Providing Transaction Class-Based QoS in in-Memory Data Grids Via Machine Learning

Pierangelo Di Sanzo, Francesco Maria Molfese, Diego Rughetti, Bruno Ciciani
DIAG, Sapienza University of Rome

***Abstract*—Elastic architectures and the so-called "pay-as-you-go" resource pricing models offered by many cloud infrastructure providers may seem the right choice for companies dealing with data centric applications characterized by high variable workload. In such a context, in-memory transactional data grids have demonstrated to be particularly suited for exploiting advantages provided by elastic computing platforms, mainly thanks to their ability to be dynamically (re-)sized and tuned. Anyway, when specific QoS requirements have to be met, this kind of architectures are complex to be managed without the stand of mechanisms supporting run-time automatic sizing/tuning of the data platform and the underlying (virtual) hardware resources provided. In this paper, we present a neural network-based self-regulating architecture where an in-memory data grid is automatically tuned in order to provide transaction class-based QoS while minimizing the cost of the computing infrastructure. We also present some results showing the effectiveness of our architecture, which has been evaluated on top of Future Grid IaaS Cloud using Red Hat Infinispan in-memory data grid and the TPC-C benchmark.**

I. Introduction

Today the availability of cloud infrastructures is constantly increasing thanks to the presence in the market of even more cloud providers, which also offer diversified and higly customizable architectures. Additionally, even more companies are investing in cloud technologies for its own private data centers.

A potentially disruptive feature of cloud architectures is the elasticity, i.e. the amount and the type of used (virtual) computing resources can be very rapidly changed. E.g. current technologies allow to start-up a number of new virtual servers, and to add them to an existing server pool, in less then a minute. Nowadays, most of cloud providers adopt the so-called "pay-as-you-go" pricing model, where users pay on the basis of the actual usage of resources (e.g. on the basis of the usage time, the amount of stored data, etc). As a consequence, the possibility to (re-)configure on-demand this kind of architectures allows to extremely reduce computing infrastructure costs incurred by companies. In-fact, the amount of used resources could be constantly regulated over time, on the basis of the current workload, in a way to exploit only those actually required in order to, e.g., meet pre-established Service Level Agreement (SLAs).

When dealing with data management systems, particularly with those characterized by high dynamic workload, the last generation of in-memory transactional data grids (such as Red Hat Infinispan [1], VMware vFabric GemFire [2], Oracle Coherence [3] and Apache Cassandra [4]) have revealed to be good candidates for taking advantage of elastic architectures. In fact, these products allow fast resizing of the data platform in terms of number of cache servers, being able to re-distribute data object among them (also on the basis of differentiated data placement policies) without requiring any specific human action. Additionally, they offer a set of configuration parameters (such as the data object replication degree, i.e the number of replicas to be stored for any data object) allowing to tune the data platform in order to achieve the expected (tradeoff between) performance and fault tolerance level.

Anyway, establishing the amount of resources to be used, as well as the optimal data platform configuration, in order to achieve a given performance level is hard to be performed by humans. The complexity of this task is particularly exacerbate by the fact that performance indicators (e.g. system throughput, transaction response time) typically show non-linear dependencies with respect to data platform configuration parameters (e.g. number of used cache servers, data object replication degree),

In this paper we present a neural network-based architecture which automatizes the resizing/configuration of an in-memory transactional data grid deployed on top of a elastic cloud architecture. Particularly, the resizing/configuration process is driven by the need for both: (1) meeting explicit transaction class-based SLAs and (2) exploiting the minimal amount of (virtual) hardware resources in order to minimize the total infrastructure cost. The architecture that we present has been designed leveraging some results presented in [5], where neural networks have been exploited to predict the expected system throughput and the average transaction response time as a function of three system configuration parameters, i.e. the number of clients, the number of cache servers and the data object replication degree. Essentially, the study cited above acts as a proof-of-concept for the architecture we present in this paper. In fact, we consider a more complex scenario where we deal with transaction class-based QoS and a dynamic workload profile. Particularly, we assume that, in addition to variation of the overall transaction arrival rate, also the arrival rate of each transaction class may vary over the time, exactly as is the case of many real life transactional applications.

Our architecture leverages a neural network-based approach, where neural networks are used to predict transaction response times for each transaction class featuring the system workload. We note that the response time of a given transaction class may be also affected by (the number and the profile of) concurrent transactions belonging to other

transaction classes. Thus, the mix of concurrent transactions has to be accounted for in order to make reliable predictions. In our approach, we train the neural networks using samples including measurements of system workload parameters related to all transaction classes and of parameters describing the current system configuration.

