1+\frac{T_{Bm}^{B}}{b})^{-(a+1)}\}\times$$

$$(1+\frac{l^{B}}{b})^{-(N_{B}-n_{B}^{B})a} \qquad (27)$$

This is the general likelihood case. For our data sets to date (Gashi, Popov et al. 2004), (Gashi, Popov et al. 2007), there are no doubly *super*scripted *T*s (i.e. all the doubly *super*scripted *n*s are 0). That is to say, no common fault was *encountered randomly* during testing on *more than one* DBMS product (although we did note faults which were *present* in more than one DBMS product).

Paper-7. Uncertainty Explicit Assessment of Off-The-Shelf Software

Abstract: *Assessment of software COTS components is an essential part of component-based software development. Poorly chosen components may lead to solutions with low quality and difficult to maintain. The assessment is based on incomplete knowledge about the COTS component itself and other aspects, which may affect the choice such as the vendor's credentials, etc. We argue in favour of assessment methods in which uncertainty is explicitly represented ('uncertainty explicit' methods) using probability distributions. We provide details of a Bayesian model, which can be used to capture the uncertainties in the simultaneous assessment of two attributes, thus, also capturing the dependencies that might exist between them. We also provide empirical data from the use of this method for the assessment of off-the-shelf database servers which illustrate the advantages of 'uncertainty explicit' methods over conventional methods of COTS component assessment which assume that at the end of the assessment the values of the attributes become known with certainty.*

Co-authors: Dr. Peter Popov, Mr. Vladimir Stankovic
Journal: Elsevier Information and Software Technology Journal
Date of submission: Dec-2006
Status: *Under review*
Number of reviewers: TBC
Publication date: TBC

Full citation: TBC

1. Introduction

The use of commercial-off-the-shelf (COTS) components in software development is ubiquitous. There are many benefits to using COTS components stemming from the incentive to cut-down on cost and development time and to improve quality by using tried and tested components. An essential part of component-based software development is the assessment of available COTS components. Various assessment methods have been proposed (Ncube and Maiden 1999), (Kontio, Chen et al. 1995), (Jeanrenaud and Romanazzi 1994), (Tran and Liu 1997), (Ochs, Pfahl et al. 2001), (Alves and Castro 2001), (Phillips and Polen 2002), (Boehm, Port et al. 2003), (Dean 2000), (Kunda and Brooks 1999), (Gregor, Hutson et al. 2002), (Burgués, Estay et al. 2002), (Comella-Dorda, Dean et al. 2002), (Ruhe 2003). The results of these assessment techniques crucially depend on assuming that the values of the assessed attributes will be known with *certainty* at the end of the assessment. However, since the assessment is carried out with limited resources of time and budget the outcome is subject to *uncertainty*.

We propose an assessment method in which the assessment results are subject to explicitly stated uncertainty and discuss how this may impact the decisions about the use of COTS software. The method also enables representing the dependencies that exist between the uncertainties associated with the values of the COTS component attributes which affect the decision about which of the available COTS components to choose and also encourages assessing the dependent attributes simultaneously, thus speeding up the assessment. We provide empirical results from a study with off-the-shelf database servers, which demonstrate how the assessment method can be used in practice.

The paper is structured as follows: Section 2 contains an overview of the problems that need to be addressed during COTS component assessment; in Section 3 we describe models of assessment, in which model parameters (values of the attributes to be assessed) are not known with certainty and argue in favour of using probability distributions as an adequate mechanism to capture this uncertainty; in Section 4 we give details of an empirical study with off-the-shelf database servers and also some contrived numerical examples which illustrate the advantages of handling uncertainty and dependence between the values of the attributes; Section 5 contains a discussion of the scalability and applicability of the method proposed; Section 6 contains a brief review of related work on

COTS component assessment and attribute definitions and finally in Section 7 we present conclusions and possible further work.

2. Problems with COTS component assessment

2.1 Motivation

Any assessment is conducted with limited resources and under various assumptions, which may not hold true in real operation. As a result the outcome of the assessment is subject to uncertainty – the assessor may never be 100% sure that what they concluded during the assessment (both about the values of the attributes as well as the choice of a COTS component) will be confirmed when the COTS component is used in operation. This is clearly true for some parameters, which can be estimated *objectively*, e.g. failure rate, performance, etc. For failure rate, for instance, even after a very thorough testing one can only identify a range of rates which are more likely than others. For instance, Littlewood and Wright have shown (Littlewood and Wright 1997) that starting with indifference between the values of the failure rate (i.e. uniform distribution of the failure rate in the range [0, 1]) and seeing a protection system process correctly 4600 demands translates into 99% confidence that this system's probability of failure on demand (*pfd*) is no worse than 10^{-3}. The same equally applies to attributes assessed subjectively, e.g. using the Likert scale (Likert 1932), widely used in the COTS component assessment. It may be difficult for an assessor to justify that a COTS component must be ranked at exactly, say 7 out of 10, according to a chosen scale but he/she may be certain that the 'true' value of the attribute is in the range [6,7].

The value of expressing the assessment results in the form (value, confidence) has been recognized in some other technical areas which dealt with assessment. The best performing software reliability-growth models (RGM) which predict the failure rate from the observed failures in the past, for instance, are those in which the model parameters are treated as *random variables* (Brocklehurst, Chan et al. 1990). In these models the 'true' values of the attributes being assessed are never assumed known with certainty. Instead the attribute is characterized by a probability distribution from which the true value of the attributes will come (i.e. are seen as drawn at random). For each reliability target, then,

the assessor can tell the probability that the true reliability is lower than the target. Such models *systematically outperformed* the alternative simplistic methods in which the parameters were assumed known with certainty (Lyu 1996). If the 'uncertainty explicit' models have been best with one specific method of assessment – software reliability – it seems natural to try similar 'uncertainty explicit' methods for other assessments, e.g. evaluation of COTS software and selecting the best out of a set of comparable alternatives. This is the focus of this paper.

There are various methods for representing uncertainty (Wright and Cai 1994). Bayesian approach to probabilistic modelling is one of the best-known ones and used with some success in reliability assessment (Lyu 1996), (Littlewood and Wright 1997). It allows one to combine, in a mathematically sound way, the prior belief (which may be 'subjective' and possibly inaccurate) about the values of a parameter or a set of parameters to be assessed with the ('objective') evidence from seeing the modelled artefact in operation. Combining the prior belief and the evidence from the observations in a mathematically correct way leads to a posterior belief about the values of the assessed attribute(s).

How does 'uncertainty explicit' assessment differ from the conventional deterministic assessment? With deterministic assessment point estimates of the attributes are used. A common approach of comparing the alternatives is then to use a *weighted sum* of the estimates for each of the alternatives. When uncertainty is used, this approach is still possible – we can use various characteristics of the posterior distributions (mean, median, etc.) of the attributes as estimates and then calculate the weighted sum for each of the COTS components included in the assessment before deciding which is the best one. When uncertainty is explicitly used in the assessment, however, more refined ways of comparison are possible: from the posterior one can express *the uncertainty in the value of the comparison criterion*, e.g. the weighted sum of the attributes. Since the value of the weighted sum is now uncertain we have a range of options. We may give preference to the COTS component for which the mean (median) value of the weighted sum is best (as we would have done with point estimates of the attributes). With uncertainty stated explicitly a range of new options exists, which is illustrated in Fig. 15.

In this figure we illustrate the value of handling uncertainty explicitly even when dealing with a single assessment attribute, COTS component reliability. Let us assume that Fig.

15 illustrates the cumulative distribution function (c.d.f.) graph for two COTS components with the same average *pfd*. If we wanted to choose the COTS component that has the highest probability of having a *pfd* of no worse than 6.10^{-3} (i.e. the value of the x-axis of 0.006) then we would choose COTS component A, whereas the COTS component with the highest probability of having a *pfd* of no worse than 4.10^{-3} is COTS component B. We can also see clearly that the distribution of the *pfd* of COTS component B is much more spread than that of COTS component A (in fact the distribution of COTS component A is uniformly spread across all the values from 0 to 10^{-2}). Therefore there is a much higher uncertainty associated with the values of COTS component B than those of COTS component A. Stating uncertainty explicitly, thus, offers the assessor a wider range of options in selecting the most appropriate COTS component.

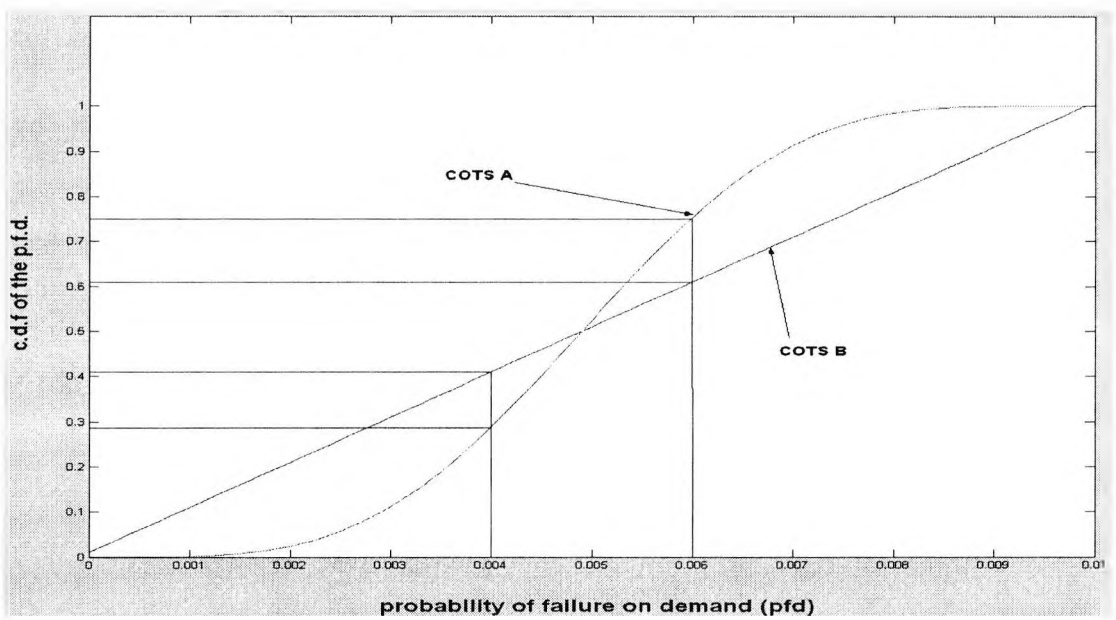

Fig. 15 - The pfd for two different COTS components.

2.2 Dependence among attributes

COTS component assessment requires dealing with multiple attributes of the COTS components being compared. The selection of a particular COTS component, thus, is a multi-criteria decision which taken under uncertain values of the attributes naturally leads to the question about the dependence between the uncertainties associated with the individual attributes. Ignoring the possible dependence between the attributes represents a particular form of belief: that assessing attribute X one can learn *nothing* about another

attribute, Y. For example, performance of a COTS component will hardly tell anything about the quality of its documentation and vice versa. It is quite obvious, however, that not all COTS component attributes are like that. In many cases while assessing an attribute X the assessor may infer something about another set of attributes. For instance if we devise a prototype in order to assess the functionality of a COTS component in the process we will learn something about the performance (how quickly this COTS component responds to requests) and how reliable the COTS component is. A more subtle, but very useful concept, as we will see later, is that the uncertainties associated with the assessed attributes may be *dependent*. Informally, assume that we want to assess the reliability and performance of a COTS component. We may assume that the uncertainties associated with these two attributes are independent, in the statistical sense. Under this assumption learning something about reliability will tell us nothing about performance and vice versa. Now suppose that we have run a very long testing campaign and have repeatedly observed that whenever the response was late it was also incorrect and no other incorrect response has been observed. With such evidence of a strong positive correlation between the failures (incorrect responses) and the responses being late, we may accept that any change of our belief about the rate of failure should also be translated into a change in our belief about the rate of late responses. The assessment models surveyed invariably assume that the attributes are independent and do not allow for dependencies between their uncertain values to be captured adequately.

In summary, we can draw a two by two contingency table to illustrate a possible categorization of an assessment method (Table 27) with respect to the method's handling of the uncertainty between the values and the dependence between the values of the attributes.

In the assessment method that we propose in this paper we are in quadrant I: we aim to both handle the uncertainties in the values of an attribute and the dependence that exists between the values of the different attributes. The existing assessment methods surveyed tend to be in quadrant IV.

