A Machine Learning Approach to Performance Prediction of Total Order Broadcast Protocols(*)

Maria Couceiro
INESC-ID / IST
mcouceiro@gsd.inesc-id.pt

Paolo Romano
INESC-ID
romanop@gsd.inesc-id.pt

Luís Rodrigues
INESC-ID / IST
ler@ist.utl.pt

***Abstract*—Total Order Broadcast (TOB) is a fundamental building block at the core of a number of strongly consistent, fault-tolerant replication schemes. While it is widely known that the performance of existing TOB algorithms varies greatly depending on the workload and deployment scenarios, the problem of how to forecast their performance in realistic settings is, at current date, still largely unexplored.**

In this paper we address this problem by exploring the possibility of leveraging on machine learning techniques for building, in a fully decentralized fashion, performance models of TOB protocols. Based on an extensive experimental study considering heterogeneous workloads and multiple TOB protocols, we assess the accuracy and efficiency of alternative machine learning methods including neural networks, support vector machines, and decision tree-based regression models. We propose two heuristics for the feature selection phase that allow to reduce its execution time up to two orders of magnitude incurring in a very limited loss of prediction accuracy.

***Keywords*-Total Order Broadcast; Performance Prediction; Machine Learning**

I. Introduction

Total Order Broadcast (TOB) [5] is a fundamental building block for developing strongly consistent replicated systems. TOB greatly simplifies the development of fault-tolerant applications by ensuring that messages are delivered at all replicas in the same order despite variable communication delays and the occurrence of failures, hiding the issues associated with enforcing system-wide agreement on the streams of updates generated by the replicas. TOB is, in fact, at the heart of the classic, general-purpose, active replication scheme [25], as well as of a number of specialized replication protocols tailored for, e.g., database systems [19] and transactional memories [3].

Over the last decades, a wide body of literature has been devoted to the design and evaluation of TOB protocols (extensively surveyed by Defago et al. [5]). However, we are not aware of any work proposing engineering methods and tools capable of providing real-time, fine-grained (i.e. on a per message basis) forecasts of the performance of TOB protocols, when deployed in real systems and subject to complex workloads.

(*) This work was partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and Aristos project (PTDC/EIA-EIA/102496/2008) and by the European Commission through the Cloud-TM project (FP7-257784).

In this paper, we investigate, to the best of our knowledge for the first time in literature, the challenges associated with using machine learning techniques to derive fine-grained performance prediction models of TOB protocols. The machine-learning based approach proposed in this paper allows forecasting the TOB's latency on a per message basis, providing a fundamental building block for architecting self-optimizing replication schemes [23]. This is a promising research area that is, at current date, still largely unexplored precisely because of the unavailability of effective TOB's performance predictors.

We start by presenting a semi-opaque self-monitoring architecture that relies on the tracing of a basic set of protocol independent performance metrics, possibly augmented with protocol specific context information in a modular fashion via the use of standard programmatic interfaces. Our generic (i.e. protocol independent) monitoring tools track the usage of system resources (such as network bandwidth, CPU and memory) across multiple time scales spanning several orders of magnitude. This allows to combine information representative of stationary phenomena (captured by long term averages) as well as transient burstiness (captured by short term averages) which can significantly affect the latency of ongoing TOBs.

Next we discuss the results of an extensive experimental study based on:

- three machine learning methods, namely neural networks [12], support vector regression [26], and regression decision trees [21].
- three highly heterogeneous and demanding (in terms of amount of injected traffic) workloads, consisting of a synthetic traffic generator that allows us to widely span in the workload's parameters space, and two complex applications running on top a distributed software transactional memory platform [3] that generate high contention on the computational and memory resources locally available at each node.
- two different TOB algorithms relying on radically different approaches for establishing agreement on the delivery order (centralized vs distributed) and aiming at optimizing distinct performance metrics (latency vs throughput).

