A Closer Look at Database Replication Middleware Architectures for Enterprise Applications *

J.E. Armendáriz-Iñigo[1], H. Decker[1], F.D. Muñoz-Escoí[1], J.R. G. de Mendívil[2]
[1] Instituto Tecnológico de Informática, Campus de Vera, 46022 Valencia, Spain
[2] Universidad Pública de Navarra, Campus Arrosadía, 31006 Pamplona, Spain
{*armendariz, hendrik, fmunyoz*}*@iti.upv.es, mendivil@unavarra.es*

Abstract. Middleware database replication is a way to increase performance and tolerate failures of enterprise applications. Middleware architectures distinguish themselves by their performance, scalability and their application interface, on one hand, and the degree to which they guarantee replication consistency, on the other. Both groups of features may conflict since the latter comes with an overhead that bears on the former. In this paper, we review different techniques proposed to achieve and measure improvements of the performance, scalability and overhead introduced by different degrees of data consistency. We do so with a particular emphasis on the requirements of enterprise applications.

1 Introduction

Although middleware-based replication has been widely discussed in the literature for quite some time [1,2,3], such architectures are only recently emerging as a promising approach to raise the performance and availability of web services for enterprises operating across geographically distant sites [4]. Apart from the usual delay of technological innovations to gain commercial appeal, this is mainly due to two factors: the lack of support from established DBMS vendors, and unsatisfactory solutions for the key challenge of guaranteeing a sufficient degree of consistency and up-to-dateness of replicated data for enterprise applications.

We are confident that the database industry will sooner or later buy into middleware replication technology, since otherwise, the high availability and performance of replicated servers equipped by different vendors would remain an untapped bounty. Hence, we deal in this paper with the remaining issue, viz. the provision of adequate consistency guarantees that are tuned to the requirements of enterprise applications, which may vary from case to case.

There are two canonical alternatives to achieve database replication: by extending the DBMS core code [5,6,7,8], or using a middleware layer [1,2,3,9,10,11]. The former has an immediate performance advantage due to a low overhead, but is vendor-specific and hence handicapped in terms of system interoperability, application portability and migration to new versions or other installations [6,7]. The latter is vendor-independent, hence it is straightforwardly interoperable, facilitates the portability of applications and is easily mirated. It may compensate its higher overhead by an elegant use of built-in SQL constructs, such that concurrency and transaction control is delegated back

* supported by the Spanish government research grant TIC2003-09420-CO2.

to where it belongs, i.e., the DBMS core. Enterprise applications clearly benefit from vendor-independent middleware solutions, on which we therefore shall focus in the remainder.

Given the DBMS core vs middleware tradeoff as highlighted above, overhead reduction is clearly the prime goal for middleware architectures. Early solutions have been encumbered by concurrency control tasks that natively belong to the underlying DBMS, and by the necessity to modify given enterprise applications in order to support the middleware's replication management [1,12]. Both issues have rightfully been perceived as burdens or even bugs, rather than features [9]. Further development has then brought forward standardized application interfaces, usually based on JDBC, that offered predefined procedures for web applications (e.g., automatized form dialogues) to be called through applications [2].

This move towards application independence is taken a step further by permitting applications to execute any kind of statement. Concurrency control then is delegated to the underlying DBMS while consistency management is left to a given replication protocol, as in RJDBC [11] and C-JDBC [10]. However, both solutions suffer from scalability problems, since an update statement must be executed at all available network nodes before the next statement can be executed. Scalability is enabled in the MADIS architecture [3], which realizes concurrency delegation exclusively by using SQL constructs, and in MIDDLE-R [2,9], which, apart from also using SQL statements, also uses the write-ahead log to propagate updates.

Applications access the distributed system data transparently by way of transactions. All accessible data are persistently stored in an underlying DBMS, the replication of which is hidden from the applications' interface. The transaction isolation level provided by the DBMS co-determines the degree of consistency obtainable in replicated systems. Most commercial DBMSs provide Snapshot Isolation (SI) [13]. It relaxes serializability and thus gains in performance, while concurrency anomalies that cannot easily be worked around are avoided by weakening the SQL-92 standard's definition of isolation levels [13]. This clearly amounts to an immense advantage for web-based applications, since read operations never get blocked with SI [14].

A de-facto standard notion for ensuring transaction correctness in replicated databases is One-Copy Serializability (1CS) [15]. Essentially, it means that the interleaved execution of transactions must be equivalent to some sequential execution. In [14,16], a theory for achieving 1CS using SI has been developed. Moreover, the notion of SI has been extened to Generalized SI (GSI) [14] and One-Copy SI (1CSI) [9], thereby clarifying the notion of *system snapshot* in the context of replicated systems.

