A Distributed, Multi-Staged, High-Throughput Middleware for Relational Databases

Rafael de Paula Herrera[1,2], **Alan Salvany Felinto**[1]

[1]Department of Computing – State University of Londrina (UEL)
Londrina, PR – Brazil

[2]Veltec Technological Solutions
Londrina, PR – Brazil

herrera.rp@gmail.com, alan@uel.br

Abstract. *In this paper we present a distributed middleware for relational databases. Its development was motivated by the need of improvements on a set of legacy systems from the automotive logistics and tracking industry. The solution proved to be effective once it was possible to increase the throughput of handled Requests and offer a minimum level of fault tolerance by employing a pipelined architecture along with distributed data structures. Its deployment ensured that the same legacy applications ecosystem could evolve and operate under growing commercial demand.*

Resumo. *Neste trabalho apresentamos um middleware distribuído para bancos de dados relacionais. Seu desenvolvimento foi motivado pela necessidade de se aprimorar um conjunto de sistemas legados da indústria de logística e rastreamento automotiva. A solução demonstrou ser eficaz uma vez que foi possível aumentar a vazão de requisições tratadas e oferecer um nível mínimo de tolerância a falhas ao se empregar uma arquitetura multi-estágios em conjunto com estruturas de dados distribuídas. Sua implantação garantiu que o mesmo ecossistema de aplicações legadas pudesse evoluir e operar sob crescente demanda comercial.*

1. Introduction

In technology-based automotive industry, one of main trends is that their applications being migrated to Cloud Computing model. It is usual that back-end legacy systems remains in production over years on infrastructures prone to failures at several levels. Technological and architectural restrictions imposed on scenarios like these, make them not supporting smooth transition strategies to a distributed nature. Also, organizations operating growth, caused by fleet and user base increase, makes legacy systems become gradually saturated.

The main factors that prevented the horizontal scaling of service was how they were strongly attached to the application of so-called *Relational Database Management Systems* (RDBMS), coupled with its waiting time caused by blocking execution. The development model used before and not subject to brief architectural changes, was completely thread-centric and suffered from high concurrency overhead on reading and writing information into relational databases.

On the other hand, implementing a fault tolerant mechanism is not a trivial task in applications whose core activities relies exclusively on storing internal states in memory. To ensure operations consistency of offered service, despiting application servers suffering from outages, data must be replicated over a sufficient number of nodes in the network. Being the service dependent on multiple Internet links and having its physical infrastructure subject to periodic maintenance, it becomes ineffective to employ some solution whose performance is limited to LAN context and single data center.

These reasons lead us to create a distributed middleware as an alternative to bring legacy systems to some level of survival, enough, unless they can be completely migrated to an architecture compatible with market needs.

In section 2, we are going to walk over a set of works that addressed performance and fault tolerance issues. They directly influenced the chosen design of middleware, whose higher level architectural details about how distributed data structures and algorithms interact are explained in section 3. A performance measurement is shown in 4, where classic database committing algorithms and proposed approach are confronted, and its results are properly explored.

2. Related Work

In [Bisbal et al. 1999], difficulties found during the process of maintaining legacy information systems was addressed. This can be increased even more, if distributed back-end features introduction is needed on services, in production environment and designed before to working as singular instances. We propose the idea that when using a middleware whose characteristics are comparable to the presented in this work, it is possible to ease much of migration process to a distributed environment. This is depicted by a re-reading performed over the figure 1, which relates the severity of architectural modifications introduced in information systems.