Table 27 – A categorization of an assessment method with respect to *uncertainty* and the *dependence* of the attribute values

		Is the *Uncertainty* handled?	
		Yes	No
Is the *Dependence* handled?	Yes	I	II
	No	III	IV

3. Assessment of COTS components: Bayesian approach

In this section we briefly summarize how the Bayesian approach to assessment is normally applied to assessment of a single attribute. Assume that the attribute of interest is the component's probability of failure on demand (*pfd*). If the system is treated as a black box, i.e. we can only distinguish between *COTS component's* failures or successes (Fig. 16), the Bayesian assessment proceeds as follows.

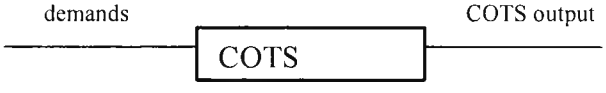

Fig. 16 - Black-box model of a COTS component. The internal structure of the component is unknown. Only its output (success or failure) is recorded on each demand and used in the inference of component's pfd

Let us denote the system *pfd* as p, with prior distribution (probability density function, *pdf*) $f_p(\bullet)$, which characterises the assessor's knowledge about the COTS component *pfd prior* to observing the COTS component in operation. Assume further that the COTS component is subjected to n demands, independently drawn from a 'realistic' operational environment (profile)[43], and r failures are observed. The posterior distribution, $f_p(x \mid r,n)$, of p after the observations will be:

$$f_p(x \mid r,n) \propto L(n,r \mid x) f_p(x), \tag{28}$$

where $L(n,r \mid x)$ is the *likelihood* of observing r failures in n demands *if* the *pfd* were exactly x, which in this case of independent demands is given by the *binomial* distribution, $L(n,r \mid x) = \binom{n}{r} x^r (1-x)^{n-r}$. For any prior and any observation (r, n) the posterior can be calculated for all the COTS components included in the assessment.

[43] An operational profile (Musa 1993) can be defined as a quantitative characterization of how the component will be used in its 'true' environment

Even if no failure is observed (i.e. $r = 0$), the posterior can be calculated. Other measures of interest can also be derived from this posterior, e.g. the probability that the COTS component will survive the next 5000 randomly chosen demands. This probability can be calculated for each of the COTS components included in the assessment as follows:

$$\int_0^\infty (1-p)^{5000} f_p(p \mid r,n) dp$$

Then the best COTS component will be the one, for which the integral above gets a maximum value.

3.1 Model for assessment of 2 non-independent attributes

Typically the COTS component assessment is a multi-criteria decision with dozens of attributes usually assessed and taken into account (as detailed in (Kontio, Chen et al. 1995), (Ncube and Maiden 1999), (Ochs, Pfahl et al. 2001), (Boehm, Port et al. 2003)). The Bayesian assessment can be applied to multiple attributes, too. For simplicity we first demonstrate the approach with two attributes and then discuss the implications of scaling it up to many attributes.

Let us assume that two non-functional attributes must be assessed, such as the COTS component's *pfd* and performance, the latter assessed in the form of whether the response is received on time or not, i.e. the probability of a *late response* on demand, *pld*. Using a binary score – on time vs. late – is an adequate approach when the COTS component is planned for integration in a larger system. In these circumstances using an absolute scale, e.g. how long it takes a COTS component to respond to a demand, may be unnecessary: it will be sufficient to know whether the response is received with an acceptable delay as dictated by the wider system. In terms of comparison of several COTS components using the binary scale (on time/late) seems also adequate. Any COTS component, which responses with an acceptable delay, is equally good from the point of view of the system's integrator.

Here we define a model to help with the comparison of COTS components assessed by subjecting them to a *series of independently selected demands*. Both, the COTS component's *pfd* and *pld*, are used in the comparison and different comparison criteria are discussed.

On each demand the response received from the COTS components is evaluated from two different viewpoints: correct/incorrect and on time/late. Clearly 4 combinations exist, which can be observed on a randomly chosen demand, as shown in Table 28.

Table 28 – The outcomes, their frequencies and probabilities for a random demand

Event	Correct Response (Reliability)	Response On-Time (Performance)	Number of observations in n demands	Probability
α	No	Yes	r_1	p_{10}
β	Yes	No	r_2	p_{01}
χ	No	No	r_3	p_{11}
δ	Yes	Yes	r_4	p_{00}

The four probabilities given in the last column sum to 1. So if the last three probabilities are 0.2, 0.4 and 0.3, respectively, then the first one p_{10} = 1- (0.2 + 0.4 + 0.3) = 0.1. This constraint remains even if we treat the probabilities in Table 28 as random variables: their sum will always be 1. Thus, the joint distribution of any three of these probabilities, e.g. $f_{p_{01},p_{10},p_{11}}(\bullet,\bullet,\bullet)$, gives an exhaustive description of the COTS component's behaviour. In statistical terms, the model of the COTS component with two binary attributes has three degrees of freedom.

The marginal probabilities of getting an incorrect response on a random demand, let's denote it p_I, and of getting the response late, p_L, respectively, can be expressed as:

$p_I = p_{10} + p_{11}$ and $p_L = p_{01} + p_{11}$.

p_{11} represents the probability of receiving late an incorrect response and, hence, a notation $p_{IL} \equiv p_{11}$ will capture better the intuitive meaning of the event it is assigned to. Instead of using $f_{p_{10},p_{01},p_{11}}(\bullet,\bullet,\bullet)$ another distribution, which can be derived from it through a functional transformation, can be used. In this section we use $f_{p_I,p_L,p_{IL}}(\bullet,\bullet,\bullet)$.

It can be shown that for a given observation (r_1, r_2, and r_3 in N demands) the posterior joint distribution can be calculated as:

$$f_{p_I,p_L,p_{IL}}(x,y,z \mid N,r_1,r_2,r_3) = \frac{f_{p_I,p_L,p_{IL}}(x,y,z)L(N,r_1,r_2,r_3 \mid p_I,p_L,p_{IL})}{\iiint_{p_I,p_L,p_{IL}} f_{p_I,p_L,p_{IL}}(x,y,z)L(N,r_1,r_2,r_3 \mid p_I,p_L,p_{IL})dxdydz} \quad (29)$$

where

$$L(N, r_1, r_2, r_3 \mid p_1, p_L, p_{IL}) = \frac{N!}{r_1! r_2! r_3! (N - r_1 - r_2 - r_3)!} (p_1 - p_{IL})^{r_1} (p_L - p_{IL})^{r_2} p_{IL}^{r_3} (1 + p_{IL} - p_1 - p_L)^{N - r_1 - r_2 - r_3} \quad (30)$$

is the multinomial likelihood of the observation (r_1, r_2, r_3 , N).

A similar model has been used in the past in assessing reliability of various systems built with components (Littlewood, Popov et al. 2000), (Popov 2002).

3.2 Combination of uncertainties in the values of attributes

For comparison of the COTS components we will define the following criterion:

> Probability of an *inadequate response*, P_{Ser}, by the COTS component: of getting either an incorrect or late response. Clearly, $P_{Ser} = P_1 + P_L - P_{IL}$. Its posterior distribution, $f_{p_{Ser}}(\bullet \mid N, r_1, r_2, r_3)$, can be derived from the joint posterior, $f_{p_1, p_L, p_{IL}}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$, by first transforming it, to for example $f_{p_1, p_L, p_{Ser}}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$, and then integrating out the nuisance parameters P_1 and P_L.

An often used selection method (Port and Chen 2004) in the literature is the weighted sum of the values of the attributes. The weighted sum of the two attributes in our study can be calculated as follows: $P_S = kP_1 + (1\text{-}k)P_L$, in which the constant k is defined by the assessor. High values of k correspond to cases when incorrect results are highly undesirable while late results may be tolerable. On the contrary, low values of k correspond to cases when incorrect results may be tolerated by the system while late responses may have serious consequences. In order to derive the marginal distribution of P_S first the joint distribution $f_{p_1, p_L, p_{IL}}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$ is transformed to $f_{p_1, p_L, p_S}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$ and then the nuisance parameters P_1 and P_L are integrated out, as we did above for P_{Ser}. However we will not be using this method of selection since the new variable P_S does not have an obvious intuitive meaning. This difficulty is compounded in our case since the uncertainty is stated explicitly. It is impossible to say what a confidence of say 99% associated with a particular value of P_S tells us about the COTS component being assessed.

3.3 Partitioning the demand space

In some areas of software engineering, especially in testing, the usefulness of *partitioning the demand space* has been recognised (Jeng and Weyuker 1991), (Hamlet and Taylor 1990), (Musa 1993). The demand space partitions typically represent *different types of demands*, which may have different likelihoods of occurring in realistic environment. Realistic testing, thus, would require generating mixes of demands, which take into account the likelihood of the types of demands.

In our context, operating in a partitioned demand space may imply that the uncertainty associated with the attributes of interest may differ among the partitions, e.g. as a result of different number of observations being made for the different partitions.

If the demand space is partitioned into M partitions $\{S_1, S_2, \ldots S_M\}$, then for each of these the assessment will be performed as described above, e.g. with two attributes the description provided in Section 3.1 will apply. As a result M *conditional distributions* will be associated with each COTS component, e.g. using reliability and performance these can be denoted as $f_{p_I, p_I, p_{II}}(\bullet,\bullet,\bullet \mid S_i)$, from which the conditional distribution $f_{p_{Ser}}(\bullet \mid S_i)$ will be expressed. This distribution characterises the probability of failure (incorrect or late response), $P_{Ser} \mid S_i$, of the particular COTS component in the specific partition. Finally, in order to compare the competing COTS components the *unconditional distribution* $f_{p_{Ser}}(\bullet)$ should be derived for the particular profile defined over the set of partitions, which represents the targeted operational environment.

Let us denote the profile of the targeted environment as $\{P(S_1),\ldots,P(S_M)\}$[44], and assume that these are *known with certainty*[45]. The marginal probability of failure of a COTS component, according to the formula of full probability is:

$$P_{Ser} = \sum_{i=1}^{M} P_{Ser} \mid S_i \times P(S_i) \qquad (31)$$

[44] The meaning of these random variables is that a demand chosen at random with probability $P(S_i)$ will be drawn from S_i.

[45] This assumption is needed for the comparison only. We do not require here that we know the 'real' operational environment, in which the system together with the chosen COTS component will be deployed. Taking into account the uncertainty about the profile is possible at the expense of making the calculations more complicated, which is beyond the scope of this paper.

The distribution of this random variable, P_{Ser}, depends on the joint distribution, $f_{P_{Ser}|S_1,\ldots,P_{Ser}|S_M}(\bullet,\ldots,\bullet)$, i.e. of the conditional probabilities of failure in sub-domains. In some setups it may be plausible to assume that the conditional probabilities of failure (in the partitions that is) are *independently distributed*, i.e.:

$$f_{P_{Ser}|S_1,\ldots,P_{Ser}|S_M}(\bullet,\ldots,\bullet) = \prod_{i=1}^{M} f_{P_{Ser}|S_1}(\bullet)\ldots f_{P_{Ser}|S_M}(\bullet). \quad (32)$$

Such an assumption represents the assessor's belief that learning something about the probability of failure, $P_{Ser} \mid S_i$, of a particular COTS component in partition i will not change their belief about the probability of failure, $P_{Ser} \mid S_j$, of the same COTS component in another partition. The assumption is consistent with applying inferences to the individual partitions, i.e. conditional on the demands coming from a particular partition.

Under (32) the unconditional probability of COTS component failure (31) can be expressed as a *convolution* of the distributions of the random variables $P_w(i) = P_{Ser} \mid S_i \times P(S_i)$, i.e.:

$$P_{Ser}^{w} = \otimes P_w(i) \quad (33)$$

The selection of the best COTS component, out of the available alternatives, then will be based on the marginal distributions, $f_{P_{Ser}^{w}}(\bullet)$, associated with the available COTS components.

4. Numerical examples: a study with off-the-shelf database servers

We have reported recently results of studies on dependability and performance of database servers (Gashi, Popov et al. 2004b) (***the preceding reference forms part of this thesis as Paper-1***), (Gashi, Popov et al. 2004a) (***the preceding reference forms part of this thesis as Paper-3***), (Stankovic and Popov 2006). The focus of these earlier studies was in measuring the amount of "diversity", in both correctness and response time, which exists between different servers, i.e. certain server might give an incorrect and/or late response in one input but the other one might not. The motivation behind this work was

to get preliminary measurements on the improvements in reliability and performance that can be had from using more than one component in parallel in a multi-channel diverse configuration.