The overhead introduced by a middleware architecture is of course determined by its design and implementation, i.e., by the underlying DBMS, the manner by which messages are propagated, how transaction operations are intercepted and managed, etc. However, besides such comparatively superficial technicalities, the overhead is perhaps influenced most crucially by the deployed kind of replication strategy [17,18]. Replication is usually classified by the orthogonal distinctions of: eager or lazy update propagation; executing updates in a dedicated "primary copy" node or permitting update execution everywhere in the network; the degree of communication among sites which may be of constant or linear interaction; and, whether transaction terminate either voting

or non-voting. These binary classification criteria span a space of replication protocols with combinations of properties that can be tailored to the specific needs and requirements of various kinds of enterprise applications, that differ with regard to latencies, abortion rates and overhead/performance tradeoff.

In this paper, we propose different transaction correctness criteria in replication architectures that may be satisfied by given enterprise applications. Correlated to the overhead introduced by the architecture, we also review different performance measures. In particular, we review metrics proposed for MIDDLE-R and metrics developed by the DBMS community such as TPC-W [19], which also serves as a benchmark for web transactions. The workload for which the performance is measured is generated in the framework of a controlled internet commerce environment that simulates the activities of a transactional web-based enterprise server.

The rest of the paper is organized as follows: Section 2 paradigmatically describes the main components of a prototypical middleware-based database replication architecture. Its interaction with client applications is described in section 3. System performance is discussed in section 4. Section 5 concludes the paper.

2 System Model

Figure 1 shows a typical generic configuration of a replicated database middleware architecture. It is composed by N nodes which communicate among each other via message exchanges. for that, they use a group communication subsystem (GCS), which provides a membership service, i.e., knowledge about alive and broken nodes [20]. Applications submit transaction requests to the system. The database replication middleware (DRM) intercepts these requests and manages the execution of remote transactions at all DBMS replica. The replication protocol, embedded inside the DRM, coordinates the execution of transactions at all nodes in order to ensure data consistency [9,14,15]. Concurrency control, however, is delegated to the DBMSs.

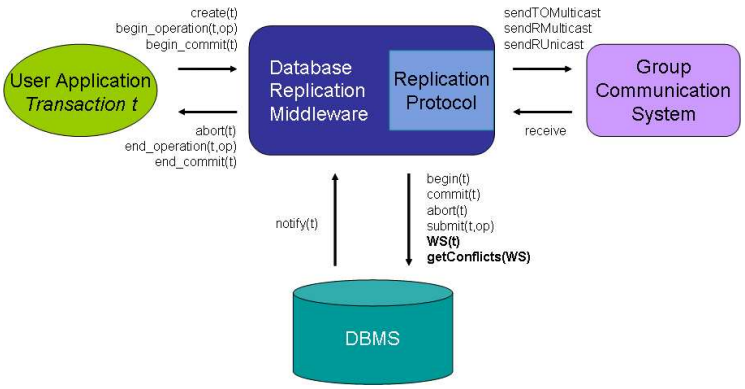

Fig. 1. Sample node of a replicated database middleware architecture.

2.1 Protocols for Transaction Execution

The choice of an appropriate replication protocol to execute transactions should be guided by the classifications discussed in [17,18]. In fixed, tighly connected networks, eager update-everywhere replication is the preferred kind of protocol. Transactions are firstly executed at the node which is closed to where they were requested. Updates are then regrouped and sent to the rest of nodes. Once they are delivered, usually in total-order form, the commit phase is started, either with some coordination among nodes (such as in the 2-Phase-Commit protocol (2PC) [15] or without node interaction but with a certification phase, i.e., a test that decides if a transaction may or may not commit [7,9,14,15]. In loosely connected networks, lazy update propagation is common. Then, serialization conflicts may arise much more frequently. Related reconciliation processes can be automatized, either by the replication protocol or by notifying the involved applications. In general, it is always necessary to take care of possibly conflicting transactions [21] and to re-attempt execution of successfully certified transactions until they have been scheduled and committed [9]. This cab be accomplished by the DRM, along with schema modifications and stored procedures as described in [3].