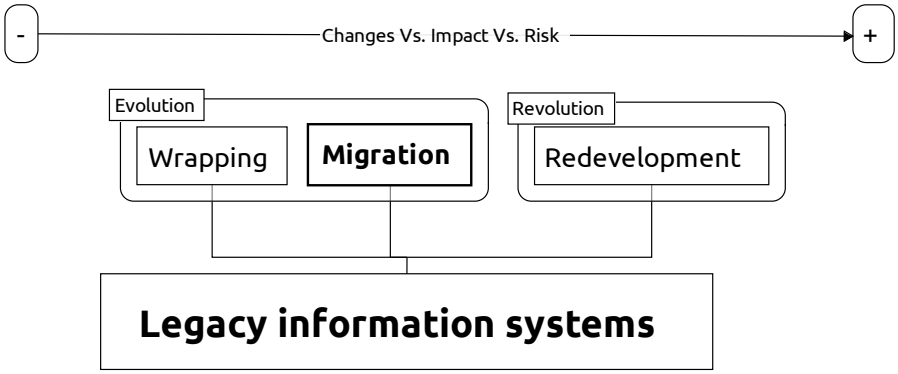


Figure 1. Severity changes on legacy information systems. The migration strategy is eased by use of middleware.

The work [Rubel et al. 2006] shows the importance of employing fail-over on applications whose operation is real-time. In many ways, its problem is similar to that found in the application we have chose to migrate to a distributed environment.

A discussion is promoted in [Barga et al. 2002] on ways to deal with failures occurring in databases, when the access is done by multi-tiered applications. In our

approach, there was linked signals to specific Responses of Requests which caused errors, when their respective SQL statements would be processed by the RDBMS. In addition, we also employed logs for audit.

Back-end services, multi-tiered interconnected by queues are discussed in [Urgaonkar et al. 2005]. Multiple processing stages connected by queues are proposed in [Welsh et al. 2001]. Both works have directly influenced the design of middleware internal processing stages. The way information is retrieved and inserted in the distributed data structures were adjusted for events that would guide all the flow in reactive way.

A prototype was developed, presented in [Luo et al. 2002], whose main motivation was to provide an intermediate caching layer between applications and database. It addressed the possibility of multiple layer instances, which occurs similarly with middleware presented here. [Cecchet et al. 2007] discusses data replication and performance characteristics, both having been taken into consideration during prioritization of distributed data structures which would be replicated across the WAN.

3. Architecture

The middleware reacts to presence of information in their structures of input and output, both replicated across multiple nodes in network. Uses distributed computing resources, provisioned by Hazelcast framework. Its components were developed based on conventions adopted by Google Guice framework.

Its architecture was influenced by the Inversion of Control design pattern [Martin 1996], implemented via Dependency Injection [Fowler 2004, Yang et al. 2008], providing high modularity and loose coupling. This made the input and output structures fully replaceable, since their implementations conforms to the established contract made by their interfaces BlockingQueue ConcurrenMap, respectively.

The data are assigned so that each node, n, is responsible for sharing its active amount along the grid nodes, while it retains a data partition load as backup from its predecessor node, $n - 1$, whose is responsible. Similarly, his successor node, $n + 1$, charges the active data from node n as its backup data partition. This training circular list, followed by the processes of partitioning and backup, guarantees a minimum fail-over level over information stored in memory grid.

A basic API was built, so the developer can interact with the grid in a simple way: inserting, removing and retrieving information over distributed data structures. Thus it was possible to fully decouple all the calls with direct references to RDBMS, performed by legacy systems, while the backward source code has been fully maintained, ensuring operating compliance for native implementations present in numerous locations such as those who are in agreement with contract established by ResultSet interface.

3.1. Stages

The middleware execution flow follows a pipeline composed of multiple stages. As shown in figure 2, stages are being classified as:

1. **Identification** The accumulated Requests in distributed input queue are consumed according to a particular algorithm. Each Request processed must have a corresponding SQL statement. This association is accomplished by means of an

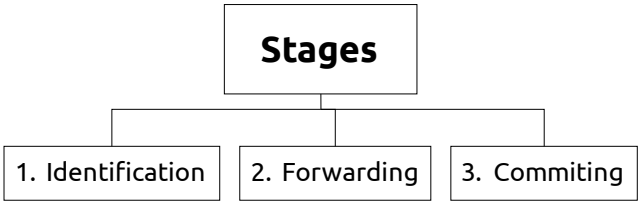


Figure 2. Ordered pipeline stages according to default operating flow.

internal correspondences map so it can cross the possible Requests kinds with the SQL statements related ones. Once obtained, the SQL statements are classified by its semantics.