Our experimental results highlight that the set of context information (also called features in the machine learning literature and in the remainder of this paper) that maximizes the machine learners accuracy varies significantly when one considers heterogeneous, realistic workloads. We also evaluate to what extent incorporating time series, protocol dependant information and garbage collection metrics can allow enhancing the accuracy of the machine learners.

We then focus on the issue of feature selection, a problem of combinatorial nature that becomes rapidly intractable in scenarios, such as the one evaluated in this paper, characterized by a large abundance, and redundancy, of input variables. Our experimental data highlights that, while being certainly more efficient than an exhaustive exploration of the feature space, existing heuristics approaches [11] to the feature selection problem still have prohibitively high execution times. This can represent a major impairment in scenarios demanding frequent re-training of the performance predictors, due to, e.g., workload fluctuations or alterations of the group size caused by failures or dynamic expansions/contractions triggered by spikes of the load pressure. To tackle this issue we propose and evaluate two alternative solutions:

1) An optimized search heuristic, whose search trajectory in the features' power set is drastically restricted with respect to classical greedy search heuristics. This is achieved by exploring exclusively the combinations of features which were found to generally maximize the accuracy of the machine learners. Such a specialization allows reducing the feature selection execution time on average by a factor 10x at a negligible cost in terms of accuracy degradation ($<2\%$) across the whole spectrum of considered workloads.
2) A technique based on the ensemble of a small set of models, each one relying on different (and largely non-overlapping) subsets of features, and whose predictions are combined on the basis of the expected confidence intervals of the individual models to operate in the corresponding region of their features space. When compared with classical greedy heuristics for feature selection, this ensemble technique allows boosting feature selection by two orders of magnitude, at the cost of an average 10% degradation of the prediction accuracy.

The remainder of this paper is structured as follows. In Section II we present the architecture of our system, discussing the key implementation issues of our real-time monitoring tools. Section III overviews the machine learners, the workloads and the TOB algorithms used in our evaluation study. The results of the experimental evaluation are discussed in Section IV. In Section V we discuss related work. Finally, Section VI concludes the paper.

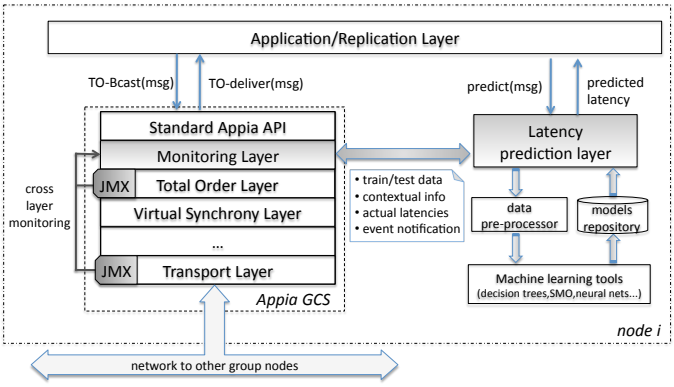

Figure 1. Architectural Overview (Single Node Perspective).

II. System Architecture

The architecture of our system is depicted in Figure 1. It considers a distributed application, such as a transactional database replication manager [19] or a distributed transactional memory [3], which is supported by a group communication service (GCS) [15]. The system is augmented with a monitoring layer and a latency predictor, which are the key contributions of this work. Before detailing the description of these components, we will first provide a brief overview of the interdependencies among the system components and discuss some key principles underlying their design.

In our system, each node develops, in an independent and fully distributed fashion, predictive models of the TOB latency as observed by applications residing on that same node. More specifically, the latency predictor component provides the client of the TOB service with a latency estimator. The predictor is able to forecast the time it takes to self-deliver (after being totally ordered) a message of a given size sent by the application.

The monitoring layer is the component responsible for collecting training data for the machine learners, as well as to provide the latency predictor with information concerning the actual workload characteristics and resource utilization levels. Additionally, it provides feedback to the latency predictor component regarding the accuracy of its forecasts, as well as notifications on the occurrence of relevant changes in the system configuration that may affect the quality of the currently employed predictive model, for instance, a change in the number of active replicas (as the performance of TOB is typically a function of the number of participants).

