
A New Architecture for
Enterprise Applications with
Strong Transactional Semantics

Sérgio Miguel Fernandes
INESC-ID / IST
sergio.fernandes@inesc-id.pt

João Cachopo
INESC-ID / IST
joao.cachopo@inesc-id.pt

May 2011
1This technical report is a preliminary version of the paper to appear in SPLASH Wavefront’2011.
Abstract

Despite ongoing evolutions in software architecture, one thing has remained almost constant over the years: 3-tier applications use a relational database both for data persistence and transactional support. We argue that such design, once justified by hardware limitations, endured mostly for legacy reasons and is no longer adequate for a significant portion of modern applications.

Current developments in hardware, coupled with new efficient transactional memories, enabled us to design a new architecture for the development of complex domain-rich applications. In this paper we present this new design, which is based on shifting transactional control from the database to the application server tier. With this change we are able to provide strict-serializability semantics for the programmer, and we do so with increased performance when compared with database-centric transactional support.

We have been using this new architecture since 2006 in the development of real-world complex object-oriented applications, with very good results, both in application performance and in development productivity due to inherent code simplifications enabled by strict-serializability semantics.
1 Introduction

Over the last 30 years, the development of enterprise applications evolved in many aspects, influenced by such diverse factors as the changes in hardware, or the changes in the users’ expectations about how applications should behave. All of these factors have a reflection in the applications’ architectures. Still, one thing has remained the same for many applications: The underlying database provides, not only persistence, but also the transactional semantics on which application developers rely for programming business transactions.

The generally accepted idea that strict-serializability is incompatible with performance burdens programmers with additional non-trivial development effort when they have to program concurrent transactions. Recent advances in hardware, leading to the growth in software development for parallel computation, have exacerbated this problem.

In this paper we present an architecture that we have been using for programming enterprise object-oriented applications. This architecture not only provides strictly-serializable transaction semantics, it also improves performance over traditional implementations, for the typical workloads of these applications. In Section 2 we provide a brief historical overview of the development of client/server applications and relate it to the status quo of complex enterprise object-oriented application development. We believe that historical reasons have led to some limitations in the programming model, namely with regard to transactional semantics. In Section 3 we discuss those limitations and how they affect programmers and software development. Then, in Section 4 we describe our architecture, which is based on the shift of transactional control from the database to the application server tier. Finally, in Section 5 we describe a real-world case of a large application developed with this architecture, showing evidence that it can be applied to typical software development with good performance results.

The architecture we present is not a panacea: It excels when applied to typical enterprise object-oriented applications that have complex object structures and exhibit many concurrent accesses that are mostly read-only. Fortunately, according to the evidence that we have collected over the past six years, during which we employed this architecture to several applications and collected statistics about their workload patterns, this configures the majority of the usage scenarios for today’s mainstream web-applications.

2 The shift from a 2-tier to a 3-tier architecture

A core component of most modern enterprise applications is a Database Management System (DBMS), and its use in such applications goes a long way back. The first networked client/server applications were 2-tiered. The clients made their requests directly to the database server, which in turn executed the requested operations and gave back the results. This architecture was adequate in a scenario where all the clients were on a trusted network and most of the computational power resided on a big server. In this architecture, transaction control was placed directly to the database server, which in turn executed the requested operations and gave back the results. This architecture was adequate in a scenario where all the clients were on a trusted network and most of the computational power resided on a big server. In this architecture, transaction control was placed in the (only) obvious location: The database server. Each client request would perform within a database transaction, and the server ensured the ACID properties. Database servers were expensive and they had to cope with a growing usage demand. Eventually, different semantics for the ACID properties appeared, which relaxed some of the properties (e.g. isolation), mostly for performance reasons. As client machines became more and more powerful, more computation could be performed on the client side, which would also take part in ensuring data consis-
tency. Business logic consistency started to get more complex than simple low-level consistency checking (e.g. referential integrity), including complex high-level consistency (e.g. a list can only contain odd numbers).

With the growth of the internet and the World Wide Web, a new architecture emerged. As organizations felt the need to interconnect their systems, the 2-tiered architecture no longer served their purposes. There were several reasons for this, including systems security and network bandwidth. The clients (now including systems in diverse geographical locations and outside the control of the intranet) were no longer trusted. The number of clients grew. The available bandwidth was far less than it was previously available on the intranet, such that sending large amounts of data, as the result of a database query, was no longer viable. This led to a 3-tiered architecture on which the server side was decomposed in two tiers: One for the application server and another for the database server. The database server was still responsible for the data persistence and ensuring transactional access. The application server was responsible for executing the complex business logic and interfacing with the clients, which would provide the user interface. This new architecture presented several advantages: The data was kept safe within the intranet and it was only accessed directly by a trusted system; Large queries could be obtained and processed on the server-side before sending the results over the internet to the clients. Notably, relaxed transactional semantics was kept in place, because database performance was still an important aspect in this architecture.