In this paper we also present a real implementation of our architecture, which we built on top of Infinispan in-memory data grid. Finally, in order to demonstrate the effectiveness of our architecture, we present results of an experimental study we carried out on top of Future Grid IaaS Cloud [6] , where we used an implementation of the TPC-C benchmark.

The remainder of this paper is structured as follows. We discuss related work in Section II. A brief introduction on in-memory transaction data grids and a brief recall on neural networks are provided in Section III and in Section IV, respectively. Details on our architecture are described in Section V. Finally, we depict and discuss results of the experimental study in Section VI.

II. Related Work

While the attempt to develop self-regulation mechanisms for optimizing performance of in-memory transactional data grid is quite recent, various proposals have been instead presented in the general context of parallel and distributed systems. Several proposals leverage control theory techniques. Most of these assume linear performance models, which are dynamically updated at run-time when the system shifts from one operating point to another. These proposals include, e.g., first-order auto-regressive models aimed to allocate computing power (in terms of number of CPU) to Web servers [7]. Further, linear multi-input-multi-output models have been exploited in multi-tier systems [8] to manage different kinds of resources, as well as to manage CPU allocation for minimizing interferences on the same physical node between virtual machines [9]. With respect to solutions based on adaptive linear models, our architecture is expected to identify the optimal configuration also in presence of non-linear dependencies between system performance indicators and data platform configuration parameters (which is what typically happens in real cases).

Moving to the context of multi-tier architectures in cloud environments, machine learning has been used for dynamically resizing the number of back-end servers, as in [10], [11], [12], [13]. Anyway, none of them targets in-memory data grids, but they address traditional data base management systems, which are not suitable to deployed on top of elastic cloud architectures. Additionally, proposed solutions mostly target processing of complex queries on relational data. Our approach is not biased towards read-only accesses, thus resulting more general. Finally, in our work, we also deal with the more general and scalable case of partial replication of data objects, instead to limit our study to the case of full replication.

The work presented in [14] leverages a mixed methodology where analytical modeling and machine learning predictors are exploited to tune in-memory data grids. The analytical model used in this work requires knowledge of the specific algorithms used by the data platform for managing local transaction concurrency and distributed data synchronization. Conversely, our approach is completely black-box, i.e. no knowledge about internals of the in-memory data grid is required. Further, the work in [14] only deals with full replication of data objects.

Finally, the work presented in [15] depicts algorithms aimed to self-tune distributed transactional data management systems, specifically targeting the case of in-memory transactional data grids. Also in this case, tuning leverages an hybrid approach which combines analytical performance models and local exploration. The latter is aimed to progressively enhance the accuracy of the analytical model. Also in this case, differently from our work, it does not exploit a pure black-box approach. Additionally, the proposed solution has been evaluated through simulation. Conversely, we tested our architecture on top of a real cloud infrastructure.

III. Overview on transactional in-memory data grids

Transactional in-memory data grids act as a distributed/replicated cache memory for applications, providing a basic programming interface for storing and retrieving data objects, including PUT and GET methods. These platforms hide away to programmers all issues concerning data distribution and replication, providing a number of setting parameters allowing to change the configuration of the data platform, such as the number of used cache servers, the data object replication degree, the transaction isolation level, etc. With respect to traditional database management systems, a key feature of this kind of platforms is that data storing operations (e.g. transaction commit operations) execute data updates only in main memory, without performing any kind of operation on stable storage. After an update operation is terminated, data, if needed, are asynchronously moved to stable storage. On one hand, this entails a drastic reduction of the transaction commit time (thus of the transaction response time) with respect to the case of executing update operations on stable storage. Additionally, this extremely simplifies operations as adding/removing cache servers. In fact, all operations requiring synchronous data transfer between servers (such as the server state transfer operation) only involve data which are stored in main memory, so that these operations become extremely fast and simple to be performed. Essentially, this feature makes an in-memory data grid particularly suited to exploit elastic architectures offered by cloud environments. On the other hand, the main drawback arising from executing data update operations only in main memory is that the system reliability is reduced with respect to the case of performing data updates also on stable storage. To cope with this issue, these data platforms allow to tune the number of used cache servers and the data object replication degree, so that the desired reliability level can be achieved. Anyway, as we already hinted in Section I, modifi-

cations to the data platform configuration typically affect the system performance according to non-linear dependencies, so that deciding the optimal data platform configuration becomes a non-trivial task.

IV. Neural Networks Recall

A neural network is a machine learning method [16] providing the ability to approximate various kinds of functions, including real-valued ones. Inspired to the neural structure of the human brain, a neural network consists of a set of interconnected processing elements which cooperate to compute a specific function, so that, provided a given input, the neural network can be used to calculate the output of the function. By relying on a learning algorithm, the neural network can be trained to approximate an unknown function f exploiting a data set $\{(\mathbf{i}, \mathbf{o})\}$ (training set), which is assumed to be a statistical representation of the function f such that, for each element $(\mathbf{i}, \mathbf{o})$, $\mathbf{o} = f\{\mathbf{i}\} + \delta$, where δ is a random variable (also said *noise*).