A. Monitoring Layer

Our monitoring layer has been developed for the Appia [15] GCS. Appia follows an architectural design that allows to compose layered stacks of micro-protocols according to the application needs. The flow of information among the layers of the Appia stack is supported by the exchange of events that are propagated upwards and downwards through

the stack. In Appia, this flow of events is regulated by a single, dedicated thread which we will refer to in the following as Event Scheduler (ES) thread.

The monitoring mechanisms are implemented as a layer that can be transparently disabled/enabled at run-time, ensuring that there is no monitoring overhead when the tracing functionality is disabled. As depicted in Figure 1, the monitoring layer sits between the TOB layer and the interface towards the application. This allows to achieve total transparency for the application, as well as to straightforwardly trace any event generated by or delivered to the application. Thus, the monitoring later is able to intercept TO broadcast/delivery events and events notifying of changes in the group membership. As noted before, membership changes may have a significant impact on the TOB performance, thus they can be used to trigger the generation of a new performance model.

When the monitoring layer is enabled, it collects context information using the following set of metrics:

1) *Network related metrics:* moving averages across multiple time scales of i) the number of TO broadcast/delivery events, and of ii) the amount of bytes sent/received by the TO layer; additionally it keeps track of the number of TO broadcast events generated by the application layer and for which it has not been generated the corresponding TO delivery event yet.

2) *CPU related metrics:* moving averages across multiple time scales of the total CPU utilization, and of the CPU utilization of Appia's ES thread.

3) *Memory related metrics:* the free memory in the Java Virtual Machine (JVM), as well as two metrics describing the activity of the JVM's Garbage Collector (GC) thread namely, i) the time occurred since the last garbage collection cycle, and ii) the percentage of time elapsed since the last garbage collection cycle with respect to the time between the last two garbage collection cycles. Note that, since there is no standard Java API to directly track the status of the GC thread, to trace the GC activity in a portable manner we extended the *finalize()* method of a dummy object to keep track of the time in which the GC thread is activated (and re-instantiate the dummy object).

In addition to the above context information, the monitoring layer has been designed to support also cross-layer tracing in a modular fashion. Specifically, at system's bootstrap, and upon any alteration of the Appia stack, the monitoring layer queries the whole set of Appia's layers via the standard Java Management Extensions (JMX) interface to determine whether there are any layers that externalize information related to their internal state that could be exploited by the machine learners to generate more accurate performance models. Our approach allows developers to specify which attributes, among those monitorable via the JMX interface, should be traced by our monitoring layer as deemed potentially beneficial to enhance the machine learners' accuracy. As we will further discuss in Section III, in our experimental analysis we exploit this mechanism to monitor the number of outgoing message queued at the lower layer of the Appia's stack (namely the Transport Layer), as well as to track the state of internal variables of TOB algorithms. We report the whole set of metrics gathered by the monitoring layer in Table I.

Metric	**Description**
freeMem	Free memory in the Java Virtual Machine
tLGC	The time since the last garbage collection
pLGC	% of time since the last GC cycle w.r.t. the time between the last 2 GC cycles
undelivMsgs	#TO Broadcast msgs and not yet self-delivered
bytesUp_x	#Bytes received over a x msec. time window
bytesDown_x	#Bytes sent over a x msec. time window
TOBUp_x	#TOB deliver events over a x msec. time window
TOBDown_x	#TOB broadcast events over a x msec. time window
totCPU_x	% total CPU utilization over a x msec. time window
esCPU_x	% CPU utilization by ES thread over a x msec. time window
esCPU_x	% CPU utilization by ES thread over a x msec. time window
TCPqueue	Outgoing messages queued at the Transport Layer *(protocol dependant metric traced via JMX interface)*
toTime	Elapsed time since the token was last owned *(protocol dependant metric traced via JMX interface)*

Table I
LIST OF METRICS COLLECTED BY THE MONITORING LAYER.