As the internet grew, object-oriented programming languages became commonly used in the development of large server-side applications. They were useful for their characteristics, such as component modularization, ease of reuse, data encapsulation, and abstraction. This created a gap between the persistent representation of data and its in-memory representation, known as the object-relational impedance mismatch [14]. To handle this mismatch Object-Relational Mapping (ORM) tools were developed to take care of the data mapping between the object-oriented model and the relational model. Whereas, in part, these tools simplified the programmers’ coding efforts, they also created some difficulties of their own, such as the maintenance of O/R-mapping metadata, and the varied semantics implied by object caching, lazy loading, and support for ad hoc queries. Additionally, different ORMs provided different semantics. Yet, the features provided by ORM tools kept depending on transactional support, which was still under the responsibility of the underlying DBMS, which in turn, still offered different flavors for ACID semantics, but none included support for strict serializability [15]. In fact, the isolation guarantees provided by databases have, for long, been a matter of confusion and discussion [5].

The natural evolution in software and hardware has led us to the current state, in which many developers depend on a 3-tiered architecture, develop application servers using an object-oriented paradigm, and rely on a relational DBMS for persistence. There are of course some exceptions to this, most notably in emerging large distributed systems, which may use different programming models [11] or different storage mechanisms [6] with different consistency models [18, 16]. These very large-scale distributed systems fit in a class of their own and, for the time being, are not the target of our study.

We concentrate on the development of complex object-oriented applications that require transactional support and data persistence. By complex, we mean applications that have a rich domain structure with many relationships among the classes, as well as business logic with non-trivial rules (but not typically a massive amount of data to process). There
is also an important additional observation about these applications: That most of the operations performed on the system are read-only, i.e. these systems are highly concurrent by nature. Most feature-rich web applications are examples of such applications.

3 Why we need a new architecture

We identify two problems with current implementations of the typical 3-tier architecture: One is the difficulty in ensuring consistency; the other is the reduced performance of the application server in the processing of complex operations. We address each problem in the following two subsections.

3.1 Consistency

If given the possibility, it seems clear that every programmer would prefer to have no less than strict-serializability semantics when programming concurrent transactional operations that manipulate the state of the domain objects in their code. Having such guarantee shields programmers from concurrency hazards, and allows them to write cleaner and simpler code.

To illustrate this point consider the following example: Imagine a real-time game where players can concurrently move their pieces in a map from one point to another with the restriction that after each movement is performed no player can be in the immediate vicinity of another player. Now consider the starting scenario depicted in Figure 1, which shows part of the map containing two players, P1 and P2.

Suppose that, concurrently, P1 will attempt to move one position to the right and P2 will attempt to move one position to the left. Only one move can succeed, because otherwise the two players would be left next to each other. A typical implementation of the moveTo operation could be similar to the pseudocode presented in Figure 2. The programmer checks that both the target position and its surroundings are available and, if that is the case, updates the position of the player on the map.

```java
class Player {
    void moveTo(int x, int y) {
        if (map.at(x,y).available(this) &&
            map.at(x+1,y).available(this) &&
            map.at(x-1,y).available(this) &&
            map.at(x,y+1).available(this) &&
            map.at(x,y-1).available(this)) {
            map.at(this).clear();
            map.at(x,y).set(this);
        } else {
            throw Exception("Move not allowed");
        }
    }
}

class Point {
    boolean available(Player p) {
        return this.isEmpty() || this.holdsPlayer(p);
    }
}
```

Figure 2: The moveTo operation checks the surroundings to ensure that the move is allowed and writes to the destination.

This code looks quite trivial and, when several moves are executed concurrently, each within its own transaction, the programmer might expect the ap-
plication to work just fine, i.e., after each transaction finishes, the moved player should be in a location that does not contain any adjoining players, thus maintaining the domain consistent. Sadly, this is may not be the case if transactions are executed with the isolation level provided by most of today’s mainstream databases that ensure at most snapshot isolation. Under snapshot isolation a transaction may commit even if the values that it read changed in the meanwhile, as long as concurrent writes do not intersect. In this example, if transaction T1 executes \( P1.move(2,2) \) and transaction T2 executes \( P2.move(3,2) \), then Figure 3 presents the points in the read-set and write-set of each transaction with regard to map coordinates.

<table>
<thead>
<tr>
<th>Tx</th>
<th>Read-set</th>
<th>Write-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>(2,2), (3,2), (1,2), (2,3), (2,1)</td>
<td>(1,2), (2,2)</td>
</tr>
<tr>
<td>T2</td>
<td>(3,2), (4,2), (2,2), (3,3), (3,1)</td>
<td>(4,2), (3,2)</td>
</tr>
</tbody>
</table>

Figure 3: T1 and T2 write to adjacent places concurrently. Write-sets do not intersect.

