

Software Fault-Tolerance with Off-the-Shelf SQL Servers

P. Popov[†], L. Strigini[†], A. Kostov[‡], V. Mollov[‡] and D. Selensky[‡]

[†] Centre for Software Reliability, City University, London, UK
{ptp, strigini}@csr.city.ac.uk

[‡] Department of Computing, Technical University, Plovdiv, Bulgaria
alex@obs.bg, vmollov@yahoo.com, selensky@bigfoot.com

Abstract. With off-the-shelf software, software fault tolerance is almost the only means available for assuring better dependability than the off-the-shelf software offers, without the much higher costs of bespoke development or extra V&V. We report our experience with an experimental setup we have developed with off-the-shelf SQL database servers. First, we describe the use of a protective wrapper to mask the effects of a bug in one of the servers, without depending on an adequate fix from the vendors. We then discuss how to combine the diverse off-the-shelf servers into a diverse modular redundant configuration (N-version software or N-self-checking software). A wrapper guarantees the consistency between the diverse replicas of the database, serving multiple clients, by restricting the concurrency between the client transactions. We thus show that diverse modular redundancy with protective wrapping is a viable way of achieving fault-tolerance with even complex off-the-shelf components, like database servers.

1. Introduction

The audience of this conference is well aware of the pros and cons of using off-the-shelf (OTS) software components¹. In this paper we focus on the dependability problems that OTS components pose to system integrators: their documentation is usually limited to well defined interfaces, and simple example applications demonstrating how the components can be integrated in a system. Component vendors rarely provide information about the quality and V&V procedures used. This creates problems for any integrator with stringent dependability requirements. At least in non-safety critical industry sectors, vendors often treat queries of the quality of the off-the-shelf components as unacceptable or even offensive [1]. System integrators are thus faced with the task of building systems out of components which cannot be trusted to be sufficiently dependable for the system's needs, and often are not.

¹ We use the term “components” in the generic engineering meaning of “pieces that are assembled to form a system, and are systems in their own right”. “Components” may be anything ranging from software libraries, used to assemble applications, to complete applications that can be used as stand-alone systems. We consider together commercial-off-the-shelf (COTS) and non-commercial off-the-shelf, e.g. open-source, components: the difference is not significant in our discussion. Even when the source code is available, it may be impossible to make use of it – its size and complexity (and often poor documentation) may deny the system integrator the advantages usually taken for granted when the source code is available.

As we argued elsewhere [2] fault-tolerance is often the only viable way of obtaining one's required dependability at the system level, given the use of OTS components. In this common scenario, the alternatives – improving the OTS components, performing additional V&V activities – are either impossible or infeasible without costs comparable to those of bespoke development. This situation may well change in the future, if customers with serious dependability requirements achieve more clout in their dealings with OTS component developers, but this possibility does not help system integrators who are in this kind of situation now.

Fault tolerance may take multiple forms, e.g., additional (possibly purpose-built but relatively simple) components performing protective wrapping, watchdog, monitoring, auditing functions, to detect undesired behaviour of the OTS components, prevent their producing serious consequences, and possibly effecting recovery of the components' states; or even full-fledged replication with diverse versions of the components. Such "diverse modular redundancy" seems desirable because it offers end-to-end protection via a fairly simple architecture, and protection against the identical faults that would be present in replicas within a non-diverse modular-redundant system. The cost of procuring two or even more OTS components (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. The questions, for the developers of a system using OTS components, are about the dependability gains, implementation difficulties and extra costs that they would bring for that specific system.