2. **Forwarding** Each SQL statement is sent to an internal blocking queue and then an consumer algorithm specialist in dealing with that operations family, will then generate its pure SQL statements which will be committed on RDBMS.
3. **Committing** After being consumed, according to a specific policy, the SQL statements are fired against the RDBMS through JDBC (Java ™Database Connectivity). Requests that originates Responses are related upon on a communication key, automatically generated by Grid Client, and provided on its distributed map.

3.2. Node

Each grid node corresponds to a stand-alone instance of distributed data structures, stored in RAM memory from host servers. Upon configuration, can perform discovery of other instances, if any, over the LAN/WAN. There are two main data structures, persistent and synchronized across all grid nodes, known respectively as "Requests Queue" and "Responses Map", as shown in figure 3. They are considered well-defined points of data Input/Output to all of middleware instances.

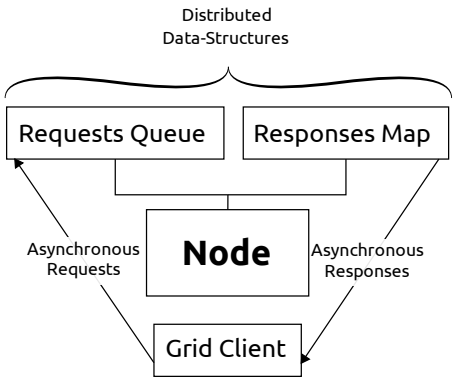


Figure 3. A grid node, having its distributed data-structures accessed by client.

Periodically, every node tries to find other active instances in the LAN, firing UDP multicast messages to the address 224.2.2.3 on port 54327. If it finds another active instance, establishes a TCP connection to it and starts the authentication process, that is validated by a group code identifier and a password, both must be known to all grid connected nodes. The communication is optionally encrypted and is based on Java™JCA (Cryptography Architecture). Being the connection successfully

authenticated, newcomers nodes are organized with others in a P2P network and begins partitioning its data among themselves. One grid can have its data-structures distributed over the WAN, as long as at least one of its nodes knows about at least one IP address located on another grid. In last case, the discovery is made solely based on TCP messages.

3.2.1. Grid Client

In order to make the information access transparent, to distributed data structures, there was developed a basic wrapping API that provides different ways to run the same set of actions on the grid. Basically, they are understood by (i) Requests Offer and (ii) Responses Recovery:

Blocking Waiting (i) Tries to insert a Request, blocking the application execution until there is an available slot in Requests Queue buffer. (ii) Attempts to retrieve the answer to a previously performed Request by blocking the application execution until the Response is available on the Responses Map.

Non-Blocking Waiting (i) Tries to insert a Request, stating the case it is not possible. (ii) Attempts to retrieve the answer to a previously performed Request, stating if it have not been made available on the Responses Map.

Timeout Driven Blocking Waiting In both cases, (i) and (ii) are extensions of their "Blocking Waiting" and "Non-Blocking Waiting" versions, respectively. (I) Tries to insert a Request, blocking the application execution until there is an available slot in the Requests Queue buffer, respecting a maximum waiting timeout and stating if it is not possible. (ii) Attempts to retrieve the answer to a previously performed Request by blocking the application execution until the Response is available on the Responses Map, respecting a maximum waiting timeout and stating if it has not been made available.