Whenever a TO broadcast event for message m is intercepted by the monitoring layer, the latter takes a snapshot of the current state of the context information. As soon as the monitoring layer intercepts the TO delivery event for message m, it determines the self-delivery latency, logs the associated context information (namely the context information at m's sending time along with its self-delivery latency) asynchronously to a memory buffered file and propagates the TO delivery event upwards. The choice of measuring exclusively the self-delivery latencies allows to circumvent the issue of ensuring accurate clock synchronization among the communicating nodes, which would have clearly been a crucial requirement in case we had opted for monitoring the delivery latencies of messages generated by different nodes. Preliminary experiments conducted in our cluster have indeed highlighted that the accuracy achievable using conventional clock synchronization schemes, such as NTP, is often inadequate for collecting meaningful measurements of the TO broadcast inter-nodes delivery latency, being the latter frequently around or less than a millisecond.

B. Latency Predictor

The latency predictor is responsible for forecasting the performance of the TOB layer when broadcasting a message of a given size. In order to build the performance models used to generate these forecasts, the latency predictor layer triggers the pre-processing of the training data collected by the monitoring layer. In this phase, the training data is prepared (properly filtered and manipulated, e.g., to generate time-series - see Section IV-A) to allow its successful

processing by the chosen machine learning tool. Our system architecture in fact supports the modular integration of alternative machine learning libraries.

The models output by the machine learners are stored in a model repository, where they are associated with metadata that captures the context in which these models were built (such as the number of machines participating in the TOB group and the TOB algorithm employed while generating the training data). Furthermore, the models are ranked (e.g. for feature selection purposes) and made available to the latency predictor layer for generating performance predictions. Note that the performance models output by the machine learners take as input features not only the size of the message to be broadcast, but also a (possibly quite large) set of system metrics. These are obtained by querying at run-time the monitoring layer. The latter makes also available information on the actual self-delivery latencies of recently broadcast messages, which can be used by the latency predictor to assess the accuracy of its predictions and possibly trigger the construction of a new model.

III. Testbed Description

A. Overview of the Evaluated Machine Learning Methods

At current date, we have integrated in our system two machine learning tools, namely Rulequest's Cubist© [20] and Weka [8], enabling the user to choose which one to use. We now briefly overview these two tools.

Cubist© is a decision tree based regression commercial tool developed by Quinlan, the author of C4.5 [22] and ID3, two popular decision tree based classifiers. Analogously to these algorithms, Cubist© builds decision trees choosing the branching attribute such that the resulting split maximizes the normalized information gain (namely the difference in entropy). Unlike C4.5 and ID3, which contain an element in a finite discrete domain (i.e. the predicted class) as leafs of the decision tree, Cubist© places a multivariate linear model at each leaf. An appealing characteristic of Cubist© is that the decision tree can be reformulated as set of human-readable rules, where each rule identifies a region in the feature space. Also, each rule contains a multivariate linear model in the "then" clause and is associated with the expected average error in the prediction. Since Cubist© generates a piecewise regression model (each multivariate linear model being applicable under certain rules), it can be more powerful than a simple multivariate linear model as it allows variables to be weighted differently as conditions change. When two rules overlap, the values predicted by using the models associated with each rule are averaged with a weight that depends on the degree of confidence in the prediction generated by the two rules.

Weka is an open-source framework providing a common interface to a large number of machine learning algorithms. In this work we evaluate two major regression techniques, namely, Neural Networks [12] and Support Vector Machines [26]. These methods are well-known and have been extensively described in the machine learning literature, so, due to space constraints, we will only briefly overview them. The neural network algorithm implemented in the Weka framework trains a multi-layered network using the classic back-propagation algorithm [24] to determine the weights that minimize the local error at each perceptron. We used the default configuration in Weka, which generates a number of hidden layers equals to half the number of input features. Concerning the Support Vector Machine technique, we also rely on the default configuration of the Weka's SMOreg package, which uses a polynomial kernel whose parameters are learnt using the algorithm in [26].