2.2 Group Communication

As already indicated, virtual synchrony [20] is supported by a GCS which supports communication and membership services. For the model sketched in fig. 1, we assume a system with partial synchrony, a communication service with reliable multicast for message exchange, and a *partial amnesia crash* [22] failure model. The latter assumption is convenient since we are going to deal with node recovery after failure. Further, we adopt the notion of a *view*, as provided by the membership service of GCS, to mean a maximal set of interconnected alive nodes that strive to have mutually consistent data (rather than a virtual table defined by a query). Any change in the composition of a view by entry or exit of a node is supposed to be reported to the recovery protocol. We assume a primary component membership [20], guaranteeing that no node belongs to two different views at any time, and for each pair of consecutive views, there is at least one node that remains alive in both. Further, *strong* virtual synchrony be used, to ensure that messages are delivered in the same view they were multicast, and that two nodes exiting view v_1 for entering view v_2 have delivered the same set of messages in v_1 [20,23].

2.3 The Underlying Database System

At each node, the DBMS stores a physical copy of the replicated database. It executes transactions as specified by the given replication protocol, guaranteeing the well-known ACID transaction properties. With regard to a given node, a transaction is either local or remote, i.e., its write set may originate from a transaction requested at some other node. Moreover, the DRM may modify the database schema, so as to support the automatized storage of replication metadata in additional as well as the triggering of stored procedures to process these metadata.

Instead of the previously mentioned, yet outdated default 2PC locking protocol, commercial DBMSs nowadays usually support Snapshot Isolation (SI), i.e., a kind of multiversion concurrency control [15]. An SI-controlled transaction always reads data from a snapshot of the meanwhile committed data as of the time the transaction started. It is never blocked from attempting reads as long as its snapshot data is still valid. The writes (updates, inserts and deletes) of a transaction *T* will also be reflected in this snapshot, to be consulted again if *T* reads or updates the data another time. Updates by other transactions active after *T* was initiated are invisible to *T*. However, a serializable isolation level can be achieved by using SI, as shown in [14,16]. Different isolation levels achieved by protocol strategy variants 1CS [15] (for serializable) and GSI [14], 1CSI [9] for SI are thereby classified into two families of consistency protocols for replicated databases. They will be outlined in more detail in section 3.

2.4 Replication Support

A by now tried-and-tested manner of reducing the overhead of replication protocols it to store and process transaction metadata in the underlying DBMS. However, middleware architectures differ in the way the transaction data are collected and transferred. In MADIS, the transaction report is built by DRM-generated triggers and stored procedures. Although this option is advantageous because recurs on nothing but standard SQL, it involves more write accesses to the database than necessary in case the needed information is obtained from the write-ahead log, which is provided in most, if not all DBMSs. This latter option saves in write operations, as observed in [2] [9].

The latter paper also pointed out that replication protocols using certification may suffer from aborts caused by the DBMS while applying the write set of already certified transactions. A remedy would be to detect conflicting local transactions. Below, we describe a technique for managing concurrency control which combines the simplicity of using DBMS core support with maintaining the product independence of a middleware solution. Instead of having to request and wait for the termination of transactions, conflicting transactions may be aborted immediately. By reducing the abortion delay, the system becomes ready faster for processing other active transactions. We have implemented and tested our approach in PostgreSQL. Our solution needs to scan the system's locking tables. Similar tables are used in virtually all DBMSs (e.g., the `V$LOCK` view in Oracle 9i, the `DBA_LOCK` in Oracle 10g r2, the `sys.syslockinfo` table of Microsoft SQL Server 2000 - converted into a system view in SQL Server 2005 -, etc.), so that this solution is easily portable to all of them, since only standard SQL constructs are used.

Serializability may be obtained also with SI [14], either by modifying the application or considering the readset of transactions. As read-only transactions are only executed at the site they are submitted to, we assume that they read data from a snapshot, and hence do not need to be isolated with regard to update transactions. As a result, it seems appropriate to design a mechanism that notifies write-write conflicts of transactions to the replication protocol. As for conflict detection, the main advantage of our approach is the use of the concurrency control support of the underlying DBMS. Only he system's locking tables need to be scanned for that, so that, again, seamless portability is warranted, since only standard SQL constructs are used. Thus, the middleware is enabled to provide row-level control (as opposed to the usual coarse-grained

table control), while all transactions (even those associated to remote write sets) are subject to the underlying concurrency control support. Its implementation is based on the following two elements:

- The database schema is enhanced by the stored function `getBlocked()`. It looks up blocked transactions in the DBMS metadata (e.g., in the `pg_locks` view of the PostgreSQL system catalog). It returns a set of pairs consisting of the identifiers of a blocked transaction and of the transaction that has caused the block. If there is no conflict when this function is called, it returns the empty set. In short, `getBlocked()` reads a system catalog table in which the DBMS keeps information about transaction conflicts. Such a table is maintained by most DBMSs. Thus, this function is easily portable to most of them. Moreover, these DBMSs only provide read access to this system table. So, reading such views or tables does not compromise the regular activity of the DBMS core nor the activity of other transactions.
- An execution thread per database is needed that cyclically calls `getBlocked()`. Its cycle is configurable and is commonly set to values between 100 and 1000 ms. It runs on the middleware layer. Once this thread has received a non-empty set of conflicting pairs of transactions, it may request the abortion of one of them. For this purpose, each transaction has a priority level assigned to it. By default, it aborts the transaction with smaller priority but takes no action if both transactions have the same priority level.