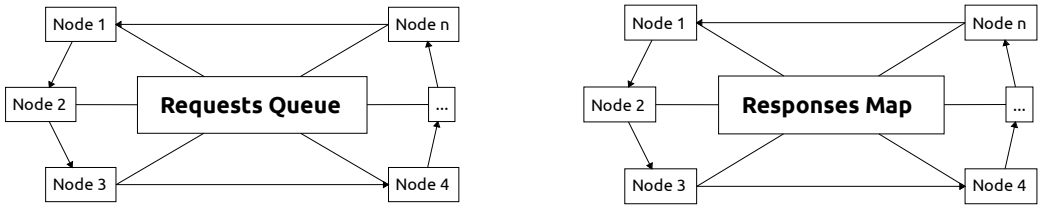
Trials Number and Timeout Driven Blocking Waiting In both cases, (i) and (ii) are extensions of their "Timeout Driven Blocking Waiting" versions. The blocking is driven by a timeout and the process is repeated according to a established number of trials.

3.3. Requests Queue Vs. Responses Map

To take advantage on middleware, applications are encouraged to make use of provided Grid Client API, instead of accessing directly the distributed data-structures. Thus, it is possible to Request and retrieve data in a transparent and simple way. It works on Requests Queue and Responses Map.

When there is need to operate over data, client grid application will input a Request in the distributed queue, exemplified by the figure 4(a). A middleware instance soon will be responsible for consuming the said Request and process it in the pipelined stages.

Once processing have been generated in RDBMS, the answer is input back on distributed map, exemplified by figure 4(b). The application just Requested that operation should be watched, asynchronously, so a Response is signaled by its presence on the Responses Map. To make this possible, one should choose a method of consumption among those implemented by Grid Client.



(a) Requests Queue (b) Responses Map

Figure 4. I/O distributed data-structures over grid nodes.

Distributed data-structures eases the backup process, employing a circular linked list on its self-organizing way, that guarantees a minimum level of fail-over on outages.

3.4. Request Vs. SQL Statement Vs. Response Vs. Communication Key

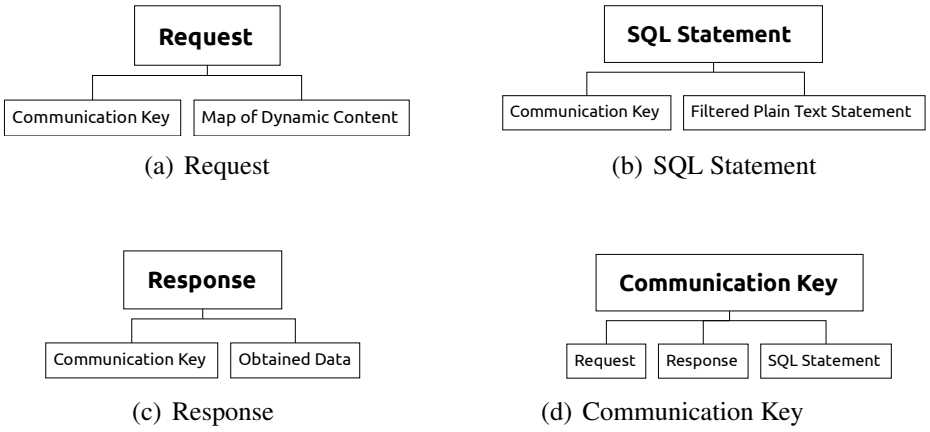
As shown in figure 5(a), every Request must contain both Communication Key and Map of Dynamic Content. The Communication Key is automatically generated by grid client, which can be replaced by a Long value, provided that it guarantees uniqueness along the grid until its related Response be consumed. The Map of Dynamic Content should be implemented based on Map interface contract, so that its key-value must corresponds to "String x String" format.

During SQL Statement mapping to related Request, an iteration is performed over all records from Map of Dynamic Content, in order to find labels inside the plain text SQL stored. If occurrence is found, the label is then replaced by its associated value, being assured that attempted SQL injection attacks will be barred.

A SQL statement should have a 1x1 relationship with a Request. It has in its content, as shown in figure 5(b) statement, a plain text representing SQL statement that should be offered to the RDBMS as an operation after being filtered. When operations are invariant its representation is static. When dynamic data should integrate statement fields, a label is searched exactly like it had been inserted into Map of Dynamic Content at original Request. If it has been found, one would have its textual token representation replaced by its corresponding value.