Note that write-sets do not intersect, and as such, snapshot isolation will allow both transactions to commit, leading to an inconsistent domain state. This problem is well known [12] by the name of write skew. The current solution, however, is to put in the programmer’s shoulders the responsibility of remembering to \texttt{clear()} forcefully the surrounding positions, thus causing an intersection in the write-sets [9]. This is definitely something undesirable from the programmer’s perspective, and very much error-prone in complex applications where the problem might not be easily identified, as it may occur due to the interaction of many functionalities.

3.2 Performance

So, why don’t current transactional implementations change to support strict-serializability? The fact is that there is the generalized idea that providing strict-serializability would impose unacceptable performance penalties. Such may actually be true for today’s standard architectures, which ultimately rely on the DBMS for transactional support.

When ORMs were not in use or applications had simple domain models, it was common for programmers to implement a complex database query to return exactly the results sought. Generally, this meant that very few database round trips (often just one) were enough to process each unit of work requested by the client. Most of the business logic computation was embedded on the database query and handled by the DBMS.

Today, programmers can execute complex computations that require reading into main memory many of the application’s objects. The use of ORMs and object-oriented programming facilitates and promotes object navigation instead of the creation of custom queries. This type of programming greatly increases the number of database round trips required to answer to a client’s request, with negative influence on performance. Trying to alleviate this, ORMs cache data on the application server tier, but still they depend on the underlying database to provide the transactional semantics. Unfortunately, developers of ORM tools have followed suit with databases in terms of the transactional properties provided to the application programmer.

4 A new architecture: Shifting transactional support

For a long time it was very expensive to own hardware capable of handling many concurrent requests that might operate on large amounts of data with acceptable performance. Thanks to recent hardware developments, consumer computers with many-core architectures and very large memory heaps are a re-
ality. This enables us to change the role of the application server tier.

The core idea in our architecture is to shift the responsibility for transaction control. Transactional control no longer lies on the persistence tier and is shifted to the application server tier. We use a Software Transactional Memory (STM), specifically JVSTM [7], to provide in-memory transactional support. JVSTM is a word-based STM that supports transactions using a Multi-Version Concurrency Control (MVCC) method that ensures strict-serializability. In the JVSTM, read-only transactions have very low overheads, and they never contend against any other transaction. In fact, read-only transactions are wait-free [13] in the JVSTM. JVSTM’s design goals originate from the observation and development of real-world domain-intensive web applications. These applications have rich and complex domains, both structurally and behaviorally, leading on one hand to very large transactions, but having also a very high read/write ratio. [8, 10].

For this type of applications, reducing the number of database round trips is essential for performance. This new architecture maintains the typical 3-tier layout, but uses the persistence tier only to ensure the Durability property of ACID. Atomicity, Consistency, and Isolation are ensured by the application server tier.

Conceptually all persistent data is mapped in memory, and it is managed by an Object Cache. For read-only transactions database access is only required when objects are not available in memory, which tends to be uncommon, because once loaded, objects remain in memory as long as they fit. They may be garbage collected when not in use by any transaction to make room for other needed objects. This implies that for applications whose entire persistent data set fits in memory, after warming up, the application server will never access the database for read-only transactions.

Notice that this object caching behavior is very different from the one in current implementations that rely directly on ORMs to provide them the persistent data. All ORMs that we know of ensure transaction isolation by delivering different copies of the same object to different transactions. In such implementations, new objects are constantly allocated and deallocated, as more transactions execute (this is regardless of whether the object came from the database anew or was read from some second-level in-memory cache). The object cache of our architecture ensures that, at any given time, the same persistent object is represented by at most one instance in memory. Objects can be shared among all transactions, because they are transactionally-safe. Using JVSTM this is trivially achieved, because objects are in fact immutable and have their mutable state contained in Versioned Boxes, which are JVSTM’s representation of transactional memory locations. The number of concurrently running transactions does not directly affect the required memory, in terms of persistent objects accessed by the transactions, unlike other solutions.

Nevertheless, we still use a standard ORM to persist data on a relational database. Our current implementation uses Apache’s ObjectRelationalBridge (OJB) [17] for historical reasons, but could be developed with any other ORM tool. In fact we do not need a relational database to store the objects. Any currently available storage mechanism could be used, but using the mainstream persistence infrastructure allowed us to maintain compatibility with legacy applications that still queried the database directly.