In this paper we will use the data collected in those studies to demonstrate our approach to COTS component selection. SQL servers are a very complex category of off-the-shelf components, with many reported faults in each release. In total six off-the shelf SQL servers from four different vendors were used. Four of the servers are open-source, namely PostgreSQL 7.0, PostgreSQL 7.2, Interbase 6.0 and Firebird[46] 1.0. The other two servers are commercial closed development servers, anonymised here due to the restrictive 'End User License Agreements'. We will refer to these components as CS1 (Commercial Server 1) and CS2 as they are from different vendors.

An ideal selection of an SQL server based on the results of statistical testing of the COTS components may be problematic in practice. We will highlight two circumstances in which these difficulties can occur:

- Assume that we are interested in choosing between several SQL servers, based on their reliability and performance. The ideal situation for choosing the most appropriate SQL server based on measurements *after* deploying the COTS components is clearly *unrealistic* since we would like to select the best server *before* the application is developed.
- Assume that the system integrator (e.g. a software house) would like to make a *strategic* choice of a SQL server for use in the foreseeable future. In this scenario the application(s), which may be developed in the future may be even unknown at the time of making the selection.

Given these difficulties we can use alternative options:

- Using well-known benchmark applications. In the context of SQL servers this might be the TPC-C benchmark for on-line transaction processing (TPC 2002). In this case, the performance of the components can be measured directly on the target platform, but there might be problems observing failures. This is because it would be reasonable to expect that an SQL server would correctly process the statements defined in the TPC-C benchmark application. Thus, in this case the

[46] Firebird is the open-source descendant of Interbase. The later releases of Interbase are issued as closed-development by Borland.

selection of the SQL server would be significantly influenced by the performance attribute. Even if failures are observed, such a measurement of the reliability of the COTS components may be very expensive; the likely candidates to choose from will be reliable components. Thus the amount of testing to observe a few failures may be prohibitively high (Adams 1984). We illustrate the assessment method with data collected from experiments with an implementation of the TPC-C client benchmark. For the TPC-C experiments we used all six of afore mentioned SQL servers.

- Using *stressful environments* (in terms of the reliability attribute) may be sought in such circumstances for comparing the components, i.e. environments which increase the likelihood of failures occurring, even if we do not know how likely these are to occur in operation. The set of bugs of a particular COTS component (in our case SQL server) defines one such stressful environment for a server. The union of the bugs reported for all the compared COTS components will form a demand space, in which there will be a partition stressing each of the components. We have collected known bug reports for four of the SQL servers in our studies, namely PostgreSQL 7.0, Interbase 6.0, CS1 and CS2 and used them as a sample from a 'stressful' environment, in which to compare the COTS components.

Detailed results for each of these studies are given in the next two sub-sections. We did not use *partitioning of the demand space* approach in the study with the TPC-C Benchmark application (even though the TPC-C transactions types could form basis for such partitioning). This is because we did not have any reason to expect that the servers will perform differently (in terms of timeliness and correctness) for each transaction type. We however did use partitioning of the demand space in the study with the bug reports of the servers, since we had compelling reasons to expect that the servers will perform differently (this will be explained in Section 4.2).

4.1 Study with the TPC-C benchmark application

We first describe the results obtained using the TPC-C benchmark application as a basis of selecting the best SQL server. In the empirical study we used our own implementation of TPC-C. The benchmark defines five transaction types (New-Order, Payment, Order-

Status, Delivery and Stock-Level) and sets the probability of execution for each, i.e. the particular transaction mix (profile) is defined. The specified performance measure is the number of *New-Order* transactions completed per minute. However, our measurements were more detailed - we recorded the transaction response times instead. The benchmark specifies explicitly an upper bound on the 90th percentile values for each transaction type. It requires that a response time of each transaction type is less than or equal to the respective 90th percentile value. The values are as follows:

- New-Order - 5 seconds
- Payment - 5 seconds
- Delivery – 80 seconds
- Order-Status – 5 seconds
- Stock-Level – 20 seconds

The test harness consisted of two machines:

- a server machine, on which one of the six database servers was run.
- a client machine, which executed a JAVA implementation of the TPC-C standard

Each experiment comprised the *same sequence* of 1000 transactions. We ran two types of experiments:

- *single client* - a TPC-C compliant client modifies the database by executing the specified transaction mix
- *multiple clients* - a TPC-C compliant client modifies the database and additional 10 clients concurrently execute read-only transactions (Order-Status and Stock-Level).

Multiple clients experiment enabled us to increase the load on the servers and measure the effect of the increased load on their performance.

A timeout value, specific to each transaction type, was used to distinguish between late and timely responses. We defined two sets of timeouts[47]:

- the 90th percentile values specified by TPC-C (*TPC-C timeout*),
- one fifth of the 90th percentile values (*short* timeout).

We defined four scenarios, varying the number of clients and timeout values respectively:

[47] The choice of these was made after a personal communication of one of the authors with a TPC-C affiliate and auditor who confirmed that the values were conservative for a wide range of on-line transaction processing applications.

- Scenario 1 - single client / TPC-C timeouts
- Scenario 2 - single client / short timeouts
- Scenario 3 - multiple clients / TPC-C timeouts
- Scenario 4 - multiple clients / short timeouts

The SQL servers were compared for each of the scenarios.

4.1.1 Prior distributions

The prior, $f_{p_I, p_L, p_{IL}}(\bullet,\bullet,\bullet)$, was constructed under the assumption that P_I and P_L are independently distributed random variables, i.e. $f_{p_I, p_L}(\bullet,\bullet) = f_{p_I}(\bullet) f_{p_L}(\bullet)$. We made this assumption since we did not have any objective evidence to believe otherwise. In case there are reasons (objective or subjective) then the assumption of independence maybe be dropped. In this case the particular form of $f_{p_I, p_L}(\bullet,\bullet)$ should be defined explicitly. Additionally the conditional distributions $f_{p_{IL}|P_I, P_L}(\bullet \mid P_I, P_L)$ were defined for every pair of values of P_I and P_L, in the range [0, min(P_I, P_L)] since the probability of incorrect and late responses cannot be greater than the probability of *either* of the two individually. In passing we note that the choice of the prior is not critical here since with this benchmark application an arbitrarily large number of demands can be generated, which can correct any inaccuracies of the priors, i.e. 'the data will speak for itself'.

We anticipated observing mainly late responses while the incorrect result failures were expected to be very rare. We have assumed 'ignorance prior' (Uniform distribution) for performance in the range 0 to 1. For incorrect result failures we have also assumed ignorance but using an upper bound of 10^{-2}, likely to be very conservative in the context of TPC-C. We assumed ignorance priors for both P_I and P_L since we did not have any preference regarding their values. In this study we used the same distribution for all the servers since for the scenarios tested we did not have any reason to prefer one server over the others. There might, however, be cases – some discussed later in Section 5.4 - whereby the assessor may have different prior beliefs about the competing COTS components.

A summary of the distributions used and the range in which they are defined is given in Table 29.

Table 29 - The Prior distributions (identical for all six servers and all four scenarios)

Prior Distribution	Range	Distribution Type
Reliability $f_{p_I}(\bullet)$	0 – 0.01	Uniform
Performance $f_{p_L}(\bullet)$	0 – 1	Uniform
Conditional distribution: $f_{p_{IL}\|p_I,p_L}(\bullet \mid P_I, P_L)$	0 – min(P_I,P_L)	Uniform

4.1.2 Observations

The observations from the TPC-C experiments are given in Table 30. The number of demands for all servers is 1000. Five out of six servers exhibit late result failures only. Incorrect result failures are observed only for CS2. In addition, whenever a result was incorrect on CS2 it was late, too. The incorrect results observed were due to the specific concurrency control mechanism used by CS2 (Popov, Strigini et al. 2004). The locks on resources, e.g. database rows, were not released properly when the lock holding transactions were completed. To resolve the problem we had to install timeout watchdogs and abort transactions when the timeout expired. Each aborted transaction was repeated as many times as necessary to eventually commit successfully. We decided to use *transaction repetition count* as the criterion of an incorrect response on CS2. In particular, we defined a threshold of 5 as a value, beyond which the transaction would be considered to have failed.

We used transaction timeout values and *transaction repetition count* to classify each demand on each server in the categories r_1 to r_4 (defined in Section 3.1).

Table 30 - The observations of the six database servers for the four scenarios. The number of demands (N) is 1000 for each server. We did not observe any incorrect-only failures, i.e. r_1=0 for all servers

COTS	Scenario 1			Scenario 2			Scenario 3			Scenario 4		
	r_1	r_2	r_3	r_1	r_2	r_3	r_1	r_2	r_3	r_1	r_2	r_3
PG 7.0	0	1	0	0	30	0	0	0	0	0	644	0
PG 7.2	0	6	0	0	33	0	0	3	0	0	489	0
IB 6.0	0	0	0	0	24	0	0	1	0	0	434	0
FB 1.0	0	0	0	0	1	0	0	0	0	0	439	0
CS1	0	0	0	0	33	0	0	19	0	0	303	0
CS2	0	0	0	0	4	0	0	0	1	0	329	1

4.1.3 Posteriors

The percentiles derived from the posterior distribution for the 4 scenarios are given in Table 31. One can see that the ordering between the servers changes as the number of clients and/or the timeout values vary (to improve the readability of the table we have explicitly shown the ranking order of the servers in each scenario).

Under Scenario 1 (the least demanding scenario) four servers (IB 6.0, FB 1.0, CS1 and CS2) produce identical results since they completed without any failure (i.e. on time and correctly) the 1000 transactions. We are indifferent in the choice among them. The two versions of PostgreSQL exhibit late responses and they are ranked lowest. When we decrease the timeout value (Scenario 2) the ranking changes: now there are late responses with all the servers. The two worst servers are still PostgreSQL 7.2 and CS1. Interestingly, Firebird 1.0, an open-source server, is ranked the best.

In Scenario 3 the percentile values are close again as in the first scenario, though the earlier version of PostgreSQL, PG 7.0, is ranked the best, alongside Firebird 1.0 while CS1 is the worst performing server. Firebird 1.0 is consistently among the best servers in the first 3 scenarios. An interesting observation is the 50^{th} percentile value of the posteriors CS2 and IB 6.0. Even though the total number of failures for these two servers were the same (1 each, see Table 30), the nature of the failure was different: the result from CS2 was both incorrect and late whereas from IB 6.0 it was only late. Exploring this dependence we can still see a difference in the 50^{th} percentile values of these two servers (even though the difference is marginal and on the chosen accuracy of expressing the percentile values is not observed in the 99^{th} percentile). We will further scrutinize the interplay between the failures of the individual components and the correlation between their failures with contrived examples in Section 4.4.

The ranking changes again in the most demanding scenario (Scenario 4). The best server is now CS1.

Table 31 – Percentiles (abbreviated P-tile) for the distribution of the system quality $P_{Ser} = P_1 + P_2 - P_{12}$ classified per scenario. To improve the readability we have also provided the Ranking order for each of the servers based on the percentiles values. The prior distribution is the same for all servers across all scenarios

P-tile	COTS	Prior	Scenario 1		Scenario 2		Scenario 3		Scenario 4	
			Posterior	Rank	Posterior	Rank	Posterior	Rank	Posterior	Rank
0.5	PG 7.0	0.502	0.0021	5	0.0310	4	0.0012	1	0.6436	6
	PG 7.2		0.0071	6	0.0340	5	0.0041	5	0.4888	5
	IB 6.0		0.0012	1	0.0250	3	0.0021	4	0.4340	3
	FB 1.0		0.0012	1	0.0021	1	0.0012	1	0.4392	4
	CS1		0.0012	1	0.0340	5	0.0200	6	0.3032	1
	CS2		0.0012	1	0.0051	2	0.0020	3	0.3300	2
0.99	PG 7.0	0.992	0.0076	5	0.0456	4	0.0060	1	0.6780	6
	PG 7.2		0.0152	6	0.0492	5	0.0108	5	0.5256	5
	IB 6.0		0.0060	1	0.0384	3	0.0076	3	0.4704	3
	FB 1.0		0.0060	1	0.0076	1	0.0060	1	0.4756	4
	CS1		0.0060	1	0.0492	5	0.0324	6	0.3376	1
	CS2		0.0060	1	0.0124	2	0.0076	3	0.3652	2

4.2 Study with the known bugs of the servers

Now we compare the servers using the methodology described in Section 3.3. We have collected known bug reports for four SQL servers. We will use the union of the bugs reported for each of these SQL servers. Each of these bug reports will constitute a 'demand' to the server. These demands form a partition of the demand space for each server[48]. In contrast to the TPC-C study where partitioning of the demand space was not used, in the study with the bug reports we apply inferences to the partitions. The reason for doing so was due to the very different prior beliefs about the performance of servers in the different partitions as will be discussed in Section 4.2.1. The logs of the *known bugs* are treated as a sample (without replacement[49]) from the corresponding partition (representing the server, for which the bug has been reported). We label the

[48] We have observed no bugs reported for two or more servers, thus the logs of the known bugs indeed formed partitions of the union of the bugs. Even if we had observed bugs reported form more than one server we could construct a partition of the intersection of the bugs reported for several servers by putting them in their own partition. Thus a server may have more than one own partition in the demand space and the description provided here will apply.