A Response must contain Communication Key and Retrieved Data. The first must be equals to that found in the Request that originated the Response. The second must be some data-structure that implements the interface specified by ResultSet and also, capable of assuming a disconnected form. This factor is crucial on performing data binary serialization before being persisted throughout distributed data-structures. We used a CachedRowSet interface implementation that includes all of requirements.

Depending on the nature of Requests offering mechanism, it would be necessary the assignment of a unique and pseudo-random Communication Key, able to identify each Request made. Thus, any information which was submitted by Grid Client can be referenced at any stage of processing in middleware internal flow, such as the relationship shown in figure 5(d). In general, this process simply does the linkage as a cross reference between a Request and its corresponding Response among distributed data-structures. Over normal circumstances, the Communication Key is automatically generated by Grid Client and then is attached to the Request.



(a) Request (b) SQL Statement

(c) Response (d) Communication Key

Figure 5. Entities and the relationship between them.

3.5. Mapping

At the time they are consumed, Requests go through a filtering. Its kind is identified according to an internal mapping system, responsible for relating the Requests with their SQL Statements, as shown in figure 6. Thus, it is expected that all operations subject to turn into Requests and subsequent execution in RDBMS are named and linked explicitly. This feature improves security, since Requests go through network without any form of SQL dialect. Thus, the scheme employed in information architecture internal to the RDBMS, is not revealed.

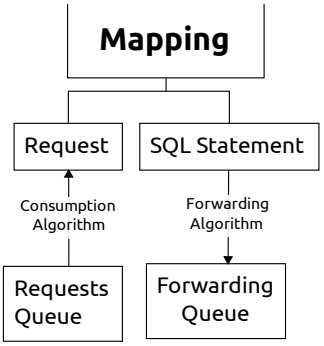


Figure 6. Mapping Requests to their respective SQL Statements.

The set of supported operations by the middleware was selected taking based on those found in the application which had been integrated. Possible operations on the standard RDBMS are understood by Select, Insert, Update, and Removing Stored Procedures.

3.6. Proxy

Having a Request received from client application and crossed with their respective SQL Statement, the routing takes place according to its kind. For each one there is an internal queue that holds their peers, as shown in figure 7. These queues have their data consumed by specialized algorithms, because each kind of SQL Statement generates a specific nature of operations on data stored in RDBMS.

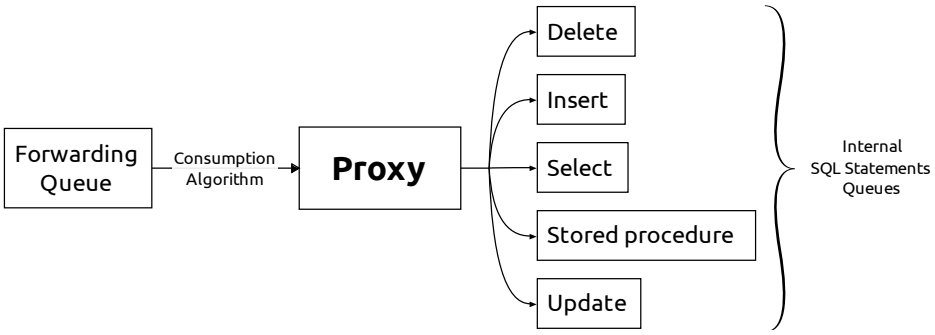


Figure 7. The Proxy forwards SQL Statements to its properly queues.

With the presented distinction, it is possible to identify the frequency with which operations are determined over submitted data. Thus, one can make a fine tunning on employed consumer algorithms, improving committing operation process in RDBMS according to optimal interval timings. Additionally, this information can be used to employ specific improvements in implementation terms.

3.7. Commiter

Once they have been consumed from their internal queues, SQL Statements are submitted to RDBMS with aid of specialized algorithms in the category of operations to be performed. Figure 8 effected shows that a pool manages database connection resources. Once SQL Statements are successfully executed, their answers will be input into the Responses Map. The same Communication Keys from original Requests are attached to the Responses.