To study these issues, we have selected a category of widely used, fairly complex OTS components: SQL database servers. Faults in the currently available SQL servers are common. For evidence one can just look at the long list of bug fixes supplied by the vendors with every new release of their products. Further reliability improvement of SQL servers seems only possible if fault-tolerance through design diversity is employed [3]. Given the many available OTS SQL servers and the growing standardisation of their functionality (SQL 92, SQL 99), it seems reasonable to build a fault-tolerant SQL server from available OTS servers. We have developed an experimental testbed which implements a diverse-redundant SQL server by wrapping a redundant set of SQL servers, so that multiple users run their transactions concurrently on the wrapped SQL servers. We are running experiments to determine the dependability gains achieved through fault tolerance [4]. In this paper, we report on experience gained about the design aspects of building fault tolerance with these specific OTS components:

- regarding diverse modular redundancy, we consider *N-version programming* (NVP) and *N-version self-checking programming* (NSCP) – to use the terminology of [5]. In NVP, the system's output is formed by a vote on the replicated outputs. In NSCP, each diverse "version" is supposed to fail cleanly, so that anyone of the replicated outputs can be used as the system's output. Both solutions depend on guaranteed consistency between the states of the diverse replicas of the database. This problem of replica consistency, despite having been under scrutiny for a long time, is still far from being solved in general for database servers [6], [7];
- regarding protective wrapping, we have outlined elsewhere [8] the idea of protective wrapping for OTS components. Wrappers intercept both incorrect and potentially dangerous communications between OTS components and the rest of the system, thus protecting them against each other's faults. For an OTS SQL server,

the protective wrapper protects the clients against faults of the server, the server against faults of the clients, and also each client against the indirect effects of faults of the other clients.

In our design approach we assume no changes to the OTS SQL servers, since we do not have access to their internals. By necessity, therefore, our solutions are based on restricting the interaction between the clients and the SQL server(s).

2. The Experimental Environment for OTS SQL Servers

The testbed has been built in collaboration between the Centre for Software Reliability at City University, London, and the Technical University in Plovdiv, Bulgaria. It allows one to run various client applications concurrently against diverse SQL servers which use a significant sub-set of the entry-level SQL-92 language. The testbed contains a wrapper for the SQL servers, implemented as a DCOM component, accessed by the client applications. Fig. 1 shows its architecture.

The testbed was created to allow experiments with 3 functionally comparable OTS SQL servers, Oracle 8.0.5, MS SQL 7.0, Interbase 6.0. The servers can run under any operating system for which there are versions of the products used; we used Windows 2000 Professional edition for experiments with the three servers and several operating systems (Win2k, Win98 and RedHat Linux 6.0) for experiments with Interbase.

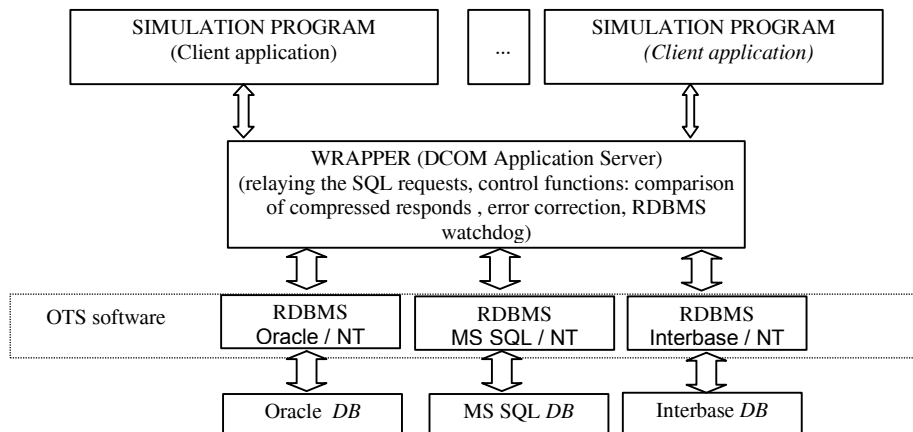


Fig. 1. Architecture of the testbed

We have experimented with between 1 and 100 clients and varying numbers of transactions per client, which include queries (SELECT) or modifications of the databases (INSERT, UPDATE, DELETE), “triggers” and “stored procedures”. We have used two applications: i) a variation of a real-life warehouse application; ii) a simplified banking application, in which funds are being transferred between accounts under the invariant condition that the total amount of funds remains constant. This invariant allows a simple correctness check (“oracle”) for whether the overall series of transactions is processed correctly by the server(s). With the “warehouse” client application, a comparison of the tables in the databases checks at predefined intervals whether the databases remain consistent, but no oracle exists to detect which of the

servers has failed in case of disagreement. A third application is under development, based on the TPC-C benchmark [9].