[49] Strictly, there might be a difference between sampling with and without replacement. Our model is based on sampling without replacement while the inference procedure described in Section 3.1 implies sampling with replacement. This is a simplification, which in many cases is acceptable (e.g. sampling from a large population of units, none of which dominates the sampling process, which seems a plausible assumption in our case of SQL servers being very complex products and likely to contain many unknown bugs).

partitions $S_{Server\ name}$. Partition S_X is called an 'own' partition for server X and a 'foreign' partition for any other server Y≠X.

4.2.1 Prior Distributions

The prior distributions $f_{p_I,p_L,p_{IL}}(\bullet,\bullet,\bullet \mid S_i)$ used in this study are explained next. The prior distribution, $f_{p_I,p_L,p_{IL}}(\bullet,\bullet,\bullet \mid S_i)$, was constructed under similar assumptions to those of the TPC-C study: that P_I and P_L are independently distributed random variables; in the general case of incorrect and late responses being non-independent events, the conditional distributions, $f_{p_{IL}|P_I,P_L}(\bullet \mid S_i, P_I, P_L)$, are specified for every pair of values of P_I and P_L.

The distributions were assumed to be identical for each of the four servers in both their 'own' and 'foreign' partitions. Again, this assumption was made because we did not have objective evidence to believe otherwise. We discuss other options in Section 5.4. A summary of the distributions used and their respective parameters including the range of each distribution are given in Table 32, and we will discuss these choices in the rest of this sub-section.

Table 32 – The Prior distributions (identical for all four severs)

	Proportion	Range	Distribution
Reliability	$f_{p_I}(\bullet \mid S_{own})$	0.72 – 1	Uniform
	$f_{p_I}(\bullet \mid S_{foreign})$	0 – 1	Uniform
Performance	$f_{p_L}(\bullet \mid S_{own})$	0 – 1	Uniform
	$f_{p_L}(\bullet \mid S_{foreign})$	0 – 1	Uniform
Conditional distribution: $f_{p_{IL}\mid p_I,p_L}(\bullet \mid S, P_I, P_L)$		0 – min(P_I,P_L)	Uniform
		0 – min(P_I,P_L)	Uniform

Prior distributions for Incorrect Results $f_{p_I}(\bullet \mid S_i)$

For 'own' partitions the prior distribution was defined as Uniform in the range [L, 1], where L < 1 accounts for the chance that some of the reported bugs might be

Heisenbugs[50], i.e. we expect most of the bugs that have been reported for a particular server to cause failures when they are run on that server (hence the probability of observing an incorrect results failure is very close to 1) but, due to Heisenbugs, not always so. As a source for L we used the study by Chandra and Chen (Chandra and Chen 2000). These authors studied the fault reports for three off-the-shelf components: MySQL database server, GNOME desktop environment and the Apache web-server and reported that 5%, 7% and 14%, respectively, of the reported bugs were Heisenbugs. Given the variation between the components we pessimistically interpreted these findings by setting L = 1-(2*0.14), that is twice the highest value of Heisenbugs reported, thus the prior is expected to be within the range [0.72, 1]. Notice that here the prior distribution for incorrect results is being defined at a range close to 1 (i.e. high unreliability). This is because of the unusual profile of the demands: since we are using known bug reports as demands we expect most of the bugs to cause failures when we run them on the server for which they were reported.

For 'foreign' partitions, however, the prior distributions were defined as uniform in the range [0, 1]. This is due to the absence of any comparative study to guide our expectation about the likely value. In passing we note that theoretical work such as (Littlewood and Miller 1989), (Eckhardt and Lee 1985) suggest that diverse software versions will tend to fail coincidentally on 'difficult' demands. Since all the bugs are 'difficult' – they are known to be problematic at least for one of the servers – we may consider them genuinely difficult, hence assume as plausible that the other servers too, are likely to fail. On the other hand, empirical studies such as (Knight and Leveson 1986), (Eckhardt, Caglayan et al. 1991), have shown that significant gains can be had via design diversity – hence low chances that a particular server will fail on bugs reported for other server are also plausible. In summary, we are indifferent between the values of the probability that a server will fail from a 'foreign' bug.

[50] Gray defines two types of bugs (Gray 1987): "Bohrbugs" for bugs that appear to be deterministic (they manifest themselves each time the bug script is executed); and "Heisenbugs" for those that are difficult to reproduce as they only cause failures under special conditions (e.g., created by the internal state affected by the other applications etc.)

Prior distributions for Performance $f_{p_L}(\bullet \mid S_i)$

We have not found a public domain sources, which would allow us to define a prior distribution for performance failures (in the context we have defined). This is also because the number of late results that would be observed would be conditional on how the timeout threshold is set. The only remaining source is to look at the data (either our own or of various vendors) from the experiments using the TPC-C (TPC 2002) benchmark. However it is not clear how accurate a prior based on these results would be due to the differences in the profile that will exist between the TPC-C client application and the bug scripts. Therefore we have decided to define the prior distribution for all proportions as uniformly distributed in the range 0 to 1, i.e. be 'indifferent' between the possible chances of the servers exceeding the set timeout.

Prior distributions for Incorrect and Late Results $f_{p_{IL}|P_I,P_L}(\bullet \mid S_i, P_I, P_L)$

All conditional prior distributions of the probability of a result being at the same time incorrect and late were defined in the range [0, $min(P_I, P_L)$] (since the probability of incorrect and late responses cannot be greater than the probability of *either* of the two individually). This is again due to the rather unique profile, under which we apply the inference and the lack of comparable studies that would enable us to define a more accurate prior, thus 'indifference'.

Priors for probabilities of a bug being selected from the partitions

For the comparison of the servers we use a distribution defined on the set of partitions, which does not favour any of the servers, i.e. we assumed that probability of each partition is 0.25 in the study with 4 servers[51].

4.2.2 Observations

The observations using the known bugs of four off-the-shelf servers are given in Table 33. We can see that the number of bugs collected for each server was different, which indicated that the empirical evidence differs between the partitions. The reasons for this were merely differences in the reporting practices operated by the vendors of the servers,

[51] We could use the number of known bugs for each of the partition to construct a profile consistent with the observations. This is not acceptable for two reasons: i) it will favour poor bug reporting practices, an ii) we would have used the bugs twice – once in the inference procedure and another time for the profile, which is theoretically unsound.

e.g. unavailability in the public domain of fully reproducible bug scripts for the commercial servers (especially CS1). Therefore, the sizes of the samples from the partitions on each server are different. Additionally, these servers are not functionally identical: they offer different degrees of compliance with the SQL standard(s) and even proprietary extension to SQL. Bugs affecting one of these extensions, thus, literally cannot exist in a server that lacks the extension. In other words, such bug scripts will provide empirical evidence for the server they were reported for but not for the other servers. We called these "dialect-specific" bugs. Due to this, not all the bugs reported for a server can be run on the other servers. Therefore the number of 'foreign' bug reports varies between the servers.

Table 33 – The observations for the 4 off-the-shelf servers on the bug reports of the different partitions. In the *partition* column we have written in brackets for which server these bugs have been reported.

COTS	Partition	Number of demands N	r_1	r_2	r_3
PG 7.0	$S_{PG7.0}$	57	41	0	11
	$S_{IB6.0}$	28	1	0	0
	S_{CS1}	4	1	0	0
	S_{CS2}	18	6	0	0
IB 6.0	$S_{PG7.0}$	24	0	0	0
	$S_{IB6.0}$	55	37	3	7
	S_{CS1}	4	0	0	0
	S_{CS2}	12	1	0	0
CS1	$S_{PG7.0}$	30	0	0	0
	$S_{IB6.0}$	31	0	0	0
	S_{CS1}	18	10	1	3
	S_{CS2}	12	0	0	0
CS2	$S_{PG7.0}$	33	2	0	0
	$S_{IB6.0}$	35	2	0	0
	S_{CS1}	4	0	0	0
	S_{CS2}	51	28	6	5

4.2.3 The Posterior results

The 50th and 99th percentiles of the marginal distribution, $f_{p^w_{Ser}}(\bullet)$, associated with each server is shown in Table 34. Since the prior distributions are identical for each of the components, then the ordering of the components in the posteriors will be determined by

the observations. The best server, across all the percentiles is CS1. This is not surprising since CS1 did not fail for any of the foreign bugs. The second best server is CS2, although IB 6.0 is very close, both at the 50% and the 99% level of confidence. This is somewhat surprising at first given that this server failed more on the foreign bugs than IB6.0. However, the total number of foreign bugs that could be run on CS2 (72) is much higher than IB6.0 (40). Additionally the number of Heisenbugs for CS2 is also much higher (23.5%) than IB6.0 (14.5%), which leads to the CS2 being better in the posteriors.

Table 34 - The table shows the percentiles of the system quality $f_{P^{w}_{Ser}}(\bullet)$ for each server

Percentiles	0.5				0.99			
COTS	PG7.0	IB6.0	CS1	CS2	PG7.0	IB6.0	CS1	CS2
Priors	0.77	0.77	0.77	0.77	0.94	0.94	0.94	0.94
Posterior	0.42	0.32	0.24	0.3	0.55	0.45	0.32	0.42

4.3 Discussion of the results for the two setups

We have seen that under the more 'stressful' profiles (i.e. Scenario 4 in the TPC-C study and the Bugs study) the best COTS component is CS1. The fact that we have come to the same conclusion using rather different testing methods and different profiles would give us an extra assurance that CS1 is indeed the best component for applications with more stringent reliability and performance requirements which operate at greater transaction load and level of concurrency. However if the concurrency is low, then even with more rigid performance requirements (Scenario 2) Firebird 1.0 server, which is open-source and freely available, comes out as the best server.

The two studies are also in agreement with respect to the worst server – these are the PostgreSQL components.

We could also use the outcome of the studies as a validation of the proposed method. CS1, which came out best, is widely accepted as the best SQL server and has by far the largest share in the market of SQL servers. This gives some confidence that both the data that we used is sufficiently informative to allow for meaningful and accurate discrimination between the competing components and the method itself is trustworthy to provide rigorous ground for accurate COTS component selection.

4.4 Further contrived examples

In the empirical study with the SQL server we could not fully illustrate the interplay between the dependence and the uncertainty in the values of the attributes due to the empirical results often being strikingly different for each server and also because the prior distributions that we started with were the same for each server. In this section we provide some further numerical examples, which illustrate the usefulness of handling uncertainty and dependence between the attribute values explicitly. We comment on the cases where the choice of the best COTS component would differ with conventional assessment methods which rely on point estimates of the attribute values and make assumptions of independence between the values of the attributes. We also discuss the effect of the priors on the selection, including different priors for each of the competing components. The choice of prior distributions and the observations serve illustrative purposes only. The prior, $f_{p_I,p_L,p_{IL}}(\bullet,\bullet,\bullet)$, was constructed under the assumption that $f_{p_I}(\bullet)$ and $f_{p_L}(\bullet)$ are both Beta independently distributed random variables, $Beta(\bullet,a,b)$, defined in the interval [0, 0.01], i.e. $f_{p_I,p_L}(\bullet,\bullet)=f_{p_I}(\bullet)f_{p_L}(\bullet)$.The conditional distributions, $f_{p_{IL}|p_I,p_L}(\bullet\mid P_I,P_L)$, for every pair of values of P_I and P_L, in the general case of incorrect and late responses being non-independent events are also assumed to be Beta distributions, $Beta(\bullet,a,b)$. Clearly they are defined in the range [0, min(P_I, P_L)]. Note that we do not provide any justification for the choice of the prior distributions used here, and neither for the interval on which the distribution is defined; the particular choice of the type of the prior is dictated by some convenience offered by Beta distribution in the examples given below. The assessor can choose any prior distribution and interval that best represents his/her prior beliefs.