V. The self-configuring architecture

In this section we describe the self-configuring architecture. Firstly, we focus on the target system model. After, we formally define the optimization problem addressed by our architecture and then we describe the neural network-based performance prediction scheme. Finally, we focus on the self-configuration scheme exploited by a controller (integrated within our architecture), which is in charge of dynamically modifying the data platform configuration as a response to modifications of the workload profile of the application.

A. Target System Model

We assume an in-memory transactional data grid where up to n^{max} homogeneous (in terms of computing resources) cache servers can be used. We denote with n the number of servers actually exploited at a given time. n can be changed over the time, given the constraint $1 \leq n \leq n^{max}$. A load balancer ensures uniform workload distribution on all n cache servers. A number of data objects are stored across the cache servers, and g is the data object replication degree (of any data object), where we have the constraint $g \leq n$. Replicas of the same data object are stored in different cache servers. Positioning of data objects (and of their replicas) depends on a data distribution function that we assume to provide uniform usage of memory across all the n cache server.

In order to ensure a desired reliability/availability level, we assume that user can establish the minimum number of replicas g^{min} for any data object that have to be maintained by (different) cache servers. Finally, as data objects can be replicated (thus giving rise to an increase of required memory), we assume to have, as a further constraint, the maximum amount of memory m^{max} that can be used to store data objects (and their replicas) on each cache server. The value of m^{max} can be established by user (e.g. on the basis of a percentage of the total amount of available memory on a cache server, or in a way to guarantee that a given amount of memory on any cache server is available for other tasks). We denote with m the actual amount of used memory to store (replicas of) data objects on any cache server at a given time. On each cache server there are k active threads (parallelism level) for executing transactions, where k can be changed over the time, given the constraint $1 \leq k \leq k^{max}$, assuming that k^{max} is the maximum number of threads that can be used on a cache servers.

Finally, we assume that the workload profile of the application running on top of the data platform includes c transaction classes. The overall transaction arrival rate is denoted with λ. The transaction arrival rate of any transaction class i, with $1 \leq i \leq c$, is a fraction f_i of λ, thus having $\sum_{i=1}^{c} f_i = 1$. Both λ and any f_i, with $1 \leq i \leq C$, can change over the time.

B. Optimization Problem

Because we aim to ensure transaction class-based QoS, we assume that for each transaction class i a given SLA, expressed in terms of maximum average transaction response time r_i^{max}, has to be met. We denote with r_i the average transaction response time provided by the system for the transaction class i.

We define the optimization problem addressed by our self-configuring architecture as:

$$min(n)$$

$$\begin{cases} r_i \leq r_i^{max}, & \forall i : 1 \leq i \leq c \\ 1 \leq n \leq n^{max} \\ g^{min} \leq g \leq n \\ 1 \leq k \leq k^{max} \\ m \leq m^{max} \end{cases} \quad (1)$$

which establishes in a formal way that we want to minimize the cost of the infrastructure (in terms of number of cache servers) given that a set of constraints, in particular including SLAs for all transaction classes, are not violated.

C. Performance Prediction Scheme

As we hinted before, the self-configuring architecture exploits a prediction scheme providing the expected transaction response time of each transaction class i depending on a set of system parameters. According to the system model we presented above, parameters that can affect the transaction response times are n, g, k, λ and f_1, ..., f_c. Thus, the aim of our neural network-based prediction scheme is to provide predictions of the average transaction response time of each transaction class by capturing their dependencies on the above set of parameters. To this aim, all the aforesaid parameters are used as input to the neural network, which

outputs the expected values of r_1, ..., r_c. Accordingly, the neural network is in charge of calculating the function

$$(r_1, ..., r_c) = f(n, g, k, \lambda, f_1, ..., f_C) \quad (2)$$

The neural network is trained exploiting a set of samples (training set) of real measurements of the input and output parameters. Each sample includes a value for each input parameters of the function 2 (thus identifying a given data platform configuration and a given workload profile) and the associated values of the output parameters measured on the system. As for collecting samples and training the neural network, they can be done both in advance (i.e. during an initial phase specifically prescribed for executing the application in order to accomplish these tasks), or during the actual execution of the application (in a way to incrementally gather on-line samples and incrementally train the neural network). We will provide further details on these issues in Section VI-B