This mechanism should be combined with a transaction priority scheme in the replication protocol. For instance, we might define two priority classes, with values 0 and 1. Class 0 is assigned to local transactions that have not started their commit phase. Class 1 is for local transactions that have started their commit phase and also for those transactions associated to delivered write sets that have to be locally applied. Once a conflict is detected, if the transactions have different priorities, then the one with the lowest priority will be aborted. Otherwise, i.e., when both transactions have the same priority, no action is taken and they remain in their current state until the lock is released. Similar, or more complex approaches might be followed in other replication protocols that belong to the update everywhere with constant interaction class [18].

2.5 Replication Middleware

The DRM is the core of the middleware system. It is independent of the underlying DBMS. In [3], we have described a Java implementation, to be used by client applications as a common JDBC driver. It facilitates the plugging and swapping of replication protocols chosen according to given needs and requirements. DRM may act as an interceptor for applications the transactions of which are executed locally at the nearest alive node, while monitoring remote update messages coming in via the GCS. For example, when a commit statement is issued, an eager protocol will start the commit process right away, interacting with the rest of the nodes by transferring the transaction updates. Hereafter, a 2PC protocol or a certification process starts to globally commit the transaction.

3 Middleware Layout for Enterprise Applications

The replication middleware architecture as outlined in section 2 is of generic character. In this section, we are going to specialize its layout with regard to typical requirements marked out by applications of enterprises the operation topology of which covers multiple branches. Needless to say, the enterprise branches and operation areas are supposed to be distributed over geographically disparate locations. In particular, we are going to touch upon application interfaces, load balancing, fault tolerance and tailoring the middleware with regard to different degrees of consistency, as required by different application profiles.

3.1 Interfaces for communication and coordination

Most middleware systems that comply with the model as characterized in section 2 export a standard JDBC interface [2,3,10,11]. Hence, applications that alreay exist do not have to be re-programmed for becoming usable on top of the middleware. In particular, this means that they can remain completely unaware of the underlying replication, conforming to the ideal of full transparency. Systems such as MADIS, MIDDLE-R, CJDBC and RJDBC merely act as interceptors for invocations performed by clients. Local transactions are forwarded to the underlying DBMS by means of SQL statements. Commit invocations initiate interaction with the rest of nodes. A ROWAA approach is assumed, with eager update-everywhere replication protocols [17].

Each node has to apply remote transactions coming from other nodes, which mainly consist of writesets of committing transactions. Such remote transactions may abort current active local transactions in case the latter causes any conflict. Then, the middleware must notify local transactions about that. The amount of such transaction abortions stronly depends on the degree of consistency as required by the application and on the deployed replication protocol. These abortions have to be done transparently, so that the application perceives them as database rollbacks instead of protocol-driven aborts.

3.2 Load Balancing

Transactions may be redirected to the least loaded node or, more generally, where network conditions are optimal. The node to which the transaction is (re-)directed and initially executed is called the *local replica*. Locality of data access supports the decrease of transaction response time. That, however, also depends on the chosen replication protocol (we shall come back on this in subsection 3.4). Anyway, read operations are always performed on the local node, and no interaction with remote nodes is needed.

Enterprise applications can be classified as services for either internal or external purposes. Internal applications typically are intranet enterprise applications, e.g., IT-based collaboration between different business units, or knowledge management, which is open for internal use but hidden to the outside world. Typical external applications are extranet services, provided via an enterprise web portal to customers and clients. A good intranet replication policy for intranet application is to replicate the database at each site. Such configurations can be likened to peer-to-peer applications. On the other hand, for data replication of extranet enterprise services means that external users access

a virtual database which does not belong to their own site. Thus, extranet users behave as clients of a virtual server which actually is a transparently distributed system the high availability, performance, fault tolerance and dependability of which is supported by a transparent replication architecture. This has been outlined in some more detail in [3].