The testbed allows different configuration parameters to be changed such as:

- the number of clients and of queries submitted by each in an experiment;
- the “demand profiles” of the clients, a probability distribution defined on the set of queries used. The query types and parameter values are chosen by the testbed, according to user-set probability distributions. A “Template Editor” tool exists for extending the set of transactions and setting the probability distributions, so that one can experiment with a wide range of loads on the servers;
- various modes of concurrency control between the clients:
 - *Free mode*, i.e. unrestricted access to servers by all clients. The level of isolation between the transactions provided by the servers is set to “serialisable”, but no mechanism (e.g. atomic broadcast) is implemented in the testbed to control the order in which the queries are delivered to the individual servers and hence executed by the servers. The clients are multithreaded, with a separate thread talking to each of the servers.
 - *Bottleneck mode*, which imposes a very restrictive total order of the access by the clients to the server, with no concurrency between the clients. The threads representing the clients are synchronised (using critical sections) and the servers are supplied with only one transaction at a time. The next transaction (coming from any of the competing clients) is only initiated after the previous transaction is either committed or rolled back;
 - *WriteBottleneck mode*, in which the wrapper allows an arbitrary number of concurrent observing (i.e. read-only) transactions to be sent to the servers but no concurrency between the modifying transactions (which contain at least one INSERT, DELETE or UPDATE statement). A modifying transaction can only be started after the previous modifying transaction is completed (committed or rolled back).
- intervals for comparison of the tables in the databases and for “ping”-ing the servers to check whether they are still functioning.

For each experiment, a detailed log of events is recorded, including, e.g., all queries as sent, all exceptions raised, ping responses, results of database comparison, with timestamps for queries and responses.

3. Wrapping Against Known Faults in a Server

A form of fault-tolerance is to deal explicitly with known faults. We give one example here.

With the Microsoft SQL v7.0 server, we observed that when the number of clients exceeded 20, the sharing of LOCKS between the competing threads created by the SQL server to serve its clients could cease to work properly. A peculiar situation could arise in which some clients acquired the locks they needed but remained in the “waiting” state, thus keeping all the other clients (trying to acquire the same locks) from continuing (this abnormal situation is not a deadlock, which the server would detect and handle by rolling back all competing transactions but one). The problem

only occurs when the number of concurrent clients is large, and become more frequent as this number increased.

As we later found out, Microsoft reported the problem as due to a fault of the SQL server (Bug #56013, [10]).

A work-around is for the administrator - or for an application - to detect the situation and intervene by killing the thread that holds all the LOCKs but remains in a "sleeping" state (i.e. is in the root of the chain of blocked threads). However, manual intervention by the administrator is costly and may still allow large delays before being undertaken. Handling the problem explicitly in the client applications is only satisfactory if *all clients* handle the situation properly.

We have found another fully automated solution, which is relatively painless and can be incorporated in a wrapper, without changes to the legacy clients. It utilises a parameter, specific for MSSQL, LOCK_TIMEOUT, which can be explicitly set *for each query*. Its default value is 0, i.e. the blocked thread would wait for the needed lock forever. Setting it to non-zero value (we used 10 seconds) would make the server raise an exception "Lock request timeout period exceeded" when the set lock timeout expires. Now the client instead of waiting forever will get the exception and can roll the transaction back, while the locks are passed on to other clients. This solution is sufficient to resolve the occurrences of "bad blocking", at the cost of some number of transactions being rolled back. It can be improved if we include in the wrapper an exception handler for LOCK_TIMEOUTs, which would gradually increase the LOCK_TIMEOUT period, or just repeat the transaction after rolling it back, and thus make the resolution of the "bad blocking" condition completely transparent to the client. The cost of our simple solution, of course, is rolling back multiple transactions: not necessarily a high cost. The alternative - killing the thread at the top of the blocking chain - also has its cost. If a server thread is killed, the connection between the client and the server is lost and a new connection will have to be established, which is a more expensive operation than rolling back a few transactions.