D. Controlling the Data Platform Configuration

We note that some of the input parameters of our prediction scheme, i.e. n, g and k, can be adjusted by reconfiguring the data platform. Conversely, this can not be done for other parameters, i.e. λ and f_1, ..., f_C, because they depend on external factors. In our architecture, a controller is used to re-size/configure the data platform by modifying parameters n, g and k. The controller takes as input the values of r_1^{max}, .., r_c^{max} and works on a periodic basis, where T is the (tunable) length of a period. Along each period, the controller measures the average values of λ and f_1, ..., f_c. In our architecture we suppose that T is small enough such that average values of parameters defining the workload profile slowly change with respect to T. Thus, we assume that, in a majority of cases, measures of values along a period can be used as sufficiently accurate prediction of the workload profile in the next period. At the end of each period, the controller tries to solve the problem 1. Specifically, for each combination of values of n, g and k which do not violate constraints of the problem 1, it provides the measured average values of parameters defining the workload profiles as input to the neural network in order to achieve the predicted transaction response times (i.e. the outputs of Function 2). We note that the problem 1 includes a constraint on the amount of required memory m for storing data objects on any cache server, which must be at most equal to m^{max}. When the data platform configuration changes, the amount of memory which will be required on a cache server can be estimated by simply measuring the current amount of memory, say m^a, required on a cache server and on the basis of the current number of cache servers, say n^a, and the current data object replication degree, say g^a. Specifically, if in the next configuration the number of cache servers and the data object replication degree will be equal to n^b and g^b, respectively, the amount of memory m^b that will be required on any cache server can be estimated as

$$m^b = \frac{m^a \cdot n^a}{g^a} \cdot \frac{g^b}{n^b} \quad (3)$$

Thus, the controller uses the above equation to exclude all combinations of values of n and g for which the memory constraint of the problem 1 is violated.

Once collected results, the controller decides for the optimal data platform configuration, i.e. it evaluates, between all the admitted combinations of values of n, g and k, which one minimizes n. Finally, it reconfigures the data platform according to the selected configuration.

A picture representing all components of our architecture and their interactions is shown in Figure 1.

We conclude this section by discussing the tuning of T. We observe that, assuming an in-memory transactional data grid deployed on top of mainstream cloud infrastructures, resizing the data platform in terms of number of cache servers and modifying the data object replication degree typically may requires a time in the order of one or more minutes. In our architecture, the length of T should be not less than the time required by the system, after any modification to its configuration, for both reaching stability and getting sufficiently accurate statistics on transaction response times (for deciding the next optimal configuration). Thus, typically, a length of T significantly lower than 5 minutes could not be sufficient to these aims. Adding to that, the value of T mainly depends on the specific application and how quickly modifications of workload (to which the system is expected to react) occur. As an example, assuming enterprise applications such as e-commerce ones, values of T in the order of 20 minutes could be adequate to react to workload fluctuations typically associated to variations of number of users along a day.

VI. Experimental study

In this section we discuss an experimental study we carried out to evaluate the effectiveness of the self-configuring architecture we presented in this paper. We first describe the experimental setting, then we discuss the achieved results.

A. Experimental Setting

In our study we used an implementation of the TPC-C benchmark running on top of Infinispan in-memory data grid, which has been instrumented for integrating an implementation of the controller described in Section V-D. An overview of the TPC-C benchmark and of Infinispan are provided ahead in this section. As for the implementation of the neural network, we used an acyclic feed-forward full connected network [17], that we coded leveraging on FANN open-source libraries (version 2.2.0) [18]. To train the network we used the back-propagation algorithm [17]. We observed that, generally, best predictions were achieved with a number of hidden nodes between 32 and 64, performing a number of iterations of the back-propagation algorithm between 3200 and 6400.

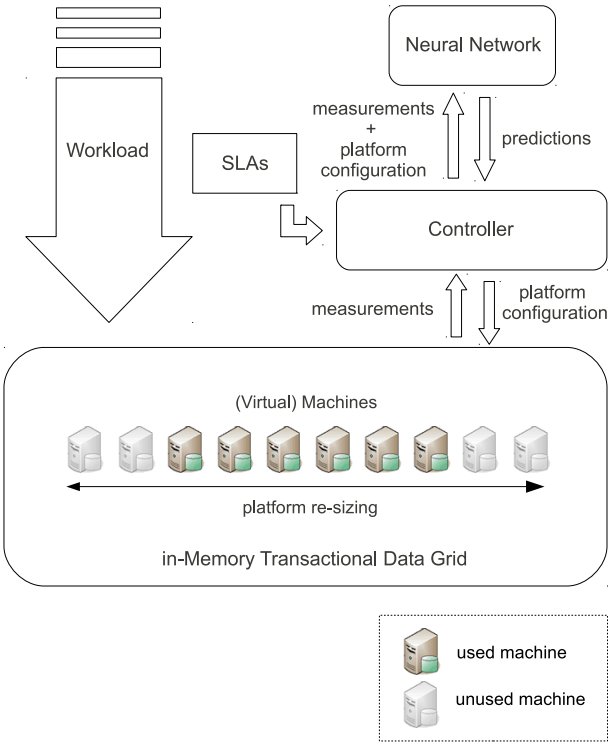

Figure 1: Reference architecture.