B. Workload Description

For our experimental study we consider the following three workloads:

Synth: this is a synthetic benchmark which injects traffic at each node following a regular and homogeneous pattern. On each node we run a single application level thread which TO broadcasts, during intervals lasting 30 seconds each, messages of growing size, namely {100, 200, 500, 1K, 2K, 5K, 10K, 20K, 50K, 100K, 200K, 500K} bytes, at an increasing sending rate, namely {1, 2, 10, 20, 50, 100, 125, 166, 333, 500, 1000} messages per second. The training and testing datasets are built collecting for each configuration at most 90 messages.

RBTree: this workload is generated by D^{2}STM [3], a distributed software transactional memory platform, running the Red Black Tree benchmark. D^{2}STM uses a TOB-based distributed certification scheme that relies on TOB to propagate the readset and writeset of local transactions, and ensure that all replicas validate transactions in the same common order. The Red Black Tree [13] is a well-know benchmark for the evaluation of software transactional memories, in which a red black tree data structure is concurrently updated (by inserting and/or removing items) by several threads. The benchmark was ported to run on the D^{2}STM platform and tuned to generate transactions entailing a variable number of operations. Note that this has the effect of further increasing the heterogeneity of the generated workload: as the number of operations issued by each transaction varies over time, the frequency of generation of TO broadcasts, and the size of the TO broadcast messages (which encode the transactions' readset and writeset) also vary accordingly. Unlike the Synth benchmark, in RBTree each replica can host a variable number of threads performing computational intensive tasks before issuing a TO broadcast. This scenario is therefore characterized by a much higher contention among (GCS and application) threads on the local resources (CPU in primis). We will see that this has a significant impact on the predictability of the GCS performance. The data set used to train and test the machine learner consists of the messages exchanged during 15 minutes, which corresponds to the time

Worst vs Best Set of Feat.	Benchmark	NAE
6 vs 50 time window	Synth (m4)	25%
500 vs 10 time window	RBTree (m2t3)	32%
without vs with GC	Synth (m4)	29%
without vs with time serie	RBTree (m2t3)	81%
without vs with token	RBTree (m2t1)	61%

Table II
MODELS' ACCURACY INCREASE (NAE) WITH SELECTED FEATURES.

needed to collect training data across the whole range of the generated workload.

STMBench7: this workload is also generated by D^{2}STM, running STMBench7 [10], a complex benchmark which manipulates an object-graph with millions of objects, featuring a number of operations with different levels of complexity. This benchmark includes both very short and very long-running transactions; the latter traverse hundreds of thousands of objects and generate extremely large readsets and writesets. As a consequence, the workload for the TO service entails both very short (on the order of few hundreds of bytes) and very large messages (on the order of several megabytes). Also, as transactions need to store in memory their readset and writesets, long-running transactions end up stressing significantly the memory system of the local JVM, triggering frequent garbage collection cycles. Like in the RBTree benchmark, this benchmark allows running multiple concurrent application level threads in each node. Each run of this benchmark lasts around 30 minutes in order to ensure that, independently of the number of machines and threads, the training and testing data sets contain approximately 12.000 entries.

C. Evaluated TOB Algorithms

In our experiments, we consider two classic, well-known TOB algorithms, described, e.g., by Defago et al. in [5]. The first one is a sequencer-based algorithm in which a single node, called the sequencer, determines the order according to which all nodes have to TO deliver messages. The second one is a token-based algorithm which ensures agreement on the TO delivery order by circulating among the nodes a token that grants the right to broadcast messages.

The choice of these two algorithms was aimed at maximizing diversity, with the ultimate purpose of widening the representativeness of our testbed. The above TO algorithms, in fact, rely on extremely different approaches for establishing agreement on the delivery order (centralized vs distributed), aim at optimizing different performance metrics (latency vs throughput), and have complementary pros and cons [5].

IV. ANALYSIS OF THE RESULTS

In this section we present the results of our experimental study. We will initially focus on the analysis of the results obtained using Cubist© and only subsequently move to compare the performance of the Neural Network and SMO methods. All the results reported in the following were obtained using a testbed of nodes equipped with an Intel QuadCore Q6600 at 2.40GHz with 8 GB of RAM running Linux 2.6.27.7 and interconnected via a private Gigabit Ethernet.