4.4.1 Same Priors

In the first example we consider 3 different COTS components, referred to as C1, C2 and C3 respectively for which the prior information does not give any reasons to prefer one to another, i.e. we are indifferent between C1, C2 and C3. The prior distributions, therefore, for all three COTS components are identical. We assumed Beta distributions, defined on [0, 0.01] as described above, with parameters given as follows:

- Beta (2, 10) for *pfd* associated with incorrect results $f_{p_I}(\bullet)$
- Beta (2, 10) for *pld* associated with late results $f_{p_L}(\bullet)$
- Beta (3,3) for the conditional distribution $f_{p_{IL}|p_I,p_L}(\bullet \mid P_I, P_L)$

This completes the definition of the tri-variate distribution, $f_{p_I,p_L,p_{IL}}(\bullet,\bullet,\bullet)$.

The assumed observations for these three COTS components are given in Table 35. For Observation 1 the total number of incorrect or late results are the same for C2 and C3: 5 each. But the failure correlations differ in the two components: for C2 these failures happen on 5 demands (i.e. each of these 5 demands gives both an incorrect and a late response), whereas for C3 they happen on 10 demands (the responses are either incorrect or late). For observation 2 both the total number of failures and the failure correlation are different in the three COTS components.

Table 35 - Observations from testing the COTS components. All observations are from test campaigns of 5000 demands. The observations differ by the number of incorrect (r1+ r3) and late (r2+ r3) responses and the number of incorrect & late (r3) responses.

Observation ID	Number of demands, N	COTS	r_1+r_3	r_2+r_3	r_3
Observation 1	5000	C1	0	0	0
		C2	5	5	5
		C3	5	5	0
Observation 2	5000	C1	20	10	10
		C2	13	13	10
		C3	10	10	0

Table 36 shows the results using the percentiles of the prior/posterior distributions of the probability of an inadequate response P_{Ser}. The posterior distribution for Observation 1 reveals that C1 is clearly the best component, since testing revealed no failures for this component. The interesting results are for C2 and C3. Even though the total number of failures observed for C2 and C3 is the same we can still distinguish between them since the types of failures observed in both cases differ. Positive correlation between the two types of failures is observed for C2 whereas the correlation observed between the types of failure for C3 is negative. As a result, the posterior distribution of C2 after testing with Observation 1 is better than that of C3 for all percentiles. Using conventional methods of assessment, where the attributes are assessed independently, this distinction would have not been possible since the marginal distributions for the two attributes are the same in

both C2 and C3 leading to identical results for these two components. We commented on a similar observation for IB 6.0 and CS2 servers in Section 4.1.3.

The posterior after the Observation 2 is also interesting. The total number of failures observed in C3 is the lowest (20 in total) in comparison with C2 (26) and C3 (30). However the correlation between the two types of failures is very different. In C3 there is a maximum negative correlation between the two types of failure (the observed failures are either incorrect or late responses but not both). For C2 we see 10 incorrect results which are also late. And for C1 we see that all late results are also incorrect. Thus, the observations indicate different degrees of correlation between the two types of failure, which as a result, translates into quite different posteriors for the three COTS components. We would choose C2 as the best COTS component despite the total number of failures (26) observed during testing for this component being higher than the total for C3 (20). This example clearly indicates that the 'uncertainty explicit' assessment method proposed in this paper and conventional assessment methods[52] would have concluded differently regarding which of C2 and C3 should be chosen. The reason for this difference is the correlation between the two types of failure, which we take into account while the conventional methods, which are based on separate assessment of the attributes, would ignore.

Table 36 - The table shows the percentiles of the chosen parameters of system quality.

Percentiles	0.5			0.99		
COTS	C1	C2	C3	C1	C2	C3
System Quality $P_{Ser} = P_I + P_L - P_{IL}$						
Priors	0.0025	0.0025	0.0025	0.0061	0.0061	0.0061
Observation 1	0.0005	0.0011	0.002	0.0015	0.0024	0.0037
Observation 2	0.0033	0.0027	0.0036	0.0051	0.0044	0.0056

4.4.2 Different Priors, same observations

In the second example we will consider 2 different COTS components, COTS 1 and COTS 2 referred to as C1, and C2. The assumed testing results for C1 and C2 are identical. The prior distributions, however, for the two COTS components are now different. We will define Beta distributions again but with different parameters for each COTS component, as given in Table 37.

[52] The conventional methods not exploring the dependence in the values of the attributes would conclude that C3 is better than C2.

Table 37 - The parameters *(a, b)* for the Beta prior distributions defined for each COTS components

COTS	Reliability $f_{P_I}(\bullet)$	Performance $f_{P_L}(\bullet)$	Conditional distribution: $f_{P_{IL}\|P_I,P_L}(\bullet \mid P_I, P_L)$
C1	(5,5)	(5,5)	(3,3)
C2	(15,14)	(15,14)	(9,9)

A high value for the *a* parameter of the Beta distribution means that the distribution is shifted to the right – in our context it represents a prior belief that the number of failures will be high, whereas the *b* parameter shifts the distribution to the left (i.e. a prior belief that the number of failures will be low). The higher the values are the smaller the uncertainty. We can see for example that C1 and C2 are going to have very similar mean values (the mean of the Beta distribution being a / (a + b)) but the prior distribution for C2 is being expressed with much greater certainty. Therefore the prior distribution of C2 will be much less 'spread' from that of C1 as is illustrated in Fig. 17.

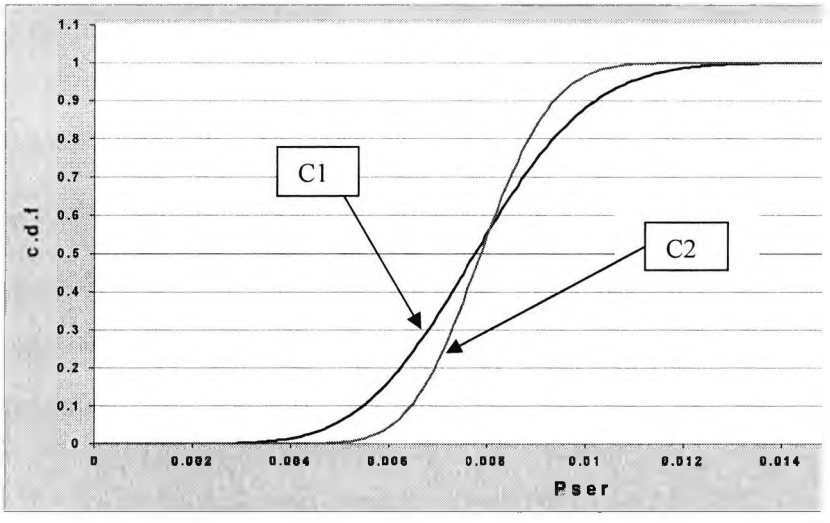

Fig. 17 - The prior distribution for the probability of an inadequate response (P_{Ser}) for C1 and C2. We can see that the prior distribution for C1 is more 'spread' than that of C2 which reflects the assessor's higher uncertainty in the prior beliefs for the values of C1.

We do not make any claims that the priors used in the examples should be used in practical assessment. They serve illustrative purposes only and yet, have been chosen from a reasonable range. For example, the mean of P_I for C1 is 5.10^{-3}, which is a value from a typical range for many software components.

One set of observations were used for the calculations with the number of trials, $N = 5000$ as shown in Table 38.

Table 38 - Observation 3 from testing the COTS components.

Observation ID	Number of tests, N	COTS	r_1+r_3	r_2+r_3	r_3
Observation 3	5000	All	4	4	1

Table 39 shows the results using the percentiles of the prior/posterior distributions of the probability of an inadequate response P_{Ser}. The structure of the table is similar to that of Table 36.

Interesting points with reference to these posterior values are:

- at the 50th percentile, if the selection is based on the prior values the then C1 is the best component. However at the 99th percentile[53] then the ordering changes: C2 is now the preferred choice over the two. For those assessors who prefer to minimize the risk of making 'wrong' decisions with high confidence (i.e. 90%+), C2 is the better choice. This type of distinction would have not been possible in the conventional methods of COTS component assessment which use point values rather than distributions.
- the posterior values of C1 have shifted significantly in comparison with the priors but not as much as those of C2, even though the testing results for these two components are the same. This is due to the prior distributions: for C1 the prior distribution was highly spread, signifying that the uncertainty was high prior to testing; the opposite is true for C2. Therefore the posterior distribution of C1 is influenced by the testing results much more than that of C2.

Table 39 - The table shows the percentiles of the chosen parameters of system quality

Percentiles	0.5		0.99	
COTS	C1	C2	C1	C2
System Quality $P_{Ser} = P_I + P_L - P_{IL}$				
Priors	0.0078	0.0079	0.0122	0.0106
Observation 3	0.0028	0.0046	0.0048	0.0065

[53] The same ordering was observed for all percentiles higher than 90th.

5. Discussion of applicability of the proposed assessment method

5.1 Many assessment attributes

We have illustrated in the previous sections how the assessment can be done for the Reliability and Performance attributes, which are usually the most important attributes for software COTS components compliant with a *known specification (e.g.* SQL servers, J2EE Application servers, Business process execution engines (Andrews, F. Curbera et al. 2003) etc.). We illustrated that, even for assessments in which only two attributes are considered, taking account of the dependence that exists between these attributes can lead to a different decision on which COTS component to choose compared with methods that do not account for this dependence.

It is a common practice that COTS components are assessed in terms of more than 2 attributes, usually many more. The obvious question, therefore, is whether the proposed 'uncertainty explicit' assessment 'scales up' to many attributes. Formally, the question is how the method applies if we have to compare COTS components, each of which is represented by a multivariate distribution $f_p(a_1, a_2, ..., a_n)$. The assessment will deliver posterior distributions $f_p(a_1, a_2, ..., a_n \mid assessment)$, which will be used in the comparison. A new variable should be defined as a function of the variates of the distribution $\{a_1, a_2, \ldots, a_n\}$, e.g. a weighted sum of all the attributes. The uncertainty associated with this aggregate variable is easily derived from the joint posterior $f_p(a_1, a_2, ..., a_n \mid assessment)$. Even though mathematically possible, Bayesian inference with multivariate distributions is difficult. The difficulty has two aspects:

- Specifying a multivariate prior distribution becomes very difficult because many non-intuitive dependencies between the attributes must be defined and *justified.*
- Manipulating a multivariate distribution is non-trivial even using the most advanced math/statistical tools. Calculating the posterior distribution is impracticable with more than 3 variates and without simplifying assumptions about the dependencies between them.

To partially overcome these difficulties a divide-and-conquer approach can be employed. First the attributes can be grouped into smaller groups so that there are dependencies within the groups, which the assessment can capture, but the groups are assumed independent. In other words, knowing the values of the attributes in one group does not change the assessor's knowledge (belief) about the values of the attributes included in the other group. Assume that our initial multivariate distribution can be represented as two independent groups of attributes:

- $f_{p_1,p_2,...p_n}(a_1,a_2,...,a_n) =$

$$f_{p_1,p_2,...p_s}(a_1,a_2,...,a_s) f_{p_{s+1},p_{s+2},...p_{s+n}}(a_{s+1},a_{s+2},...,a_n)$$

- the likelihood of observing the COTS component in operation can be expressed as:

$$L(observation \mid f_{p_1,p_2...p_n}(a_1,a_2,...,a_n)) = L(observation_1 \mid f_{p_1,p_2...p_s}(a_1,a_2,...,a_s)) \times L(observation_2 \mid f_{p_{s+1},p_{s+2},...p_{s+n}}(a_{s+1},a_{s+2},...,a_n))$$

In this case, it trivially follows that:

$$f_{p_1,p_2,...p_n}(a_1,a_2,...,a_n \mid assessment) =$$

$$f_{p_1,p_2,...p_s}(a_1,a_2,...,a_s \mid assessment) \times f_{p_{s+1},p_{s+2},...p_{s+n}}(a_{s+1},a_{S+2},...,a_n \mid assessment)$$

From the attribute definition papers surveyed (e.g. (Bertoa and Vallecillo 2002), (Torchiano and Jaccheri 2003)) we failed to find other examples of attributes which could be assessed objectively using the demand notation we explained in Section 3. An example could be "Recoverability" (which again can be characterised in terms of correctness of the recovery and the timeliness of the recovery), but treating the Reliability / Performance on the one hand and Recoverability on the other as independent groups is an unreasonable assumption since recovery will only be needed once a failure has happened (therefore strong dependency exists between Reliability attribute and the Recoverability attribute).