We performed our tests on top of Future Grid IaaS Cloud, where we used a cluster of 20 machines all equipped with an Intel Xeon X5570 @2.93 GHz quad-core processor, 2 GB of RAM and Centos 5.7 operating system.

1) TPC-C benchmark: The TPC-C is a benchmark proposed by the Transaction Processing Performance Council for comparing performance of OLTP systems. It simulates an online store where users execute orders and payments of products. TPC-C assumes a client-server architecture where clients execute transactions against a database. Transactions with five different profiles (transaction classes) are executed by clients, performing the following actions: entering new orders (New-Order transaction), recording payments (Payment transaction), checking the status of orders (Order-Status transaction), delivering orders (Delivery transaction) and checking the level of product stock (Stock-Level transaction). Because TPC-C has been originally designed for database management system-based architecture, in our experiments we used a porting of TPC-C where data are instead hosted in an in-memory transactional data grid. In order to test the ability of our architecture to re-configure the system as a response to workload variations, in our experiments we changed over the time both the overall transaction arrival rate (that we denoted with λ in our system model) and the composition of the workload in terms of mix of transaction classes (in our system model we denoted with f_i the fraction of transactions of class i). Particularly, we used f_1 to denote the fraction of New-Order transaction, f_2 for Payment transaction, f_3 for Order-Status transaction , f_4 for Delivery transaction and f_5 for Stock-Level transaction.

As for SLAs, we used the average transaction response times of each transaction class. Anyway, in our architecture also other metrics, as the percentile, can be used by simply changing the metric observed by the controller and used to train the neural network.

2) Infinispan: Infinispan is an open source in-memory transactional data grid by JBoss/Red Hat. Data objects can be distributed and replicated across an arbitrary number of cache servers. Anyway, Infinispan exposes a key-value store data model, hiding to programmers all issues concerning data distribution and replication. Commit operation of a transaction is executed only in-memory, without involving stable storage. The two-phase commit protocol is used to preserve data consistency, involving in the commit operation all cache servers which store replicas of data objects modified by the committing transaction. In order to overcome limitations due to size of memory, data objects can be asynchronously moved to stable storage after the commit operation. Infinispan allows dynamic resizing of the data platform in terms of cache servers, supporting run-time join/leave of cache servers and taking care of re-distributing (replicas of) data objects according to the new size of the data platform. Anyway, originally no possibility to vary at run-time the data object replication degree was offered, which was therefore statically defined at start-up time. This facility is instead provided by more recent beta releases, as the 5.2.0.Beta2. In order to exploit the key-value data model offered by Infinispan, the original database schema of the TPC-C has been re-designed in order to fit such a kind of data model. As a final remark, we note that our experimental study is not aimed to evaluate/compare the performance of our system when running TPC-C. Instead, it focuses on the evaluation of the our self-configuring architecture to reconfigure the system as a response to workload variations while minimizing the cost of infrastructure without violating SLAs. Thus, modifications we carried out to the system architecture of our experimental study with respect to the original TPC-C requirements do not affect our actual objectives.

B. Presentation of Tests and Results

Given the hardware features of our cluster, in our tests we set $n^{max} = 20$, and we decided to limit the amount of memory reserved for storage of data objects to 1 GB per machine, in a way to ensure that the remaining 1 GB was available to other tasks. Thus we set $m^{max} = 1GB$. As for k^{max}, we set $k^{max} = 16$. Concerning the transaction arrival rate, we observed that the maximum sustainable value by the system was, in any case, less then 1000 transactions per seconds. Thus, in our tests, we varied λ between 10 and 1000 transaction per seconds. Finally, as for transaction class mix, we used reference values established by TPC-C, introducing variations to the fraction of each transaction class as follows:

New-Order transaction and Payment transaction (i.e. f_1 and f_2) between 0.2 and 0.6, while Order-Status transaction, Delivery transaction and Stock-Level transaction (i.e. f_3, f_4 and f_5) between 0.02 and 0.2.

In all our tests we observed that by using five different neural networks, each one calculating the average transaction response time of a single transaction class i (i.e. each neural network calculates the function $r_i = f(N, G, \lambda, f_1, ..., f_C)$), the average prediction error (calculated for all transaction classes) was always reduced with respect to the case where we used a single neural network with five different outputs (each one for a single transaction classes, according to the Function 2). Thus we decided to always use a quintuple of different neural networks.