3.3 Fault Tolerance and Recovery

It is important that recovery of crashed or disconnected nodes is fast and does not block the whole system [26]. This permits to alleviate alive nodes, as more nodes will recover the replicated database state and will be able to attend new incoming transactions. Structurally the same situation is faced with new nodes joining the distribution topology on an ad-hoc basis. Most database replication approaches include a recovery protocol for crashed nodes. In our model architecture, applications may be redirected to an alive node in case its local replica fails. As long as the application is served in a primary partition, it will be able to continue its execution. Thus, the high availability of application data is assured. Moreover, thanks to virtual synchrony, a transaction may finally commit even if the serving node of that application crashes during its execution.

3.4 Consistency Levels

Enterprise applications cover a very wide range of different scenarios: Examples can be given that range from grocery chain websites, over web collaboration between customer and e-service centres, to online flight booking and scheduling. A common denominator, however, is that they all share the need to work with consistent, up-to-date data. However, consistency requirements may vary from application to application. For example, web-based flight booking is in need of accurate 24/7 up-to-dateness of data about available flights and seats transactional access with different levels of data consistency. On the other hand, the reporting service applications of interactive data warehousing of grocery chains certainly won't need to take into account the latest updates of the day for their quarterly or annual stock-keeping statistics. With regard to choosing, plugging and swapping of appropriate replication protocols for adapting to changing consistency requirements This has been looked at in more detail in [3].

Applications may manipulate stored data by SQL statements, including stored procedures. The set of data items read or written by a stored procedure can be anticipated at schema specification time. In particular, the related accesses may be adjusted so as to guarantee a serializable behavior, which clearly is more advantageous than to rely at execution time on the SI provided by the DBMS. Hence, for each stored procedure, a dedicated node can be determined as the owner of that procedure at the time it is compiled. With the knowledge about which data are accessed upon execution of the stored procedure, its owner node can anticipate control strategies and execution plans so as to avoid unnecessary conflicts. This has been looked at in more detail in [2]. In general, however, the data access patterns of user-driven applications are ad-hoc, i.e., unknown in advance. Hence, special attention has to be paid to intersecting read- and writesets of transactions in order to achieve a given level of consistency. The following item points distinguish three characteristic and often encountered levels of consistency

requirements, to be adopted for different enterprise applications with corresponding consistenmcy requirements.

- **1CS**. This is the strongest correctness criterion for replicated databases. Replication is transparent to the execution of a transaction. Its interleaved execution among other transactions in the system is equivalent to a serial execution of the transactions in a centralized database. This was introduced in [15] and it supposed that all underlying DBMS were implemented using 2PL.
  This data consistency level is appropriate for applications interested in reading the latest version of data, ensuring that no other transaction will modify the value read until the transaction commits. However, this isolation level does not prevent from missing concurrent insertions that verify the "`WHERE`" clause in a "`SELECT`" statement.
- **GSI**. This concept has been proposed recently [14], in order to provide a suitable extension of conventional SI for replicated databases. In GSI, transactions may use older snapshots instead of the latest snapshot required in SI. In [14], an impossibility result is stated which justifies the use of GSI in database replication: "there is no non-blocking implementation of SI in an asynchronous system, even if databases never fail". In a non-blocking replication protocol, transactions can start at any time without restriction or delay (even those delays produced by group communication primitives).
  This data consistency level is appropriate for generating dynamic web content. It is typically generated by a combination of a front-end web server, an application server and a back-end database. The possibly dynamic content of the web site is stored in the database of the site's host server. The application server provides methods that implement the business logic of the application. As part of that, the application typically accesses the database. The three servers (web, application and database server) may all execute on a single machine, or each one of them may execute on a separate machine or on a cluster of machines, or various combinations thereof.
- **1CSI**. Simultaneously to the GSI definition, 1CSI was introduced in [9]. It can be viewed as the counterpart of 1CS with serializable SI. A transaction uses the "*latest*" system snapshot which may imply blocking certain read operations as they are not going to see the latest snapshot. A new snapshot version is installed in the system as soon as the transaction installing the new version is firstly committed at any node. Hence, one of the main advantages of SI, viz. non-blocking read operations, is lost.

4 Performance

What makes attractive for user applications the use of a given database replication middleware is its performance. We do not pursue DBMS-core solutions any further, although they will always tend to be somewhat better [7]. But the advantage of our middleware solution is to be independent of the underlying database and to be easily portable to other DBMSs.

The standard way to measure the performance of an application is its transaction response time, i.e. how long it takes to commit a transaction in the system. Measuring

the behavior of the system may be done by checking the scalability and overall response time for our application. It is easy to see that the performance of the solution will depend on the kind of applications considered as benchmarks, such as the TPC-W [19] standard benchmark.