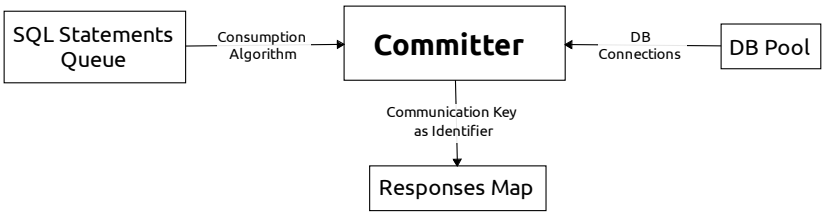


Figure 8. Each SQL Statements queue owns a specialized consumption algorithm that commits operations on RDBMS and returns a Response.

The operations consumption and committing that modifies the durable state of data can be accomplished by use of batch algorithms, since their answers can not be important, resulting in performance gains on RDBMS. It is found under the names (a) Bulk and (b) Batch. In (a) an amount of SQL Statements are concatenated and submitted together to RDBMS as a single SQL Statement. (b) is performed scheduling SQL Statements as they arrive, being validated and submitted to the RDBMS in a second time. Both can be used on the removal, updating and inserting SQL Statements.

Ones that require review about their Responses, are classified under the name of (c) Single, does not go through buffers. They are being treated and subjected directly, in specialized way, one by one. Can be used as any operation kinds and is mandatory to perform read-only queries and stored-procedures invocations.

3.8. Database Pool

In order the middleware can properly use the offered resources by RDBMS, there was employed the Apache DBCP (Database Connection Pool) component. It is a specialized database pooling solution and easily configurable for the following parameters: limiting minimum and maximum amount of active connections used by client applications, validation of connections through standard SQL query, re-establishment of lost connections, unicode support and integration with a wide JDBC drivers variety.

Figure 9 exemplifies pool access to several previously established database connections. As required, the pool checks for resources availability. If so, a fully managed connection is returned. Else, is told to wait until resource is available.

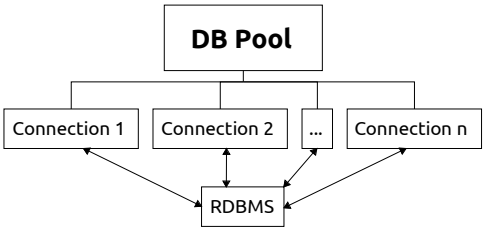


Figure 9. The Pool maintains control over RDBMS connections and provides them, as automatically managed resources.

4. Results

A core business back-end application was chosen as proof of concept on migrating to a distributed environment. The use of middleware played a key role easing the entire process. During the elaboration of data committing mechanism, we conducted a survey over a variety of SQL Statements which was found on direct RDBMS calls. The table 1 lists SQL Statement kinds with its occurrence.

Table 1. SQL Statement kinds found in proof system

SQL Statement kind	Total	≈ Total (%)
Delete	2	1.92
Insert	5	4.80
Select	14	13.46
Stored Procedure	8	7.69
Update	75	72.11

The number of SQL Statements related to updating data, about 72.11%, has motivated the blocking time measurement, which was suffered as result of this kind of application calls.

This comparison relates the number of processed operations with timing that application remains blocked until all being completed. The experiment generated 30 samples after having 1 up to 100,000 Requests processed, consumed by traditional algorithms for direct access to the RDBMS versus the suggested competitor, employed by the use of the Grid Client API, where middleware operates. All iterations, the

set of chosen operations was defined randomly, in order to not favor any caching mechanism. Relevant characteristics of computer hardware that served the experiment are: Intel®Core™i3-350M (3M Cache, 2.26 GHz), 4GB DDR3 RAM, Samsung HM321HI HD and RTL8101E/RTL8102E PCI Express Fast Ethernet controller.