We trained the neural networks using 200 samples we gathered while randomly varying parameters n, g, k, λ, and f_1, ...,f_5 within the associated intervals as defined above. Then, we evaluated the average approximation error by comparing predictions of neural networks with respect to real values we measured for 600 different data platform configurations and workload profiles. The errors we measured for the five neural networks were 20%, 12%, 13%, 19% and 11%, respectively. In Figure 2 we show dispersion charts representing the correlation between the average transaction response times as predicted by the neural networks and as measured on the system. We recall that a lower prediction error corresponds to a higher concentration of points along the diagonal straight line evidenced in the graphs. Due to space constraints, we only show the cases of Payment transaction and Order-Status transaction (since in our tests they were transactions with highest response time).

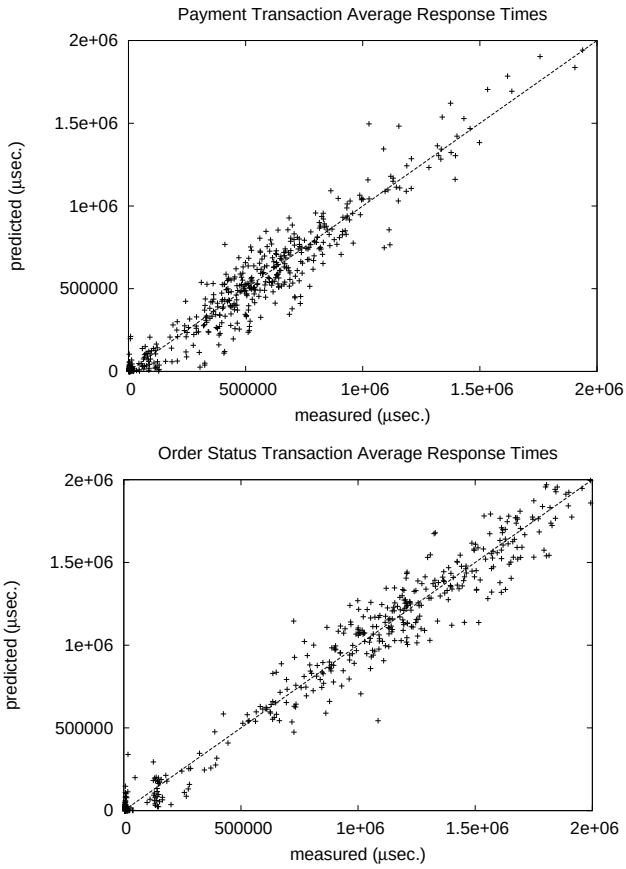

Figure 2: Neural networks prediction accuracy evaluation.

In order to show the ability of our architecture to reconfigure the data platform, we present a test where we run TPC-C by initially setting $n = 2$, $g = 1$, $k = 4$, $\lambda = 50tx/sec.$, $f_1 = 0.4$, $f_2 = 0.4$, $f_3 = 0.08$, $f_4 = 0.06$ and $f_5 = 0.06$. As for SLAs, we set $r_i^{max} = 0.5sec$ for $1 \leq i \leq 5$. Finally, we set $g^{min} = 1$ and $T = 300sec.$. With this configuration, the /average transaction class response times as measured by the controller after the first period were (expressed in seconds) $r_1 = 0.348$, $r_2 = 0.959$, $r_3 = 1.912$, $r_4 = 0.421$, $r_5 = 0.313$, while m was about $535MB$ (in our tests, we observed that the required memory for storing all data objects, excluding their replicas, was about $1.066GB$). In this case, the configuration decided by the controller was $n = 7$, $g = 6$ and $k = 4$ (we note that, in the previous configuration both constraints $r_2 \leq r_2^{max}$ and $r_3 \leq r_3^{max}$ were violated). With the new configuration, the controller measured, at the end of the subsequent period, $r_1 = 0.013$, $r_2 = 0.177$, $r_3 = 0.344$, $r_4 = 0.039$, $r_5 = 0.02$, and $m = 913MB$. We can observe that all these values satisfy constraints of the optimization problem. In order to verify the optimality of the choice executed by the controller, we checked if there were, for the same workload profile, other possible configurations with $n < 7$ and for which all constraints were satisfied. We note that, according to constraint $m^{max} = 1GB$, no configurations where $g = n$ (i.e. with full replication) were possible (because with full replication the memory requested on each cache server to store data objects and their replicas was equal to $1.066GB$, thus violating constraint $m \leq m^{max}$). Conversely, for all configurations such that $g < n$, the constraint $m \leq m^{max}$ was not violated (in the worst case, i.e. with $n = 6$ and $g = 5$, m was equal to $888MB$). Thus, we measured the average transaction response times only for admitted configurations, i.e. such that $n < 7$ and $g < n$. We observed that, in any of these configurations, the maximum sustainable value of λ for the data platform was equal to 43 transactions/sec. I.e., with higher values of λ the system in a short time reached the saturation point, giving rise to transaction response times which constantly increased. This confirmed us the optimality of the choice of the controller.