The accuracy of the machine learners is measured using the following metrics. Relative Average Error (RAE), which compares the performance of the predictor with that of a naive predictor that simply outputs the average value of the training data. When comparing two different models, say M_1 and M_2, we will rely on the Normalized Additional Mean Absolute Error (NAE), defined as $NAE = \frac{MAE_{M_1} - MAE_{M_2}}{Lat_{avg}}$, namely the difference between the Mean Absolute Error (MAE) of model M_1 (MAE_{M_1}) and the MAE of model M_2 (MAE_{M_2}) normalized by the average value of the delivery latency in the test set data (Lat_{avg}). The NAE is a scale-free metric that we deem more informative than a simple comparison between the MAEs of M_1 and M_2. In fact, small differences between the MAEs are irrelevant if the delivery latencies are, on average, large, whereas, small differences between the MAEs are relevant if, on average, delivery latencies are also small.

Finally, to assess the models' accuracy we rely on twofold cross-validation, using 60% of the available data to build the model during the training phase and the remaining 40% as test data.

A. What features to use?

A crucial challenge that has to be faced for accurately predicting the performance of any complex system via machine learning techniques is to carefully identify the set of metrics/context information to be used as input variables for the model construction [11].

One of the first problems that we had to address while building our system was related to the difficulty to identify an optimal time window for computing the moving averages concerning the percentage of utilization of CPU and network resources. Our experimental results accentuated the fact that the choice of the wrong time window could significantly affect the machine learners' accuracy. This phenomenon is clearly highlighted by the first two rows in Table II, which compares the accuracy of the predictions when using specific features when the Token algorithm is being used to disseminate messages in a group of 4 machines. The fact that the Synth workload is stable over relatively long periods of time explains the 25% decrease in accuracy (measured through the NAE) when using a time window of 6 msec rather than 50 msec. On the other hand, an opposite result is obtained when considering the RBTree workload, where the accuracy decreases by 32% when using 500 msec, rather than 10 msec, time windows. This can be explained considering that shorter time windows are more sensitive to transient burstiness phenomena; this can be beneficial

	Sequencer								Token							
	2 Machines				4 Machines				2 Machines				4 Machines			
	1Thread		3Threads		1Thread		3Threads		1Thread		3Threads		1Thread		3Threads	
	COR	RAE	COR	RAE	COR	RAE	COR	RAE	COR	RAE	COR	RAE	COR	RAE	COR	RAE
Synth	1.00	0.01	-	-	1.00	0.20	-	-	1.00	0.01	-	-	1.00	0.01	-	-
RBTree	0.44	0.42	0.63	0.39	0.69	0.37	0.77	0.42	0.80	0.31	0.95	0.12	0.57	0.49	0.84	0.30
STMB7	0.44	0.37	0.64	0.35	0.67	0.35	0.54	0.36	0.65	0.38	0.71	0.36	0.49	0.51	0.74	0.42

Table III
CORRELATION COEFFICIENT AND RELATIVE ABSOLUTE ERROR OF CUBIST© USING FORWARD SELECTION.

in presence of highly variable workloads, such as RBtree, but disadvantageous in the case of more stable workloads, such as for Synth. These results have led us to the choice of computing the moving averages across multiple time windows, ranging from 2 up to 500 msecs, and of relying on feature selection phase to filter out the ones that turned out to be uninformative or misleading for the machine learner.

Another interested finding highlighted in Table II is related to the relevance of the GC related metrics. The third row of the table reports a degradation of the model's accuracy of 29% for the Synth workload when the metric that indicates the time elapsed since the last garbage collection cycle (tLGC in Table I) is not used (and otherwise using the same set of features).