4.1 Scalability through Response Time

We may analyze the scalability and overall performance of the algorithms and the implementation we propose. Moreover, we may study the overhead introduced by the middleware and the GCS. It is important to emphasize that the absolute values of the results are only meaningful to a certain degree. They could be improved by simply using faster machines or by using a different DBMS. The important aspect of these results is the trends they show in terms of behavior as the number of sites and the load in the system increases.

4.1.1 Comparison with Traditional Distributed Locking A first question that needs to be addressed is whether the middleware we propose really solves the limitations of conventional replication algorithms (e.g., those described in [15]). Gray et al. [17] showed that these conventional algorithms do not scale and, in particular, that increasing the number of replicas would increase the response time of update transactions and produce higher abort rates. We have compared the scalability in terms of response time of our solution with the standard distributed locking implementation of a commercial product, Oracle. The test scenario is fixed to a model of update transactions with the same and fixed number of updates and repeating this pattern of transaction as the number of sites increases [2].

4.1.2 Throughput Scale-out The main motivation for this work is to provide a replication algorithm that can scale in a cluster based system. Some approaches [2,9,21] have to execute all update transactions at all sites since they do not have any additional knowledge about the database system. As a result, adding new sites in an update intensive environment might help for fault-tolerance, but cannot be used to scale up the system. Using alternative approaches [1], it may be more feasible achieve both fault-tolerance and scalability. Hence, this experiment analyzes how the throughput scales up when we increase the number of sites. It will be very interesting to run three sets of tests: read only, write only, and a mixture of both workloads. The scale-out for a given number of nodes (which is varied in determined range) is obtained as dividing the maximum throughput that can be achieved in this setting by the maximum throughput in a single-site system.

4.1.3 Response Time Analysis This is pretty similar to the previous point. The same set of transactions acts as a benchmark and we analyze the response time behavior by increasing the load, and determines at which throughput the system saturates (i.e., response times deteriorate).

4.1.4 Communication Overhead When using group communication primitives, the system built can only scale as much as the underlying communication tool. One of the typical problems of conventional replication algorithms is that they easily overload the network by generating too many messages (e.g., distributed locking generates one message per operation per transaction per site; a 10 site system running transactions of 10 operations at 50 transactions per second generates 5,000 messages per second). Hence 2PC protocols require two multicast messages per transaction whilst certification-based protocols need only one message.

4.2 Transactional Web E-Commerce Benchmark TPC-W

TPC-W [19] is a standard tool proposed by the research and industrial community to measure the performance of DBMSs. It simulates the behavior of users accessing an online bookstore where they search and buy books. TPC-W has been used by [7,9,14] so as to measure the performance of their different replicated data solutions which are implemented over SI DBMS replicas. TPC-W has different purchase behavior options, but the selection criteria of these options are exclusively based on randomly generated data.

However, although characterizing consumer behavior is a difficult task, the way consumers behave cannot be said to be totally defined by a random pattern. That is why marketing research has tried to describe consumer behavior patterns so as to help firms and practitioners to develop better marketing strategies. Nevertheless, consumer behavior online does not necessarily mimic that observed at physical stores, which has been the one traditionally analyzed. There are important differences between physical and virtual stores that make consumers behave differently. For instance, the amount and quality of the information that is available at each channel, the perceived risk or the possibility of using or not personal purchase lists are not the same at these two shopping environments [27,28,29]. The amount of information available online is said to be extremely high, but this information is always visual, neither tactile nor from the sense of smell [28,30]. So, for products such as towels, in which softness is a valued characteristic, online shopping seems not to be the best shopping option. Thus, a consumer purchasing a towel online will probably show a different behavior than that the one he would have had at a physical store -it would be logic to think that he could be more loyal to some brand he had previously purchased-.

TPC-W is a transactional web benchmark designed to evaluate e-commerce systems. It specifies a workload that simulates the activities of an online bookstore. Three separate components take part in the interaction: The System Under Test, SUT, comprises all components which are part of the application being simulated; the Remote Browser Emulator, RBE drives the TPC-W workloads creating an Emulated Browser, EB, for each user interacting with the system; and the Payment Gateway Emulator, PGE, represents an external system which authorizes payment of funds.

TPC-W specifies 14 different web pages which must be implemented in the SUT. Moreover it defines the schema of the database where data will be stored. There are described 8 tables: `CUSTOMER`, `ORDER`, `ADDRESS`, `COUNTRY`, `ORDER_LINE`, `CC_XACTS`, `ITEM` and `AUTHOR`.