5.2 Decisions on how to perform the assessment

We outlined the problems with assessment of a large number of attributes due to the complex interdependencies that may exist between the different attributes. The higher the number of attributes to be assessed and the higher the number of independence

assumptions that are made the more difficult it becomes to place a high degree of confidence in the results obtained from the assessment. The limitations we have outlined in Section 5.1 are *not specific* to our assessment method; in fact they are more serious for the conventional methods in which the individual attributes are assessed separately. We illustrated with numerical examples in Section 4.4 that even when the assessment is done using two attributes, ignoring the dependence between the values of the attributes may lead to wrong decisions: a sub-optimal component may wrongfully be chosen as the best one. If this could be observed with only two attributes, then it is bound to be much more pronounced with more than two attributes, where many more dependencies may exists between the values of the attributes.

Doing the assessment with 'independent groups' of attributes also has its problems. It can only be applied if the assessor can justify that assuming a set of independent pairs is plausible. Despite this problem, however, using small independent groups is still an improvement compared with the extreme assumption used implicitly in the existing assessment methods surveyed, that all of the attributes are independent.

It is worth pointing out that many of the attributes, such as 'has the required functions', various forms of compliance, e.g. 'complies with certain standards', "Backward Compatibility", etc. (Bertoa and Vallecillo 2002), do not require any uncertainty attached to their values. Assessment with respect to such attributes normally leads to a reduction of the number of the COTS components (which satisfy all these 'binary' attributes), for which the more thorough assessment with respect to the remaining 'non-binary' attributes needs to be done (Ncube and Maiden 1998).

5.3 The types of COTS components for which the assessment method can be applied

The method of assessment proposed in this paper would be applicable to any family of COTS components. The setup described in Section 3 and illustrated in Section 4 is particularly relevant for COTS components with *stringent reliability and performance* requirements. In Section 4 we provided empirical results using off-the-shelf database servers. There is a plethora of off-the-shelf database servers, both open source and

commercial. Deciding which one to choose among the many choices available overwhelmingly depends on the reliability of servers and their performance.

Java Virtual Machines (JVMs), various application servers, web servers and Business process execution engines (Andrews, F. Curbera et al. 2003) are also examples of COTS components were reliability and performance requirements are usually the deciding attributes for selection. For these components we may not need to deal with more than 2 attributes, i.e. our 2-attribute model proposed in Section 3 is immediately applicable without any simplifying assumptions.

5.4 Other ways of eliciting the prior distributions

The prior definition in Bayesian assessment is crucial. In our studies we have assumed that prior distributions for each component are the same. This was due to the unavailability of other known 'objective' evidence that we could use to define more accurate priors. Anecdotal evidence about the servers does exist, but is difficult to translate these subjective beliefs into priors in the form required by our method. By assuming that the prior distributions were the same for each server, the decision on which server is chosen is dictated by the observations only. As a result the decision of the types of distributions for the random variables in our study becomes less important.

However there are other ways of deriving more accurate priors. We could, for example, utilize evidence from *earlier versions* of the servers and then do multiple steps of inference, i.e. if we want to perform the assessment with later versions of the servers in our study (e.g. with versions of PostgreSQL after release 7.2 or Firebird after release 1.0) we can use the posteriors derived here as priors for the later versions, collect the new evidence for the later versions and then use the model to derive the posteriors for each. This approach has also been reported elsewhere (Littlewood and Wright 1997).

6. Related work

6.1 COTS assessment methods

There are a wide variety of COTS component assessment approaches available. All of them start with an initial statement of requirements, which defines what is being sought.

It has been proposed that the requirements initially should not be too stringent, since this would discard potentially appropriate COTS component candidates at a very early stage (Dean 2000), (Lewis, Hyle et al. 2000). It has even been suggested (Lewis, Hyle et al. 2000) that if the requirements are not flexible then the COTS-based development may not be appropriate and bespoke development could be more cost-effective. So initially (Lewis, Hyle et al. 2000) suggests distinguishing between essential requirement and those that are negotiable. The selection criteria are then based on the essential requirements.

Off-the-shelf-option (OTSO) (Kontio, Chen et al. 1995) is a multi-phase approach to COTS component selection. The phases are: the search phase, the screening and evaluation phase and the analysis phase. In the first phase COTS components are identified. In the screening and evaluation phase the components are further filtered using a set of evaluation criteria (established from a number of sources, including the requirements specification, the high level design specification etc.). In the analysis phase results of the evaluation are analysed, which lead to the final selection of COTS components for inclusion in the system. Other similar multiphase process approaches for COTS component evaluation that have been proposed include CEP (Comparative Evaluation Process Activities) (Phillips and Polen 2002), CBA Process Decision Framework (Boehm, Port et al. 2003) which in addition to defining a process for COTS component assessment also defines two other processes: COTS integration ("gluing") and COTS configuration ("tailoring"); CAP-COTS Acquisition Process method (Ochs, Pfahl et al. 2001) and PECA Process (Comella-Dorda, Dean et al. 2002).

Procurement-oriented requirements engineering (PORE) (Ncube and Maiden 1999) is a process in which requirements are defined in parallel with COTS component evaluation and selection. (Ncube and Maiden 1999) propose using prototypes to develop knowledge concerning COTS components and their use within the wider system. Other methods that are centred on the requirements to assist with the COTS component selection process are CRE-COTS-Based Requirements Engineering Method (Alves and Castro 2001), Storyboard Process (Gregor, Hutson et al. 2002), Combined Selection of COTS Components (Burgués, Estay et al. 2002) and COTS-DSS (Ruhe 2003).

CISD (COTS-based Integrated System Development) (Tran and Liu 1997) and CDSEM (Checklist Driven Software Evaluation Methodology) (Jeanrenaud and Romanazzi 1994)

are both checklist-based evaluation methodologies. STACE (Socio Technical Approach to COTS Evaluation) (Kunda and Brooks 1999) is a socio-technical approach to evaluation which builds on work of (Ncube and Maiden 1999) and (Kontio, Chen et al. 1995) and emphasizes the organizational issues related to COTS selection.

6.2 Attribute definition methods

Extensive work has been also reported on definition of COTS component assessment attributes. A comprehensive list is given in (Bertoa and Vallecillo 2002). They group the attributes in two categories depending on how they can be measured: Attributes Measurable at Runtime (which contain Accuracy, Security, Recoverability, Time Behaviour and Resource Behaviour) and Attributes Measurable during Component Life-Cycle (Suitability, Interoperability, Maturity, Learnability, Understandability, Operability, Changeability, Testability and Replaceability). These attributes are further divided into more fine-grained attributes, which are measurable using their proposed metrics of: presence, time, level and ratio. This work (Bertoa and Vallecillo 2002) follows the spirit of the guidelines for attribute definitions outlined by the international standardizing organizations ISO (ISO/IEC-9126-1:2001 2001), and IEEE/ANSI (IEEE/ANSI 1993) in a broader context, not specific to COTS component attributes. COCOTS framework by Abts et al. (Abts, Boehm et al. 2001), and Torchiano and Jaccheri (Torchiano and Jaccheri 2003) also provides COTS attribute definitions.

7. Conclusion

To handle the inherent uncertainty in the COTS component assessment we propose the use of "uncertainty explicit" methods. As Bayesian approach to representing uncertainty has been successfully applied in other contexts of assessment we have defined a Bayesian model that can be used for assessment of COTS components with respect to two related attributes. This approach complements the conventional selection procedures with more powerful calculus, which can take into account the uncertainty explicitly.

We have conducted an empirical study with off-the-shelf database servers which also illustrated the use of the method. The contribution of this paper is in several aspects:

- We have demonstrated in the context of the COTS component assessment how to apply the Bayesian methods which have had some popularity in reliability assessment of both software and hardware.
- We have described the use of the model in selecting the best off-the-shelf database server from a sample of different servers, using two sources of data:
  - Experiments using an implementation of the TPC-C client benchmark for database severs
  - Known faults reported for four different servers.
- We recommend that the 'uncertainty explicit' assessment methods be considered at least as a non-expensive warranty against badly sub-optimal decisions possible with the conventional COTS component selection methods (we provided contrived numerical examples which show examples of sub-optimal selections of COTS components if uncertainty or dependence in the values of the attributes are ignored).
- We have also demonstrated how our model can be extended and used with a partitioned demand space which allows utilization in the inference of all the evidence available from the different partitions

An interesting observation from the study with SQL servers is that the results of the inference with the more stressful setups (scenario 4 of TPC-C study and the bugs study) both lead to CS1 being preferred as the best server and PG servers being the worst. This may give the assessor further assurance of preferring CS1 for an application with more stringent reliability, performance and concurrency requirements given that it performed best under two very different but 'stressful' profiles. Interestingly, CS1 is considered by many as a leader among the SQL server vendors, which may be seen as validation of the method's usefulness for making a correct choice among several similar COTS components despite the scarcity of the data that we could use.

There are several well-known difficulties of using Bayesian assessment. Bayesian assessment does not scale up well due to:

- the difficulty with specifying a multivariate prior distribution when the number of attributes to be assessed increases, unless independence is assumed among the attributes

- defining the prior is crucial. It may be difficult for practitioners, not comfortable with non-trivial math, to express their individual beliefs as probability distributions.
- the difficulty with manipulating a multivariate distribution, which becomes impracticable with more than 3 variates if no simplifying assumptions are made.

Future work that is desirable includes:

- Methods are needed which would allow for effective assessment with a *large number of related attributes*. Currently the 'uncertainty explicit' assessment only works with a limited number of related attributes (or with independent groups of attributes in which the number of attributes in the groups is small).
- Further development of the theoretical framework is needed for cases of groups of more than 2 dependent attributes. Conceptually, the multivariate inference is no different than the 1- and 2-variate inferences. Its practical use, however, is currently problematic.

Acknowledgement

This work was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under the Interdisciplinary Research Collaboration in Dependability of Computer-Based Systems (DIRC) project. We would also like to thank Professor Bev Littlewood for his comments on an earlier version of this paper.

References

Abts, C., B. Boehm and E. B. Clark (2001), "*COCOTS: A software COTS-Based System (CBS) Cost Model*", *in proc.* Conf. on European Software Control and Metrics (ESCOM'01), London, UK, pp: 1-8.

Adams, E. N. (1984), "*Optimizing Preventive Service of Software Products*", IBM Journal of Research and Development 28(1), pp: 2-14.

Alves, C. and J. Castro (2001), "*CRE: A Systematic Method for COTS Components Selection*", *in proc.* XV Brazilian Symp. on Software Engineering (SBES), Rio de Janeiro, Brazil.

Andrews, T., F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte and I. Trickovic (2003), "*Business Process Execution Language for Web Services version 1.1*", http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

Bertoa, M. F. and A. Vallecillo (2002), "*Quality Attributes for COTS Components*", *in proc.* 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002), Málaga, Spain, pp: 54-66.

Boehm, B., D. Port, Y. Yang, J. Bhuta and C. Abts (2003), "*Composable Process Elements for Developing COTS-Based Applications*", *in proc.* Symp. on Empirical Software Engineering. (ISESE'03), ACM-IEEE, pp: 8-17.

Brocklehurst, S., P. Y. Chan, B. Littlewood and J. Snell (1990), "*Recalibrating software reliability models*", IEEE Transactions on Software Engineering 16(4), pp: 458-470.

Burgués, X., C. Estay, X. Franch, J. A. Pastor and C. Quer (2002), "*Combined Selection of COTS Components*", *in proc.* Int. Conf. on COTS-Based Software Systems (ICCBSS '02), Florida, USA, Springer-Verlag, pp: 54-64.

Chandra, S. and P. M. Chen (2000), "*Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '00), NY, USA, IEEE Computer Society Press, pp: 97-106.