It is worth pointing out that our letting the wrapper manipulate the MSSQL-specific *lock timeout* does not interfere with the setting of the *query timeout* (a different mechanism, available to the client applications with any SQL server). A complex query may take very long to complete even under light load (e.g. executing a complex query with sub-queries) and, therefore, setting a large query timeout for all queries is reasonable. During the execution of a query, multiple LOCKS can be exchanged between the server threads which compete for access to a shared resource. Without bug #56013, long query timeouts can co-exist with very short LOCK_TIMEOUT without any aborts.

With this approach of implementing work-arounds in wrappers, a *user organisation* can provide fault tolerance for bugs it discovers to be detrimental to the dependability of its installations, without waiting for the vendor to recognise the problem and issue a patch, which in any case may not completely eliminate the undesired behaviour. When known problems are left open by the vendor, the system integrator has the only choice of either introducing a protective wrapper or building a work-around in the client applications. The latter option may well be more efficient, but it is more cumbersome to manage and implement correctly: the fix and any subsequent upgrade to it must be replicated in all the client applications. Note that in our example, if some clients did not properly use the LOCK_TIMEOUT defence, they could prevent the other, "well-

behaved” clients from accessing a shared resource - forever. The well-behaved clients would be the only ones to receive multiple “Lock request timeout period exceeded” exceptions until the blocking chain of non well-behaved clients is removed somehow, e.g. by timing out the respective queries, which may take long.

In summary, implementing a fix in the wrapper reduces dependence on fixes by the vendor, and it seems *always a better option* than implementing it in the client applications, so long as feasible and the performance penalty incurred at run-time is acceptable.

4. Diverse-redundancy with SQL Servers Guaranteeing Consistency

With the testbed developed (Fig 1), we wish to answer two questions:

- are the off-the-shelf SQL servers sufficiently diverse in their failure behaviours that a diverse-redundant server would be significantly more reliable than the individual servers?
- to what extent can one build a software fault-tolerant server with OTS SQL servers without altering the internals of the servers?

Regarding the first question, preliminary work with the bug reports publicly available for two open-source SQL servers, PostgreSQL 7.0 (www.postgresql.org) and Interbase 6.0 (firebird.sourceforge.net), indicates encouraging results. We have demonstrated via manual testing that our setup can tolerate most of the bugs of the two SQL servers reported over one year. We plan to extend this work to the other SQL servers we experimented with, and to supplement it with statistical assessment of the reliability gains, via extensive automated testing under different testing profiles.

Regarding the second question, we achieve consistency between the diverse SQL servers using only synchronisation mechanisms implemented in a wrapper (Fig. 1). Achieving consistency between diverse SQL servers is difficult. All SQL servers must be guaranteed to execute the same serialisable transaction history –“1-copy serialisable execution” [11]. This is difficult for identical replicas [6] and even more so for diverse SQL servers [3]. Here we have an example of “eager replication” regarded by many as very difficult [6] and, therefore, rarely used. A recent survey of the replication mechanisms used in the leading commercial SQL servers can be found in [12]. The available solutions are for non-diverse replicas and are based on vendor specific replication mechanisms, optimised for high performance at the expense of consistency. The generic mechanisms proposed by various researchers, e.g. [13], [14], universally require access to the internals of the servers so that the replicas can achieve a consistent view on the order and the results of transaction processing. Since we use OTS SQL servers whose internals we cannot modify, none of these solutions is available to us.

Using the testbed in *Free mode* gave us plenty of examples in which the consistency between the databases was violated. Actually, this problem was exacerbated by the original implementation of our wrapper, which did not even attempt to deliver the queries within a transaction in the same order to the different SQL servers. It is worth checking whether implementing a mechanism which guarantees the deliveries of the queries to the servers in exactly the same order would have an impact on the evolution of the databases. In any case it would not avoid inconsistencies, since different servers use different optimisation strategies, and may alter the order of the execution of the

concurrent queries, compared to the order in which they are delivered to the server, in different ways.