An EB emulates the communication between the customers and the system. The interaction is done through sessions, which are a set of consecutive requests to execute some function. The session duration is controlled by a User Session Minimum Duration (USMD) time, defined as the minimum duration for which a session must last. Between two requests, the EB waits for a period of time, called Think Time.

After each interaction the EB must decide which of the navigation options will be chosen. In the TPC-W specification the probabilities of these navigation options are well defined for each of the pages. TPC-W provides three diverse patterns of behavior for the EBs, called web mixes, varying the ratio of read-only transactions vs. update transactions. The Browsing Mix presents the 95% of read-only transaction as opposed to the 5% of update transactions. The Shopping Mix specifies 80% vs. 20% and the Ordering Mix 50% vs. 50% respectively.

The workload can be adjusted by modifying the values of the mean think time and the number of EBs which take part in the simulation. The size of the database tables is calculated in function of the number of items that the system offers and the number of EBs that participates in the interaction, in order to maintain the scalability of the system.

The TPC-W primary metrics are the Web Interactions Per Second (WIPS) and system cost per WIPS, $/WIPS, calculated using the Shopping Mix. There are also defined another two secondary metrics, corresponding with the Browsing Mix, WIPSb, and with the Ordering Mix, WIPSo. TPC-W establishes a response time requirement for each type of web interaction. At least 90% of each type of web interaction must be returned in the time specified.

During the session, the item selection (that guides the TPC-W requests) can be done from three different webs or by clicking at promotional items:

- *The Best Seller Web Interaction*: it shows the 50 most popular items for a concrete subject among the 3333 most recent orders sorted by descending number of ordered items. The item is selected using a uniform random distribution.
- *The New Products Web Interaction*: it shows the list of the 50 newest products for a concrete subject sorted by descending release date. The item is selected from the list as before.
- *The Search Result Web Interaction*: it shows the list of items which match the criteria given on the previous page, Search Request Web Interaction. There the user selects a search type and defines a search string. TPC-W chooses the search type from a uniform distribution over the values author, title and subject. The search string is filled in function of the selected search type so that a specific match rate is guaranteed. This is achieved by using similar generation functions in the search string and in the field when the database is populated.
- *The Promotional Items*: the promotional items are 5 items whose images are shown on the top of some pages.

5 Conclusion

Middleware database replication architectures are becoming increasingly attractive for enterprise applications, due to their independence of the underlying DBMS. To a large

extent, that independence facilitates the modularity, maintenance, migration and portability of applications. Multiple copies of application data are stored at distributed sites and are accessed transparently by way of transactions. Hence, data locality, availability, fault-tolerance and performance are increased, at the cost of maintaining replicated data consistency. However, that may introduce an undue overhead. A focal point of this paper has been the minimization of that overhead.

Existing applications face the problem of having to adapt themselves to these architectures. Some applications even have to be completely rewritten [1], while others may remain the same [2,3,10,11], thanks to a JDBC interface. However, an issue of which application designers and users are not sufficiently taking into account is the required degree of consistency. We have reviewed different levels of data consistency that depend to a significant extent on the isolation level offered by the underlying DBMS: serializable or SI.

Another important aspect considered in this paper is the measurement of the overhead as introduced by replication middleware architectures. A key indicator for the overhead is measured by the transaction response time. That is parametrized by the kind of application and the desired degree of data consistency. That, in turn. is influenced, if not determined, by the chosen replication protocol. We have reviewed two different approaches for measuring the overhead introduced by a middleware architecture. The first one is based on measuring the response time compared to a centralized solution and its scalability. The second one consist in defining an e-commerce bookstore application and run the well-known TPC-W benchmark. The first alternative may fit a given application very well but it does not give any indication about how attractive it may or may not fit to other applications. The second approach is not very attractive either. Although it offers a rich environment for emulating many web service applications, it does not reflect the entire range of web service or application server requirements [19]. Moreover, the extent to which an application can achieve the results enabled by a middleware is highly dependent on how closely TPC-W approximates the customer application. The relative performance of systems derived from this benchmark does not necessarily hold for other workloads or environments. This is to say that extrapolations to any other environment are not recommended.