Comella-Dorda, S., J. Dean, E. Morris and P. Oberndorf (2002), "*A Process for COTS Software Product Evaluation*", *in proc.* Int. Conf. on COTS-Based Software Systems (ICCBSS '02), Florida, USA, Springer-Verlag, pp: 86-92.

Dean, J. (2000), "*An Evaluation Method for COTS Software Products*", http://www.stc-online.org/cd-rom/cdrom2000/webpages/johndean/paper.pdf.

Eckhardt, D. E., A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk and J. P. J. Kelly (1991), "*An experimental evaluation of software redundancy as a strategy for improving reliability*", IEEE Transactions on Software Engineering 17(7), pp: 692-702.

Eckhardt, D. E. and L. D. Lee (1985), "*A theoretical basis for the analysis of multiversion software subject to coincident errors*", IEEE Transactions on Software Engineering 11(12), pp: 1511-1517.

Gashi, I., P. Popov, V. Stankovic and L. Strigini (2004a), "*On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*", *in* Architecting Dependable Systems II, R. de Lemos, Gacek, C., Romanovsky, A. (Eds.), Springer-Verlag, 3069, pp: 191-214.

Gashi, I., P. Popov and L. Strigini (2004b), "*Fault Diversity Among Off-The-Shelf SQL Database Servers*", *in proc.* Int. Conf. on Dependable Systems and Networks (DSN '04), Florence, Italy, IEEE Computer Society Press, pp: 389-398.

Gray, J. (1986), "*Why Do Computers Stop and What Can be Done About it?*" *in proc.* Int. Symp. on Reliability in Distributed Software and Database Systems (SRDSDS '86), Los Angeles, CA, USA, IEEE Computer Society Press, pp: 3-12.

Gregor, S., J. Hutson and C. Oresky (2002), "*Storyboard Process to Assist in Requirements Verification and Adaptation to Capabilities Inherent in COTS*", *in proc.* Int. Conf. on COTS-Based Software Systems (ICCBSS '02), Florida, USA, Springer-Verlag, pp: 132-141.

Hamlet, D. and R. Taylor (1990), "*Partition testing does not inspire confidence*", IEEE Transactions on Software Engineering 16(12), pp: 1402-1411.

IEEE/ANSI (1993), "*Recommended Practice for Software Requirements Specifications, International Standard 830-1993*", IEEE.

ISO/IEC-9126-1:2001 (2001), "*Information technology – Product Quality – Part1: Quality Model*", International Standard ISO/IEC 9126, International Standard Organization, June, 2001.

Jeanrenaud, J. and P. Romanazzi (1994), "*Software Product Evaluation: A Methodological Approach*", *in proc.* Software Quality Management II: Building Software into Quality, pp: 55-69.

Jeng, B. and E. J. Weyuker (1991), "*Analyzing partition testing strategies*", IEEE Transactions on Software Engineering 17(7), pp: 703-711.

Knight, J. C. and N. G. Leveson (1986), "*An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming*", IEEE Transactions on Software Engineering 12(1), pp: 96-109.

Kontio, J., S. Y. Chen, K. Limperos, R. Tesoriero, G. Caldiera and M. Deutsch (1995), "*A COTS Selection Method and Experiences of Its Use*", *in proc.* Twentieth

Annual Software Engineering Workshop,NASA Goddard Space Flight Center, Greenbelt, Maryland.

Kunda, D. and L. Brooks (1999), "*Applying Social-Technical Approach for COTS Selection*", *in proc.* UK Academy for Information Systems (UKAIS'99), University of York, England.

Lewis, P., P. Hyle, M. Parrington, E. Clark, B. Boehm, A. Abts and R. Manners (2000), "*Lessons Learned in Developing Commercial Off-The-Shelf (COTS) Intensive Software Systems*",
http://www.cebase.org/www/researchActivities/COTS/LessonsLearned.pdf.

Likert, R. (1932), "*A Technique for the Measurement of Attitudes*", New York, McGraw-Hill.

Littlewood, B. and D. R. Miller (1989), "*Conceptual Modelling of Coincident Failures in Multi-Version Software*", IEEE Transactions on Software Engineering 15(12), pp: 1596-1614.

Littlewood, B., P. Popov and L. Strigini (2000), "*Assessment of the Reliability of Fault-Tolerant Software: a Bayesian Approach*", *in proc.* Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP '00), Rotterdam, the Netherlands, Springer, pp: 294-308.

Littlewood, B. and D. Wright (1997), "*Some conservative stopping rules for the operational testing of safety-critical software*", IEEE Transactions on Software Engineering 23(11), pp: 673-683.

Lyu, M. R., Ed. (1996), "*Handbook of Software Reliability Engineering*", McGraw-Hill and IEEE Computer Society Press.

Musa, J. D. (1993), "*Operational Profiles in Software-Reliability Engineering*", IEEE Software (March), pp: 14-32.

Ncube, C. and N. Maiden (1998), "*Acquiring COTS Software Selection Requirements*", IEEE Software 15(2), pp: 46-56.

Ncube, C. and N. Maiden (1999), "*PORE:Procurement Oriented Requirements Engineering Method for the Component-Based Systems Engineering Development Paradigm*", *in proc.* Int. Workshop on Component-Based Software Engineering.

Ochs, M., D. Pfahl, G. Chrobok-Diening and B. Nothhelfer-Kolb (2001), "*A Method for Efficient Measurement-based COTS Assessment and Selection -Method Description and Evaluation Results*", *in proc.* 7th Symp. on Software Metrics, London, England, IEEE Computer Society, pp: 285-294.

Phillips, B. C. and S. M. Polen (2002), "*Add Decision Analysis to Your COTS Selection Process*", http://www.stsc.hill.af.mil/crosstalk/2002/04/phillips.html.

Popov, P. (2002), "*Reliability Assessment of Legacy Safety-Critical Upgraded with Off-the-Shelf Components*", *in proc.* Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP '02), Catania, Italy, Springer, pp: 139-150.

Popov, P., L. Strigini, A. Kostov, V. Mollov and D. Selensky (2004), "*Software Fault-Tolerance with Off-the-Shelf SQL Servers*", *in proc.* Int. Conf. on COTS-based Software Systems (ICCBSS '04), Redondo Beach, CA USA, Springer, pp: 117-126.

Port, D. and S. Chen (2004), "*Assessing COTS assessment: How much is enough?*" *in proc.* Int. Conf. on COTS Based Software Systems (ICCBSS '04), Redondo Beach, California, Springer-Verlag, pp: 183-198.

Ruhe, G. (2003), "*Intelligent Support for Selection of COTS Products*", *in proc.* Web, Web-Services, and Database Systems, Springer, pp: 34-45.

Stankovic, V. and P. Popov (2006), "*Improving DBMS Performance through Diverse Redundancy*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '06), Leeds, UK, IEEE Computer Society, pp: 391-400.

Torchiano, M. and L. Jaccheri (2003), "*Assessment of Reusable COTS Attributes*", *in proc.* Int. Conf. on COTS Based Software Systems (ICCBSS '03), Ottawa, Canada, Springer-Verlag, pp: 219 - 228.

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

Tran, V. and D.-B. Liu (1997), "*A Risk Mitigating Model for the Development of Reliable and Maintainable Large-Scale Commercial-Off-The-Shelf Integrated Software Systems*", *in proc.* Reliability and Maintainability Symp. (RAMS'97), IEEE Print, pp: 361-367.

Wright, D. and K.-Y. Cai (1994), "*Representing Uncertainty for Safety Critical Systems*", PDCS2 Tech. Rep. 135. Centre for Software Reliability, City University, London.

VII. Conclusions

1. Introduction

Each of the papers detailed in chapters IV-VI had their own Conclusions section. However the conclusions section of each paper will naturally only be with respect to the results and analysis presented on that respective paper. Additionally, since the papers were written as the research was progressing the conclusions drawn there had to be revisited and updated, especially for the papers published during the earlier stages of research. The purpose of this chapter is to link and present those conclusions in a single coherent chapter. It will also outline the provisions for further work.

2. Summary of conclusions

The main purpose of this thesis was to explore and estimate the possible advantages of using modular-redundant diversity in complex off-the-shelf software. To this end two studies were conducted with samples of bug reports from four popular off-the-shelf SQL DBMS products and later releases of two of them. The purpose of these studies was to check for bugs that would cause common-mode failures if the products were used in a diverse redundant architecture. Such common bugs were found to be rare. For most bugs, failures would be detected (and may be masked) by a simple two-diverse channel configuration using different DBMS products. In summary:

- out of the 273 bug scripts run in both the studies conducted, very few bug scripts were found that affected two DBMS products and none that affected more than two.
- only five of these bug scripts caused identical, non-detectable failures in two DBMS products:
  - of these five, one caused non-detectable failures on only a few among the demands affected.

The results of the second study, on later releases of the same products, substantially confirmed the general conclusions of the first study: one may conclude that the factors that make diversity useful do not disappear as the DBMS products evolve and become more reliable.

Other interesting observations include:

- there is strong evidence against the fail-stop failure assumption for DBMS products. The majority of bugs reported, for all products, led to "incorrect result" failures rather than crashes (64.5% vs 17.1% in the first study; 65.5% vs 19% in the second), despite crashes being more obvious to the user (i.e. easier to detect and report). Even though these are bug reports and not failure reports, *this evidence goes against the common assumption that the majority of failures are engine crashes*, and warrants more attention by users to fault-tolerant solutions, and by designers of fault-tolerant solutions to tolerating subtle and non fail-silent failures;
- it may be worthwhile for vendors to test their DBMS products using the known bug reports for other DBMS products. For example, in the first study 4 MSSQL bugs were observed that had not been reported in the MSSQL service packs (previous to the observation period in which the bugs were collected). Oracle was the only DBMS product that never failed when running on it the reported bugs of the other DBMS products;

Using successive releases of the *same* product for fault tolerance also appeared useful, although less than using diverse products from different vendors. A high level of fault diversity between successive releases of PostgreSQL was found: most of the old bugs had been fixed in the new release; many of the newly reported bugs did not cause failure (or could not be run at all) in the old release. This more limited form of diversity may be especially useful for legacy applications written for an older release of a product. New releases are usually written so that they are backward compatible with the older releases, but new bugs may also be introduced (as we have observed with PostgreSQL). Therefore using different releases of the same product may bring some dependability improvements for the legacy applications.

These results were very encouraging and pointed to serious gains in dependability from using diverse off-the-shelf DBMS products. The architectural solutions that facilitate the use of this diversity were then explored and analysed. The mechanism of "data diversity" (Ammann and Knight 1988) and its application with SQL DBMS products was conducted as part of this analysis. 14 generic "rephrasing rules" were defined which can be implemented in a "rephrasing" algorithm. These rules can then be applied to specific

SQL statements and generate logically equivalent statements. By generating additional responses from the DBMS products "rephrasing" can thus help with failure diagnosis and state recovery. It was also argued that since these rules are transformations of the SQL language syntax, they are amenable to formal analysis, and dependability gains from employing rephrasing are achievable despite the need for development of bespoke new code.

The analysis of possible gains from using *diverse DBMS products*, *different releases of the same DBMS product* and also *data diversity* allows users who wish to explore fault tolerance with these products various architectural options. These users therefore have various trade-offs available between the wishes to exploit dialectal features and to get effective diversity.

As discussed extensively in the thesis, the results derived from the bug reports must be treated with caution, and their immediate implications vary between users, but for some classes of DBMS product installations, diversity could already be recommended as a prudent and cost-effective strategy. Examples of these installations are those that use mainly the core features of DBMS products (recommended by practitioners to improve portability of the applications), have modest throughput requirements for write statements (which make it easy to accept the synchronization delays of a fault-tolerant diverse server) or, most importantly, have serious concerns about dependability (e.g., high costs for interruptions of service or for undetected incorrect data being stored). The need for middleware to manage diverse DBMS products is an obstacle to users wishing to try out diversity in their applications. But the results in this thesis provide a good business case for implementing the required middleware software. Separate add-on "wrapping" components may also be developed for DBMS products to enable a more seamless integration at a higher middleware layer. For example, the SQL dialect translators which need to be defined for each DBMS product may be implemented as add-ons for each respective DBMS: the "wrapped" DBMS product is then connected to the middleware of a diverse fault-tolerant server.

The performance penalty due to controlling concurrency via the middleware would be a problem with write-intensive loads, but not if concurrent updates are rare (Stankovic and Popov 2006).