In summary, we need to constrain concurrency to guarantee consistent evolution of the diverse databases. What are the non-intrusive (i.e. not altering the internals of the SQL servers) options available? An obvious option is to eliminate concurrency completely, via the *Bottleneck mode* of operation of our wrapper: a single query at a time is executed, guaranteeing that all servers will execute the *same serialisable history* (*1-copy serialisability*). Provided the servers process the query correctly, the databases will stay consistent. With tests with the *Bottleneck mode* of operation, consistency was indeed preserved². As a by-product of using this mode, deadlocks between clients are eliminated. However, the limitations imposed are so restrictive that this mode is hardly of any interest. SQL servers are designed to provide high throughput under a wide range of circumstances, including heavy load and thousands of concurrent transactions, benefits which are denied in the *Bottleneck mode*.

Our alternative is to use the *WriteBottleneck mode*, in which many read transactions can be executed concurrently together with *at most one write transaction*. The reason for implementing this is that in most real-life applications of databases, most transactions are *observing transactions* (i.e. SELECT queries). This is particularly true for most web applications. In the extreme case of some large web-based databases, the only on-line operations are SELECTs, while updates are only run off-line. In the *WriteBottleneck mode*, the *changes occur in the same order* for all databases, guaranteeing the consistency of the changes across the servers. The transaction histories on the servers may only differ in the order of their read transactions. This may lead to inconsistent results returned by the read transactions: voting on the outputs from the servers may produce mismatches even if the servers work correctly. Voting is, thus, no longer a trustworthy error-detection mechanism. Without voting, this solution is only fully fault-tolerant so long as the database servers have a fail-silent or crash-fail semantics [15] - which is, on the other hand, still assumed by most developers of database applications.

The *WriteBottleneck mode* also allows two more standard tricks for improving the throughput of the read transactions:

- i) the wrapper can return to the client the first among the redundant results from a SELECT query: diversity will bring some performance improvement if different servers perform best on different queries;
- ii) the wrapper can perform load balancing for read transactions by forwarding queries to only some servers, and thus reduce the load on the individual servers and increase overall system throughput. This may compensate for the delays on write transactions due to the *WriteBottleneck mode*.

The *WriteBottleneck mode* eliminates, as a by-product, all problems caused by concurrent execution of write operations by the servers. A recent study reports that SQL servers employing snapshot isolation, e.g. Oracle, have trouble handling the “write skew” problem [16]. This problem, first described in [17], is as follows. Suppose that

² We always explicitly tested the effects of any solution we designed. This experimental confirmation is important, even if the solution, as specified, can be proven to have a desired property, and even if the solution is correctly implemented in the wrapper, because its operation relies on the functioning of the SQL servers - whose internal details the integrator cannot verify - and thus may violate that property.

X and Y are data items representing bank balances for a married couple, with the constraint that $X+Y>0$ (the bank permits that either account be overdrawn as long as the sum of the account balances remains positive). Snapshot isolation is reported to have a problem with two transactions which concurrently attempt to withdraw from the two accounts, X and Y. It is possible to commit both transactions and leave the accounts with the constraint $X+Y>0$ violated. The *WriteBottleneck mode* clearly eliminates the problem. SQL servers using snapshot isolation will work properly if accessed via a wrapper in *WriteBottleneck mode* of operation. If this mode of operation is acceptable, there is no need for changes of the client applications as proposed in [16] to guarantee that a set of sufficient conditions for serialisability are met.

The expectation to see the diverse databases stay consistent under the *WriteBottleneck mode* has also been confirmed by testing with a few millions of transactions on the three diverse SQL servers of Fig 1.

To conclude, the *WriteBottleneck mode* is less restrictive than the *Bottleneck mode* and has *some practical relevance*, since its performance penalty may be negligible. Predictably, in our tests the performance of the testbed (measured by the time to execute a set of transactions) was better in *Free* than in *Bottleneck mode*. However, with a light load of write queries the *WriteBottleneck* and the *Free modes* are comparable. Of course, as the load of write queries increases the performance under the *WriteBottleneck* decreases, approaching the performance of the *Bottleneck mode*.