These experiments have also highlighted the usefulness of incorporating time series information in the set of features used by the machine learners. To this end we pre-process the training data generated by the monitoring layer in order to include, in the set of features provided to the machine learner, the latencies of the last k TO broadcasts self-delivered by that node. As shown by the 4th row of table II, when time series information is not employed, the Cubist's© predictions accuracy increases by 81% in the Synth workload scenario. This is due to the fact that, especially in the less fluctuating workloads, there is often a significant correlation among the delivery latencies of recently broadcast messages.

Finally, the last row in Table II reports a 61% decrease in the accuracy for the RBTree workload when the machine learner is not provided with information concerning the elapsed time since the token was owned by the node for the last time (toTime in Table I). In the token-based algorithm, in fact, the delivery latency is strongly affected by the time elapsed before the token is owned by the sending node, and the latter is closely correlated with toTime. Overall, these results confirm the relevance of our semi-opaque monitoring approach, which provides the TOB layers' developers with standard interfaces to instruct the monitoring layer to trace protocol-dependant state information.

Based on the above analysis, we identified a total number of 53 relevant features (43 of which being directly traced by our monitoring layer, and 10 additional ones obtained by building a time serie on the delivery latency). When faced with such an abundance (and redundancy) of available metrics, it is easy to fall prey of the, so called, curse of dimensionality [17]. As the number of dimensions in the feature space (i.e. the degrees of freedom of the model to be built by the machine learner) increases, in fact, the amount of training data required to ensure an equivalently dense sampling coverage of the feature space grows exponentially. This makes the machine learners much more exposed to the risk of overfitting [6], a phenomenon in which the machine learner infers erroneous dependencies among random features of the training data with no causal relation to the target function, with the result of increasing their accuracy in fitting known data (hindsight) while actually degrading the accuracy in predicting new data (foresight).

Note that the feature selection problem is of combinatorial nature, as identifying optimal solutions entails exhaustive searching the powerset of the feature set. This motivated the design of a number of alternative heuristic approaches that enhance efficiency at the cost of not achieving optimality. In the machine learning literature, greedy algorithms are probably the most used for implementing feature selection. There are two variants of this approach: forward selection (FS) and backward elimination (BE). In FS, features are progressively added to build larger models, whereas in BE one starts with the set of all features and progressively eliminates the least promising ones. At each iteration, one feature is added/removed and cross-validation is used to identify the best performing subsets of features. Both approaches stop when adding/removing one more feature to/from the remaining set of features no longer improves accuracy.

We report in Table III the Relative Absolute Error and correlation coefficient achieved by using the FS heuristic across all the considered workloads (we omit report results for BE as they are extremely close to those achieved by FS). The plots are relative to scenarios where the number of machines varies between 2 and 4, and the number of threads in the RBtree and STMBench7 benchmarks vary from 1 to 3. Results show that the prediction accuracy clearly depends on the complexity of the considered workload. When considering the Synth workload, the accuracy and correlation of the predictor output are extremely high, even if this workload is highly heterogeneous and encompasses phases where nodes generate both very low and high network traffic. This happens because, in each phase, the fluctuations of the delivery latencies are rather limited. The reasons for the

	FS	BE	OSH	ENS
# Models Built	401	484	72	7
Time (sec)	250	579	53	2.1

Table IV
AVERAGE EXECUTION TIME OF FEATURE SELECTION ALGORITHMS.

observed stability in each phase are twofold. First, nodes are very lightly loaded, not running any computational or memory intensive tasks. Second, nodes send messages at the same rate, which makes the performance of the GCS in each phase rather stable.

The other two considered workloads are, on the other hand, definitely more challenging. First, the applications lack a well defined traffic pattern and second, nodes execute computational intensive applications. Together, these factors induce a strong variance in the self-delivery latencies. As a result, even though the set of features selected by the FS algorithm varies significantly for each workload, the performance of the predictors is similar across the two workloads, with an average correlation factor of 66% and an average RAE of 37%.