We continued to execute the test by modify SLAs. Specifically, we set $r_i^{max} = 0.15sec$ for $1 \leq i \leq 5$, thus entailing a violation of the constraint $r_3 \leq r_3^{max}$ (we recall that in the last configuration decided by the controller r_3 was equal to $0.344sec.$). After this modification, the configuration decided by the controller at the end of the next period was $n = 16$, $g = 15$ and $k = 6$. Hence, next measures of the average transaction response times were $r_1 = 0.012$, $r_2 = 0.090$, $r_3 = 0.126$, $r_4 = 0.016$, $r_5 = 0.02$, and m

was equal to $998MB$. We can observe that, again, none of these values violates related constraints. For evaluating the optimality of the decision, also in this case we measured the average transaction response times for configurations for which $n < 16$ and $g < n$. We note that, as for the required memory, also for all these configurations in the worst case, i.e. with $n = 15$ and $g = 14$, m was equal to $994MB$. Additionally, for these configurations we observed that, fixed the number of cache servers, say j, the minimum value of r_3 (which was the only one violating the associated SLA in the current configuration) was achieved with a data object replication degree equal to $j - 1$ for any j. This can be justified by the fact that the higher replication degree is the less remote accesses to data objects are required by transactions (because the probability to find a local replica of the accessed data object is greater). Anyway, this is not generally true as concerns the response time of the commit operation of transactions, because the probability that more cache servers are involved in a commit operation is higher when the replication degree is higher. However, in our case, being r_3 the average transaction response time of Order-Status transaction, which is a read-only transaction, no write operations on any (remote) cache server are executed at commit time. In Figure 3 we show values of r_3 we measured for all configurations such that $7 \leq n \leq 15$ and for which g was equal to $n - 1$. As we can see, value of r_3 progressively decreases as the number of cache servers increases. However, in any case the constraint $r_3 \leq r_3^{max}$ is violated. Again, this confirms the optimality of the decision taken by the controller.

Finally, we changed the transaction mix, thus setting $f_1 = 0.2$, $f_2 = 0.4$, $f_3 = 0.18$, $f_4 = 0.16$ and $f_5 = 0.06$. Using this mix, the new average transaction response times were $r_1 = 0.016$, $r_2 = 0.091$, $r_3 = 0.152$, $r_4 = 0.018$, $r_5 = 0.017$. Again, the constraint $r_3 \leq r_3^{max}$ is violated. In this case, the controller did not find a configuration for which all constraints were satisfied. Specifically, none of the possible configurations was expected to satisfy the constraint $r_3 \leq r_3^{max}$. In our implementation no action is carried out by the controller in these cases, because these situations are out of the main scope of our work. Anyway, in order to validate the prediction of the controller, we measured the average transaction response times also for the configurations that we did not check before, i.e. from 17 up to 20 cache servers. Firstly, we recall that, also in these cases, configurations with full replication were not possible due to the memory constraint. Additionally, among the above configurations, also for those with $g = n-1$ the constraint $m < m^{max}$ was not satisfied. In fact, in the best case, i.e. with $n = 17$ and $g = 16$ the estimated amount of required memory was equal to $1.003GB$. As for other admitted configurations, i.e. with $g < n-1$, we measured higher average transaction response times with respect to the last configuration decided by the controller (i.e. $n = 16$, $g = 15$ and $k = 6$). This can be due to the fact that with the above-mentioned configurations for which $g < n - 1$ a higher fraction of operations of transactions requires remote accesses to data objects with respect to the case with $n = 16$, $g = 15$. Thus, this confirmed us the proper prediction of the controller.

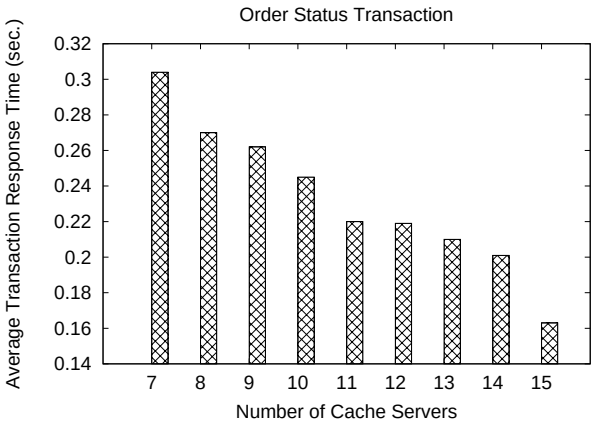

Figure 3: Average Order Status transaction response times vs. number of cache servers.