The *Bottleneck mode* has the advantage of allowing voting for error masking, at the cost of severe performance limitations. The *WriteBottleneck mode* reduces the latter disadvantage at the cost of making voting either unusable or more expensive (e.g., if discrepancies are rare enough they can be handled by aborting and retrying the transactions involved).

5. Related Work

Replicated databases are common, but most designs are not suitable for diverse redundancy. We have cited in the previous section some of the solutions proposed. Recent surveys exist of the mechanisms for eager replication of databases [7], and for the replication mechanisms – mainly lazy replication – implemented in various SQL servers [12]. The Pronto protocol [13] 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. We have observed that the syntax of the SQL statements varies between commercial SQL servers, even for a standardised set of statements. This would create a problem with diverse SQL servers and would require translating the statements into the SQL “dialects” spoken by the diverse SQL servers involved in the replication, which inevitably will slow the replication down.

In software development, the idea of wrapping at various levels has been exploited as a means of making a particular function compatible with the component framework used, e.g. COM (DCOM), Java RMI, CORBA, etc.

Protective wrapping has received some attention recently [18], [19]. In [20] we described an approach in which protective wrappers are seen not only as a means of tolerating *known bugs* but also as a means of correcting *suspicious system behaviour*.

The ideas proposed in [2] are also enjoying some popularity for the purpose of security (intrusion tolerance). For instance, HACQIT (Hierarchical Adaptive Control of Quality of service for Intrusion Tolerance) [21], uses diverse off-the-shelf web-servers to detect failures (including maliciously caused ones, like defacement of web content) and initiate recovery. [22] proposes an architecture with multiple, diverse, COTS application servers. The Cactus [23] and SITAR [24] architectures support diversity among application modules to enhance survivability.

6. Conclusions

Software fault tolerance, in the form of checker/monitor components (protective wrapping) or diverse modular redundancy (N-version or N-self-checking systems) recommends itself as a cost-effective solution for improving the dependability of off-the-shelf software [2]. However, it may be objected that it is only feasible with simple software components, and is thus of very limited applicability. Our experience gives some evidence against this pessimistic view. We showed that simple wrapping techniques allow reasonable protection against design faults of the OTS SQL servers or their clients. In the case of the diverse-redundant configuration, by accepting some loss of efficiency we achieve a fault-tolerant SQL server at the moderate cost of multiple OTS SQL servers. The constraints imposed to guarantee consistency do limit performance, but this penalty may still often be minor compared to that of either using an undependable database or producing a highly trustworthy server. We have not yet estimated the dependability improvement produced by the diverse-redundant configuration, but we have evidence that it tolerates most of the known faults of the SQL servers used, which is a promising first piece of evidence. We intend to measure the potential dependability gains on long runs of test cases under different loads.

There are several possible extensions of our research since our implementation of a software fault-tolerant SQL server is only a demonstration prototype. Our goal in building it was to measure the *potential dependability gains* offered by diversity, not to build a full-fledged fault-tolerant server. For instance, we have not implemented an efficient solution for recovery of a database that is found to be corrupted. Scalability too, a very important aspect of connectivity with SQL servers, is not paid adequate attention, yet.

Acknowledgement

This work was supported in part by the DOTS (Diversity with Off-The Shelf components) project funded by the Engineering and Physical Sciences Research Council of the United Kingdom. Authors would like to thank all colleagues from the Centres for Software Reliability at City University and Newcastle Upon Tyne, UK, involved in the DOTS project and in particular Dr Alexander Romanovsky for the helpful comments on an earlier draft of this paper.