Despite the use of a standard ORM to fetch data, it is possible to load objects into memory and give

---

1We actually already have implementations running on top of other backends, namely BerkeleyDB, HBase and Infinispan.
them different layouts, depending on how versioned boxes are placed within the objects. This versatility enables adaptive designs that optimize, e.g., memory usage or object concurrency: One single versioned box per object can reduce memory usage to represent an object, whereas one versioned box per attribute may be best suited for objects with many concurrent updates to different attributes.

Given that objects normally refer other objects and so forth, loading an object from database could potentially cause loading of the entire database to materialize a single object in memory. We address this issue by implementing a lazy-instantiation mechanism. We perform bytecode rewriting to inject a special constructor that is used by the framework, whenever an object needs to be materialized in memory. Every object has a unique identifier, which also encodes information about the object’s concrete class. So, materializing any object is possible given its identifier. Whenever an uncached object is requested, the allocation routine creates the object with its version boxes having an empty value (a special NOT_LOADED flag). We call this object, a hollow object. Next, the hollow object is cached and only when any value of its attributes is actually required will it be read from the database. To illustrate the usefulness of this mechanism consider a list of objects of which we only need to know the size. The object holding the list will eventually be loaded (if not already in memory) and when accessing the contents of the list, a load of the elements will be started given their identifiers. For those objects that are already in memory, the cache will return the unique reference to the object. For those not yet loaded, the cache will return one hollow object. This is enough to count the elements of the list and no database load of the contents into the hollow objects was necessary.

Write transactions are implemented with optimistic concurrency control and write to the database only at commit time, after the STM has validated the transaction in memory. The database write is guaranteed to succeed, unless there is some catastrophic failure, because the transaction has already been validated by the STM. If the database write fails for some catastrophic reason, then the transaction is aborted in memory. We do not depend on the database transactional semantics for any transaction to succeed.

5 Real-world results with the new architecture

The architecture that we propose in this paper was developed over the last 6 years alongside the development of a real-world complex application—the FénixEDU web application [3]—and has been driving the execution of that application in a demanding production environment since 2006. Since then, this architecture was materialized in the Fénix Framework [2] and adopted by several other real-world applications, but FénixEDU is still the largest application that we know of that is using the approach that we propose here. So, in this section we present an overview and some metrics of the FénixEDU application that illustrate to what extent does the new architecture achieve its intended goals.

FénixEDU is a large web application deployed as part of an academic information system for Higher Education developed at Instituto Superior Técnico (IST), the largest school of engineering in Lisbon, Portugal. IST is home to more than 6,000 undergraduate students (BSc), 4,000 graduate students (MSc and PhD), and around 1,100 faculty members and researchers. FénixEDU supports the majority of IST’s web-based functionalities for the entire school ranging from courses and academic management to administrative support, scientific support, and admissions. The functionalities supported can be as simple as logging a summary for
a class or as complex as generating and validating timetables for the entire school. Currently there are approximately 1.2 millions lines of code, over 8,000 classes, of which 1,200 represent domain entities. It has over 3,600 different web pages for user interaction. FélixEDU collects statistics about its operation since 2006. It processes a daily average of more than 1 million transactions during week-days, peaking at 3.7 million transactions during the students’ enrollment period. Averaged along one year, over 98% of the total number of transactions processed by the system are read-only transactions, and of the remaining 2% of write transactions there are on average less than 0.2% restarts due to a conflict! Even at peak times, during enrollment periods, when there are above 300 transactions/s, the read-only/write ratio increases only to 4% and the conflicts remain under 1%. Note, however, that this throughput is not limited by the hardware, but reflects the demand made to the system by its users. In fact, all this is run on a cluster of two machines (for fault-tolerance) equipped with 2 quad-core CPUs and 32GB of RAM each that are under-used. Data loaded in memory usually takes approximately 6GB, whereas the relational database size (measured by MySQL) is under 20GB. This shows that it is possible to run a real-world application running under strict-serializability semantics and with good performance. We believe that these characteristics of the FélixEDU web application are not uncommon, and that, in fact, are representative of a large fraction of modern enterprise applications, for which our new architecture provides a very good fit.

Finally, to further evaluate the effect of this new architecture on the performance of a typical web application, we resorted to the popular TPC-W benchmark [4]. We started from a standard Java MySQL-based implementation of TPC-W that used JDBC to access directly the database, and adapted the application to use the Félix Framework [1]. We executed several runs for both implementations—the original version and the one based on Félix Framework—using different workloads, and we measured the benchmark’s throughput in terms of client Web Interactions Per Second (WIPS). For the standard benchmark workloads under test, the Félix Framework implementation showed an improvement on the throughput ranging from 2x to 9x more, depending on the workload. As an example, the best benchmark results for Félix Framework showed a sustained throughput of 4145 WIPS, whereas the best benchmark for the original version toped at 605 WIPS. Highest heap usage in the application server peaked at 26GB for the Félix Framework whereas it didn’t exceed 3GB for the original implementation.

References