The final stage of the research conducted in this thesis concerned exploring methods of incorporating bug reports as empirical evidence with existing methods of reliability assessment of 1-out-of-2 systems of diverse products. The need for this research stems from the unavailability of failure data which would be much better as reliability evidence. Detailed failure data are rarely published and they may even be unavailable to the software vendors themselves. This research was in three directions:

- a model of *Bayesian* assessment developed elsewhere (Littlewood, Popov et al. 2000) was explored and extended. The model was used, with the bug reports as evidence, to perform the assessment and choose the best DBMS product pair. The results of the posteriors of DBMS product pairs were compared with those of single DBMS products and it was observed that even the worst DBMS product pair still performs much better than the best single DBMS product. This further reinforces the conclusions from the studies with the bugs that significant dependability gains may be obtained from using diverse off-the-shelf DBMS products. It is also interesting to note that the assessment concluded that the best single DBMS product is a commercial one (namely Oracle), whereas the best pair of DBMS products is the pair PostgreSQL & Interbase both of which are free and open-source.
- an existing Reliability Growth Model (Littlewood 1981) was extended for assessment of 1-out-of-2 systems of diverse products. The model parameters may be inferred with evidence from bug reports as well but they are not enough: additional data is required about the download rates of the DBMS products before proxies can be calculated for inter-failure times. This latter evidence did not exist for the DBMS product versions used in this thesis, therefore the extended model could not be utilized to make reliability growth predictions.
- in the absence of even proxies for inter-failure times a final approach was explored to use the results from the bugs study to estimate the likely gains in reliability that may be expected from switching to a 1-out-of-2 diverse system. Despite some fairly conservative assumptions that were made in the modelling it was found that, for the DBMS products used in this thesis, an order of magnitude increase in reliability may be expected when switching from a single DBMS

product to a 1-out-of-2 system made up of 2 diverse DBMS products. However it was emphasized that this result should be treated with caution, due to the small sample sizes and relatively high estimation errors.

Finally, the *Bayesian* model (Littlewood, Popov et al. 2000) used previously for assessment of a 1-out-of-2 system was adapted and applied in a different context: assessing a single software product from the viewpoint of the *timeliness* (performance) and *correctness* (reliability) of the products results. This model was then applied to select a single best DBMS product using evidence from the bugs study. It was interesting to observe that the most optimal DBMS product selected when bug reports were used as evidence was the same as the product selected by a colleague who used the same model with a different set of data as evidence (an experimental study with an implementation of a TPC-C benchmark (TPC 2002)). This may give the assessor further confidence of choosing that DBMS product for an application with more stringent reliability, performance and concurrency requirements given that it performed best under two very different but "stressful" profiles. Interestingly, this DBMS product (Oracle) is considered by many as a leader among the DBMS product vendors, which may be seen as validation of the method's usefulness for making a correct choice among several similar COTS products despite the scarcity of the data that might be available and the simplifying assumptions that the method is based on.

As discussed extensively in the thesis, the results derived from the assessment in chapter VI should also be treated with caution due to the low sample sizes and, for the Bayesian model in Papers 5 and 7, due to the crucial role of the prior distribution definitions. In paper 5 a discussion was provided of alternative sources of deriving more accurate priors, such as using evidence from previous versions of these DBMS products. Another source for deriving more accurate priors are the results from the dependability benchmarking experiments: results from these experiments are reported in (Kanoun and Crouzet 2006). Multiple steps of inference are also possible: first evidence from the bug studies are used to derive posterior distributions (as was reported in Papers 5 and 7) and then these posteriors are fed in as prior distributions in another step of inference with results from the dependability benchmarking (or other statistical testing) experiments.

Complete failure logs would be much more useful as evidence in the assessment of the dependability benefits of diversity. The results reported in this thesis indicate that using diverse open-source DBMS products may bring substantial dependability gains even when compared with very expensive non-diverse commercial products. This should serve as an incentive to the open-source community to develop accurate failure monitoring facilities (similar to those reported in (Voas 2000)): the availability of failure data would allow an assessor to obtain more accurate results with much higher confidence about the dependability benefits of diversity and may therefore lead to adoption of open-source products, in diverse configurations, in much wider applications than where they are used presently.

3. Review of aims and objectives

In the Introduction chapter of this thesis a set of questions were listed that the research conducted as part of this thesis aimed to provide answers for. These questions will be revisited in this section and short concise answers will be provided based on the results and analysis provided thus far in this thesis.

What dependability gains may be achieved from the use of fault tolerance mechanisms with OTS products?

- The results obtained from the two studies with OTS DBMS products point to serious dependability gains that may be obtained from using diverse DBMS products: very few bugs cause failures in more than one product and none of them cause failures in more than two.
- Limited, but in some cases significant, gains in dependability may also be obtained from employing different releases of the *same* DBMS product.
- Data diversity (Ammann and Knight 1988) mechanism in the form of SQL rephrasing was found to be a useful mechanism especially for aiding with fault diagnosis and state recovery.

What dependability information/data exists for OTS products and how can this information be used to assess the dependability gains that may be achieved from employing fault tolerance mechanisms?

- For OTS DBMS products the only direct dependability evidence found were the bug reports.
- Three different approaches were explored to utilise the bug reports in the assessment of the reliability of a 1-out-of-2 system of DBMS products.
  - Use of two of these approaches was demonstrated using the bug reports collected as part of this study.
  - One of the approaches (namely the extended Littlewood model (Littlewood 1981)) could not be used since additional data is required about the rate of downloads which did not exist for the release of DBMS products used in this thesis.

For fault tolerance configurations employing diverse modular redundancy, which OTS products, from the (possibly many) available ones, should be chosen to obtain the best dependability gains?

- An existing model (Littlewood, Popov et al. 2000) was extended and the evidence from the studies with the bugs was used to asses the various pairs of DBMS products so as to choose the most optimal dependable pair.

What are the implementation difficulties?

- The architectural solutions for employing diversity with OTS DBMS products were given. Users will have many options available to them with various tradeoffs between functionality, dependability and performance.

What costs (developmental, procurement, operational, maintenance etc.) may be expected?

- This question is difficult to answer as it will be application specific. The question is made harder due to the unavailability of off-the-shelf middleware that would allow users to test diversity in their applications. However, as mentioned before, the results presented in this thesis would provide a strong business case for building such middleware solutions for DBMS products. This would in turn make the estimation of total cost of employing diverse products much easier.

4. Future work

There are several strands of future work that may follow the work presented in this thesis. They include:

- Statistical testing of DBMS products to assess the actual reliability gains from diversity. Several million queries with various loads, including ones based on the TPC-C benchmark have been run by researchers in our centre so far observing no failures (however, significant potential for performance gains from using diverse DBMS products was found (Stankovic and Popov 2006)). These results may not be particularly surprising, since these benchmarks use a limited set of well-exercised features of DBMS products. It would be interesting to repeat the tests with test loads that do not suffer from this limitation. An example might be the test loads defined for DBench (Kanoun and Crouzet 2006) which apart from the *workload* (based on TPC-C benchmark) also provide a *faultload*: these results may then be used as evidence in the assessment model defined in Paper 5 and 7, as was discussed earlier in this chapter;
- Developing the necessary middleware components for users to be able to try out data replication with diverse DBMS products in their own installations. Lack of these components is the main practical obstacle in the way of the adoption and practical evaluation of these solutions. There are signs that some DBMS product vendors may also help with this problem: EnterpriseDB (EnterpriseDB 2006) and Fyracle (Janus-Software 2006) are Oracle-mode implementations based on PostgreSQL and Firebird DBMS engines, respectively. With these solutions the problem with SQL dialects is significantly reduced. Some preliminary studies have been completed by undergraduate students in our centre on implementing translators between MSSQL and Oracle dialects for SELECTs, and between Oracle and PostgreSQL dialects for SELECT, INSERT and DELETE statements;
- Methods are needed which enable effective assessment with a higher number of COTS products in a diverse setup (more than two products may be desirable in a diverse setup to enable majority voting on the results from the products). This is a problem at the moment due to the difficulty in specifying and justifying multivariate distributions in Bayesian statistics. The same problem applies for the

effective assessment of even single products but when more than two related attributes are used in the assessment (i.e. more than just reliability and performance)

- Methods for obtaining more accurate proxies for usage time are also needed to allow for effective use of the extended Littlewood model described in this thesis.
- Empirical investigations of the consistency of β factor estimates (explained in the "proportions" approach in chapter VI) in successive releases of the same DBMS product pair are desirable. The β factor estimates seem to be fairly consistent for the successive release of the DBMS products used in this thesis but the sample is too low to make more general conclusions.
- Methods presented in chapter VI should be applied to other types of off-the-shelf products (such as diverse web-servers, application servers etc).

5. Final remarks

Despite the age-old belief that "two heads are better than one" the use of full-fledged diversity, multiple diverse redundant channels, was long thought to be too expensive an approach of increasing system dependability. With the availability of myriad of off-the-shelf software products, many of which are free and/or open-source, the use of diversity becomes a much more realistic possibility. However the problems of assessing the dependability gains that are achievable still remain. The scarcity of direct dependability data makes the assessment even more difficult. The work presented in this thesis has presented evidence that potentially significant dependability gains may be achieved with a very complex and highly used category of off-the-shelf products, namely DBMS products, and has also demonstrated how the assessment of the likely gains may be performed by utilizing the only direct dependability evidence that exists for these products, namely the bug reports. To the best of my knowledge similar empirical work or assessment approach has not been reported elsewhere.

References

Ammann, P. E. and J. C. Knight (1988), "*Data Diversity: An Approach to Software Fault Tolerance*", IEEE Transactions on Computers 37(4), pp: 418-425.

EnterpriseDB (2006), "*EnterpriseDB*", http://www.enterprisedb.com/.

Janus-Software (2006), "*Fyracle*", http://www.janus-software.com/fb_fyracle.html.

Kanoun, K., H. Madeira, et al. (2004), "*DBench Dependability Benchmarks*", IST-2000-25425, http://www.laas.fr/DBench/Final/DBench-complete-report.pdf.

Littlewood, B. (1981), "*Stochastic Reliability Growth: a Model for Fault-Removal in Computer Programs and Hardware Designs*", IEEE Transactions on Reliability R-30(4), pp: 313-320.

Littlewood, B., P. Popov and L. Strigini (2000), "*Assessment of the Reliability of Fault-Tolerant Software: a Bayesian Approach*", *in proc.* Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP '00), Rotterdam, the Netherlands, Springer, pp: 294-308.

Stankovic, V. and P. Popov (2006), "*Improving DBMS Performance through Diverse Redundancy*", *in proc.* Int. Symp. on Reliable Distributed Systems (SRDS '06), Leeds, UK, IEEE Computer Society, pp: 391-400.

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

Voas, J. (2000), "*Deriving Accurate Operational Profiles for Mass-Marketed Software*", http://www.cigital.com/papers/download/profile.pdf.

List of Abbreviations

ACM	–	Association of Computing Machinery
ANSI	–	American National Standards Institute
API	–	Application Programmers Interface
CDF	–	Cumulative Distribution Function
COTS	–	Commercial Off-The-Shelf
DBMS	–	Database Management System
DIRC	–	Interdisciplinary Research Collaboration in Dependability project
DOTS	–	Diversity with Off-The-Shelf components project
DSN	–	Dependable Systems and Networks conference
EDCC	–	European Dependable Computing Conference
EPSRC	–	Engineering and Physical Sciences Research Council
FB	–	Firebird DBMS
FT	–	Fault-Tolerant
IB	–	Interbase DBMS
IEEE	–	International Electrical and Electronic Engineering
ISO	–	International Organisation for Standardisation
JDBC	–	Java Database Connectivity
MS	–	Microsoft SQL Server DBMS
MSSQL	–	Microsoft SQL Server DBMS
N.S.E.	–	Non-Self-Evident failure
OR	–	Oracle DBMS
OTS	–	Off-The-Shelf
PDF	–	Probability Density Function
PFD	–	Probability of Failure on Demand
PG	–	PostgreSQL DBMS
PLD	–	Probability of a Late response on Demand
ReSIST	–	Resilience for Survivability in Information Society Technologies project
S.E.	–	Self-Evident failure
SQL	–	Structured Query Language
TPC	–	Transaction Processing Council

UML – Unified Modelling Language

The appendix of this thesis can be downloaded from:

http://www.csr.city.ac.uk/people/ilir.gashi/thesis/