References

- [1] ECUA, "3rd European COTS User Working Group (ECUA) Workshop," in *Panel with Industrial Collaborators*. Copenhagen, Denmark, 2002.
- [2] P. Popov, L. Strigini, and A. Romanovsky, "Diversity for off-the-Shelf Components," presented at International Conference on Dependable Systems and Networks (DSN 2000) - Fast Abstracts supplement, New York, NY, USA, 2000.
- [3] J. Gray, "FT101: Talk at UC Berkeley on Fault-Tolerance", 2000, pp. 62 slides, http://research.microsoft.com/~Gray/talks/UCBerkeley_Gray_FT_Availability_talk.ppt.
- [4] P. Popov and L. Strigini, "Diversity with Off-The-Shelf Components: A Study with SQL Database Servers," presented at International Conference on Dependable Systems and Networks (DSN 2003) - Fast Abstracts supplement, 2003.
- [5] J. C. Laprie, J. Arlat, C. Beoune, and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures," *IEEE Computer*, vol. 23, pp. 39-51, 1990.
- [6] J. Gray, P. Helland, D. Shasha, and P. O'Neil, "The Dangers of Replication and a solution," presented at ACM SIGMOD International Conference on Management of Data, Montreal, Canada, 1996.
- [7] M. Weismann, F. Pedone, and A. Schiper, "Database Replication Techniques: a Three Parameter Classification," presented at 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00), Nurnberg, Germany, 2000.
- [8] P. Popov, L. Strigini, S. Riddle, and A. Romanovsky, "Protective Wrapping of OTS Components," presented at 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction, Toronto, 2001.
- [9] TPC, "TPC-C, An On-Line Transaction Processing Benchmark, v. 5.," 2002.
- [10] Microsoft, "MS SQL 7.0, BUG #: 56013, FIX: Lock Conversion Processing Does Not Properly Wakeup Lock Waiter", <http://support.microsoft.com/default.aspx?scid=kb;EN-US;236955>, 2002.
- [11] A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, Mass.: Addison-Wesley, 1987.
- [12] A. Vaysburd, "Fault Tolerance in Three-Tier Applications: Focusing on the Database Tier," presented at 18th IEEE Symposium on Reliable Distributed Systems (SRDS'99), Lausanne, Switzerland, 1999.
- [13] F. Pedone and S. Frolund, "Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases," presented at 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00), Nurnberg, Germany, 2000.
- [14] F. Pedone, R. Guerraoui, and A. Schiper, "Transaction Reordering in Replicated Databases," presented at 16th IEEE Symposium on Reliable Distributed Systems (SRDS'97), Durham, NC, 1997.
- [15] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computing Systems*, vol. 1, pp. 222-238, 1983.
- [16] A. Fekete, D. Liarakapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making Snapshots Isolation Serializable," 2000, pp. 16.
- [17] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels," presented at SIGMOD International Conference on Management of Data, 1995.
- [18] M. C. Mont, A. Baldwin, Y. Beres, K. Harrison, M. Sadler, and S. Shiu, "Towards Diversity of COTS Software Applications: Reducing Risks of Widespread Faults and Attacks," HP Laboratories, Bristol, UK HPL-2002-178, 2002.
- [19] A. Romanovsky, "Exception Handling in Component-Based System Development," presented at COMPSAC'01, Chicago, IL, 2001.
- [20] P. Popov, L. Strigini, S. Riddle, and A. Romanovsky, "On Systematic Design of Protectors for Employing OTS Items," presented at 27th Euromicro Conference, Workshop on Component-Based Software Engineering, Warsaw, Poland, 2001.
- [21] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, and K. Levitt, "The Design and Implementation of an Intrusion Tolerant System," presented at International Conference on Dependable Systems and Networks (DSN 2002), Washington, D.C., USA, 2002.
- [22] A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saidi, V. Stavridou, and T. E. Uribe, "An Adaptive Intrusion-Tolerant Server Architecture," 1999.
- [23] M. A. Hiltunen, R. D. Schlichting, C. A. Ugarte, and G. T. Wong, "Survivability through Customization and Adaptability: The Cactus Approach," presented at DARPA Information Survivability Conference & Exposition, 2000.
- [24] F. Wang, F. Gong, C. Sargor, K. Goseva-Popstojanova, K. Trivedi, and F. Jou, "SITAR: A Scalable Intrusion-Tolerant Architecture for Distributed Services," presented at IEEE Workshop on Information Assurance and Security, West Point, NY, U.S.A, 2001.