VII. Conclusions

In this paper we presented a self-configuring architecture tailored for in-memory transactional data grids. These data platforms have revealed particularly suited for taking advantage from both elastic cloud architectures and the so-called "pay-as-you-go" pricing model offered by cloud infrastructure providers. The goal of our architecture is guaranteeing that SLAs established for each transaction class of an application are not violated, and that, at the same time, the minimum amount of (virtual) hardware resources are used by the system in order to minimize the infrastructure cost. Our architecture exploits a neural network-based performance predictor, which provides estimations of transaction response times depending on the application workload profile and the configuration of the data platform. We evaluated the effectiveness of our approach through an experimental study we conducted on top of a real cloud infrastructure. Generally, our approach is simple to use and, being completely black-box, it can be easily exploited without requiring detailed expertise about internals of system. As a future direction of our work, we planned to also investigate other kind of scenarios where the initial training of neural networks is not possible, or only a partial training is possible. This can be the case of many real scenarios where, e.g., it could be complex to simulate the behavior of users in order to generate the expected workload profiles. In this case, the training set could be built while the application runs, so that the neural networks are incrementally trained. An initial exploration phase, where system configurations are randomly selected, could be exploited. Additionally, run-time prediction error measurements could be used to trigger new training phases of the neural networks using training sets with new samples gathered at-run time. Issues to be explored in this kind of

scenarios include, e.g., effective techniques to perform the initial exploration of the configuration space, as well as for avoiding that continuous updates to neural networks may lead to unstable behavior of the controller, such as continuous oscillations along a set of data platform configurations, which may cause instability of the system. Finally, we note that in our system model we did not assumed the presence of specific performance optimization mechanisms which can be exploited in in-memory transactional data grids. These include, e.g., mechanisms leveraging transaction migration for exploiting data locality (as proposed in [19]), or placement of replicas of data objects on the basis of locality patterns in data accesses of transactions (see, e.g., [20]). Addressing the presence of these kinds of optimizations could be a further extension of our work.

REFERENCES

[1] Red Hat / JBoss. JBoss Infinispan. http://www.jboss.org/infinispan, 2011.

[2] WMware. VMware vFabric GemFire. http://www.vmware.com/products/application-platform/vfabric-gemfire/overview.html.

[3] Oracle. Orache Coherence. http://www.oracle.com/technetwork/middleware/co-herence/overview/index.html, 2011.

[4] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44, 2010.

[5] Pierangelo Di Sanzo, Diego Rughetti, Bruno Ciciani, and Francesco Quaglia. Auto-tuning of cloud-based in-memory transactional data grids via machine learning. In *Proceedings of the 2nd IEEE Symposium on Network Cloud Computing and Applications*, NCCA. IEEE Computer Society, December 2012.

[6] https://portal.futuregrid.org.

[7] Zhikui Wang, Xiaoyun Zhu, and Sharad Singhal. Utilization and slo-based control for dynamic sizing of resource partitions. In *DSOM*, 2005.

[8] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, 2009.

[9] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, 2010.

[10] Jin Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '06, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Saeed Ghanbari, Gokul Soundararajan, Jin Chen, and Cristiana Amza. Adaptive learning of metric correlations for temperature-aware database provisioning. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '07, pages 26–, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Rahul Singh, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '10, pages 21–30, New York, NY, USA, 2010. ACM.

[13] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Calton Pu, and Hakan HacigümüŞ. Activesla: a profit-oriented admission control framework for database-as-a-service providers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. ACM.

[14] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional auto scaler: elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 125–134, New York, NY, USA, 2012. ACM.

[15] Diego Didona, Pascal Felber, Diego Harmanci, Paolo Romano, and Joerg Schenker. Identifying the optimal level of parallelism in transactional memory systems. In *Proc. International Conference on Networked Systems*, NETYS. Springer, 2013.

[16] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[17] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[18] http://leenissen.dk/fann/wp/.

[19] Danny Hendler, Alex Naiman, Sebastiano Peluso, Francesco Quaglia, Paolo Romano, and Adi Suissa. Exploiting locality in lease-based replicated transactional memory via task migration. *CoRR*, abs/1308.2147, 2013.

[20] João Paiva, Pedro Ruivo, Paolo Romano, and Luís Rodrigues. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. In *Presented as part of the 10th International Conference on Autonomic Computing*, pages 119–131, Berkeley, CA, 2013. USENIX.