Figure 10 shows significant throughput increase, due to asynchronous model used in detriment to direct access on RDBMS. Up to ≈5 thousand of operations, we observed that there is still a level of competitiveness between different consumption approaches. On ≈5 thousand up to ≈20 thousand of operations, becomes clear that direct access to RDBMS methods are blocking the application at levels much higher than that offered by the presented middleware.

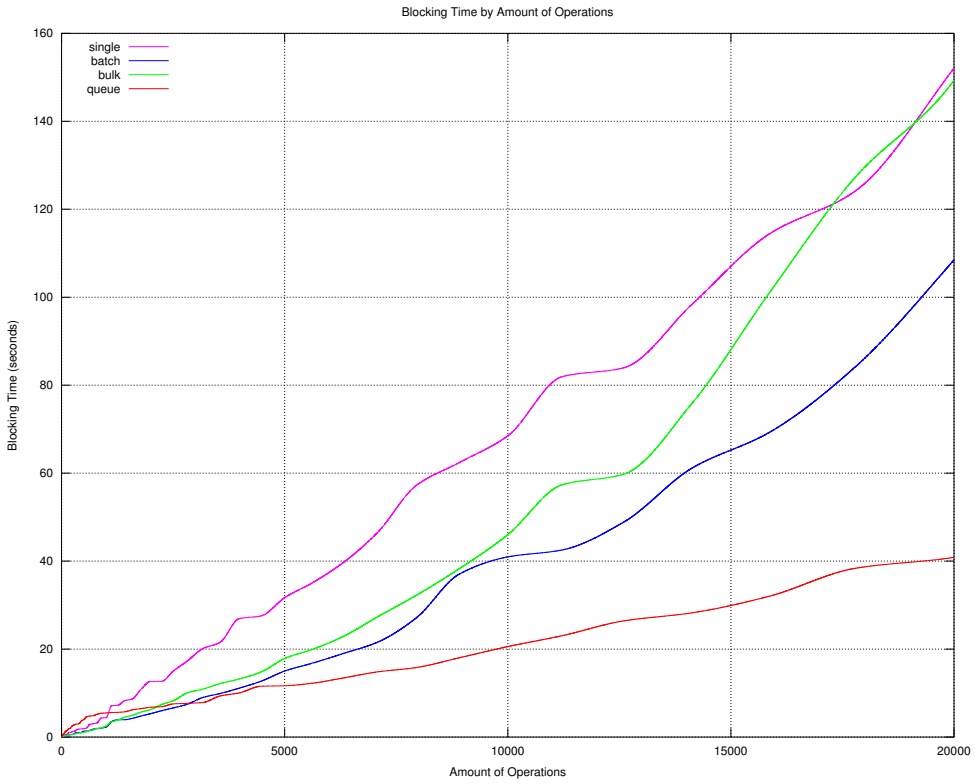


Figure 10. Application blocking time (s) Vs. amount of operations.

Figure 11 shows that when passing ≈20 thousand operations mark, we can see clearly the importance of having been used a queue based data structure as an entry point for Requests, fact that gave support to a much higher magnitude of overload with respect to that provided by traditional methods of blocking waits for RBDMS operations.