References

1. Irún, L., Muñoz, F., Decker, H., Bernabéu-Aubán, J.M.: COPLA: A platform for eager and lazy replication in networked databases. In: ICEIS'03. Volume 1. (2003) 273–278
2. Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: MIDDLE-R: Consistent database replication at the middleware level. ACM Trans. Comput. Syst. **23** (2005) 375–423
3. Armendáriz, J.E., Decker, H., Muñoz, F.D., Irún, L., de Juan, R.: A middleware architecture for supporting adaptable replication of enterprise application data. In: TEAA. Volume 3888 of LNCS., Springer (2005) 29–43
4. Gao, L., Dahlin, M., Nayate, A., Zheng, J., Iyengar, A.: Improving availability and performance with application-specific data replication. IEEE Trans. Knowl. Data Eng. **17** (2005) 106–120
5. Carey, M.J., Livny, M.: Conflict detection tradeoffs for replicated data. ACM Trans. Database Syst. **16** (1991) 703–746

6. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In Abbadi, A.E., Brodie, M.L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., Whang, K.Y., eds.: VLDB, Morgan Kaufmann (2000) 134–143
7. Wu, S., Kemme, B.: Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In: ICDE, IEEE-CS (2005) 422–433
8. Holliday, J., Steinke, R.C., Agrawal, D., Abbadi, A.E.: Epidemic algorithms for replicated databases. IEEE Trans. Knowl. Data Eng. **15** (2003) 1218–1238
9. Lin, Y., Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R.: Middleware based data replication providing snapshot isolation. In: SIGMOD Conference. (2005)
10. Cecchet, E., Marguerite, J., Zwaenepoel, W.: C-JDBC: Flexible database clustering middleware. In: USENIX Annual Technical Conference, FREENIX Track, USENIX (2004) 9–18
11. Esparza-Peidro, J., Muñoz-Escoí, F., Irún-Briz, L., Bernabéu-Aubán, J.: Rjdbc: a simple database replication engine. In: Proc. of the 6th Int'l Conf. Enterprise Information Systems (ICEIS'04). (2004)
12. Armendáriz, J., González de Mendívil, J., Muñoz-Escoí, F.: A lock-based algorithm for concurrency control and recovery in a middleware replication software architecture. In: HICSS, IEEE-CS (2005) 291a
13. Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: SIGMOD Conference, ACM Press (1995) 1–10
14. Elnikety, S., Pedone, F., Zwaenopoel, W.: Database replication using generalized snapshot isolation. In: SRDS, IEEE-CS (2005)
15. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley (1987)
16. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. **30** (2005) 492–528
17. Gray, J., Helland, P., O'Neil, P.E., Shasha, D.: The dangers of replication and a solution. In: SIGMOD Conference, ACM Press (1996) 173–182
18. Wiesmann, M., Schiper, A., Pedone, F., Kemme, B., Alonso, G.: Database replication techniques: A three parameter classification. In: SRDS. (2000) 206–217
19. TPC-W: Transaction processing performance council. Accessible in URL: `http://www.tpc.org` (2006)
20. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. **33** (2001) 427–469
21. Armendáriz, J., Juárez, J., Garitagoitia, J., de Mendívil, J.R.G., Muñoz-Escoí, F.: Implementing database replication protocols based on O2PL in a middleware architecture. In: IASTED DBA. (2006) 176–181
22. Cristian, F.: Understanding fault-tolerant distributed systems. Commun. ACM **34** (1991) 56–78
23. Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G.: Non-intrusive, parallel recovery of replicated data. In: SRDS, IEEE-CS (2002) 150–159
24. Armendáriz, J.E., Garitagoitia, J.R., González de Mendívil, J.R., Muñoz-Escoí, F.D.: Design of a MidO2PL database replication protocol in the MADIS middleware architecture. In: Proc. of 20th Intnl. Conf. on Advanced Information Networking and Applications - Vol. 2 (AINA'06), IEEE-CS (2006) 861–865
25. Kemme, B., Alonso, G.: A new approach to developing and implementing eager database replication protocols. ACM Trans. Database Syst. **25** (2000) 333–379
26. Kemme, B., Bartoli, A., Babaoglu, Ö.: Online reconfiguration in replicated databases based on group communication. In: DSN, IEEE-CS (2001) 117–130
27. Burke, R., Harlam, B., Kahn, B., Lodish, L.: Comparing dynamic consumer choice in real and computer-simulated environments. Journal of Consumer Research **19** (1992) 71–82

28. Alba, J., Lynch, J., Weitz, B., Janiszewski, C., Lutz, R., Sawyer, A., Wood, S.: Interactive home shopping: Consumer, retailer and manufacturer incentives to participate in electronic marketplaces. Journal of Marketing **61** (1997) 38–53
29. Otto, J., Chung, Q.: A framework for cyber-enhanced retailing: Integrating e-commerce retailing with brick-and-mortar retailing. Electronic Markets (**10**)
30. Degeratu, A., Rangasway, A., Wu, J.: Consumer choice behavior in online and traditional supermarkets. the effects of brand name, price, and other search attributes. International Journal of Research in Marketing (**17**)