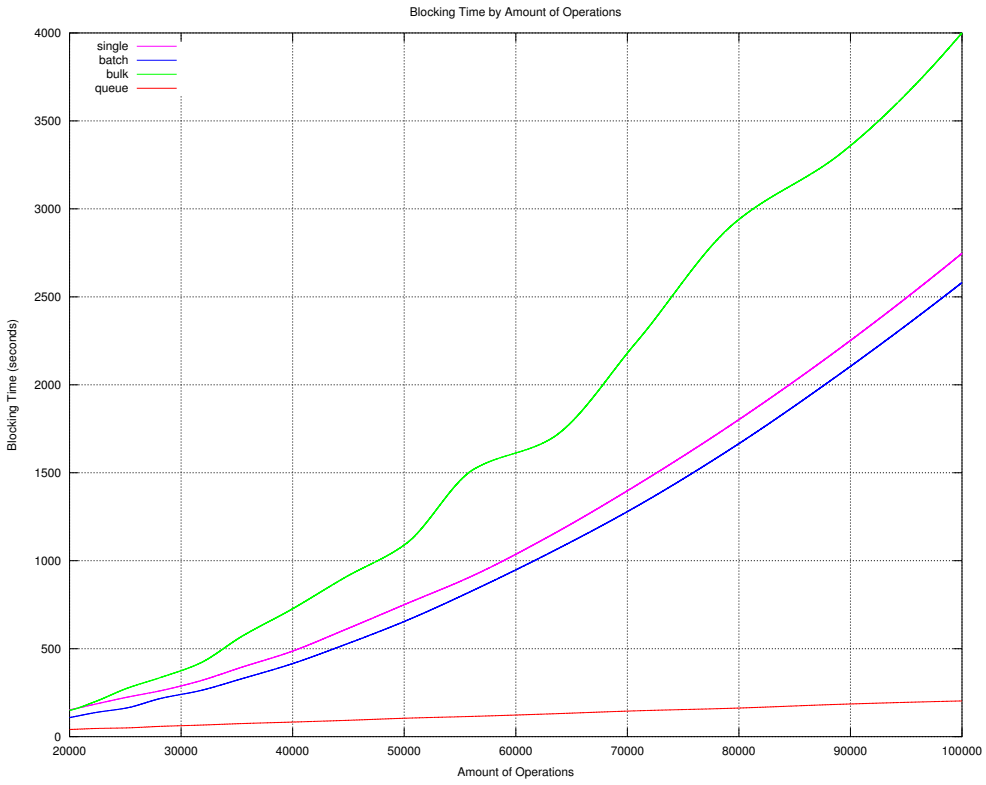


Figure 11. Application blocking time (s) Vs. amount of operations.

5. Conclusion

The middleware was successfully able to assist the process of migrating a back-end systems to a distributed environment. This was achieved because the responsibility to ensure recovery and durable writing data in RDBMS was completely detached from the application in evidence. This process, before comprised by a monolithic flux, was replaced by an event-driven approach, reactive to present data in distributed data-structures.

Information replication has eased the single points of failure elimination across entire chained applications. Being data essential to consistently working of services persisted over several nodes in the grid, it became feasible the process of back-end services instances multiplying on LAN/WAN, the main effect was the risk reduction due to outages. Having been satisfied with information flow processed, its implementation ensures a higher survival course of commercial operations for legacy applications ecosystem.

The developed migration model, able to make back-end applications operating on distributed environments, is being explored as primary means of a smooth applications transition to an infrastructure entirely based on cloud computing and the first stage was solely dependent on the successfully integration and deployment of proposed middleware.

References

Barga, R., Lomet, D., and Weikum, G. (2002). Recovery guarantees for general multi-tier applications. *Data Engineering, International Conference on*, 0:0543.

Bisbal, J., Lawless, D., Wu, B., and Grimson, J. (1999). Legacy information systems: Issues and directions. *IEEE Softw.*, 16:103–111.

Cecchet, E., Candea, G., and Ailamaki, A. (2007). Middleware-based database replication: The gaps between theory and practice. *CoRR*, abs/0712.2773.

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html.

Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B. G., and Naughton, J. F. (2002). Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 600–611, New York, NY, USA. ACM.

Martin, R. C. (1996). The dependency inversion principle. *C++ Report*, 8:61–66.

Rubel, P., Loyall, J. P., Schantz, R. E., and Gillen, M. (2006). Fault tolerance in a multi-layered dre system: A case study. *JCP*, 1(6):43–52.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2005). An analytical model for multi-tier internet services and its applications. *SIGMETRICS Perform. Eval. Rev.*, 33:291–302.

Welsh, M., Culler, D., and Brewer, E. (2001). Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35:230–243.

Yang, H. Y., Tempero, E., and Melton, H. (2008). An empirical study into use of dependency injection in java. In *Proceedings of the 19th Australian Conference on Software Engineering*, pages 239–247, Washington, DC, USA. IEEE Computer Society.