The results collected with these two workloads show an interesting trend, namely, the correlation generally grows, on average, from 59% to 73% when moving from scenarios with one thread to scenarios with three threads. This correlation is mainly due to the fact that in this context the undelivMsgs feature (which captures the number of threads that are blocked waiting for the self-delivery of a message, see Table I) becomes extremely useful in this context. Indirectly, this metric provides a measure of congestion in the system. On the other hand, this feature is only meaningful when there are at least two application level threads, as it is constantly equal to 0 in case there is a single application level thread.

B. Boosting Feature Selection

Unfortunately, despite being significantly more efficient than exhaustive searches, the FS and BE heuristics still demand the construction of hundreds of models (see the first two columns from left of Table IV) and require execution times on the order of the hundreds of seconds even on a fast (at the time of writing) machine equipped with two quad-core 2.33 Ghz Intel Xeon processors, 4 GB of RAM and running Linux 2.6.27[1]. These costs are clearly prohibitive in scenarios where models may have to be re-built rapidly to adapt to workload fluctuations.

To tackle this issue, we propose and evaluate the following two techniques:

[1]Interestingly, the average performance of BE is significantly worse (by a factor 2.3) with respect to that of FS even though the latter builds, on average, only 20% less models than than the former. This is due to the fact that the models built by BE have, on average, a larger number of features with respect to those explored by FS, and that the time taken by Cubist© to build a model is strongly affected by the number of features it uses.

- *OSH:* An Optimized Search Heuristic (OSH) that evaluates only combinations of features that were preselected based on a preliminary exhaustive experimentation across the whole spectrum of workloads with classical statistical tools, such as Primary Component Analysis [18], and cross-validation testing. This preliminary phase allowed us to identify and discard the combinations of features whose usage either provided negligible increases, or even deterioration of the prediction accuracy.
  OSH explores a total of 72 different models built by using as input features, a common set of attributes (namely, msg_size, TCPqueue, and undelivMsgS) and the combinations obtained by picking exactly one item from the following sets:
  - T={latency of the last TO broadcast, latencies of the last 5 TO broadcasts, latencies of the last 10 TO broadcasts};
  - M={no memory information, freeMem, freeMem and tLGC, freeMem and pLGC};
  - R={moving_avgs$_x$}, where moving_avgs$_x$ denotes the following set of metrics {bytesUP$_x$, bytesDOWN$_x$, TOBUp$_x$, TOBDown$_x$, totCPU$_x$, esCPU$_x$} computed over the same time window of duration x msecs, and $x \in$ {2,6,10,50,100,500} msecs.
- *Ensemble:* An ensemble of independent models built over largely non-overlapping sets of features and whose predictions are reconciliated on the basis of their estimated confidence interval. The intuition underlying this approach is that models built using diverse set of features have the potentiality to capture distinct phenomena affecting the delivery latency of TOB algorithms with different degrees of accuracy. In addition, by focusing each model on a smaller subset of attributes, they are less prone to suffer of overfitting problems. Further, by selecting the prediction generated by the model with the highest degree of confidence in the current region of the feature space, our ensemble technique may enhance the accuracy of each independent model.
  Our ensemble technique generates 7 models, where each model uses the same common set of attributes as in OSH (msg_size, TCPqueue, and undelivMsgS), but differs from the other ones as it uses either i) a time-series containing the last k (where we set $k = 10$) TOB latencies, or ii) moving_avgs$_x$ computed as before. In preliminary experiments we have evaluated several alternative methods for conciliating the predictions provided by the various models. We only report results for the best performing strategy, which is based on the simple approach of selecting the prediction associated with the smallest confidence interval.

	Sequencer								Token							
	2 Machines				4 Machines				2 Machines				4 Machines			
	1Thread		3Threads		1Thread		3Threads		1Thread		3Threads		1Thread		3Threads	
	OSH	ENS	OSH	ENS	OSH	ENS	OSH	ENS	OSH	ENS	OSH	ENS	OSH	ENS	OSH	ENS
Synth	0.2%	0.7%	-	-	2.5%	14.2%	-	-	7.4%	7.5%	-	-	10.7%	10.6%	-	-
RBTree	0.2%	3.4%	1.8%	20%	0.2%	14.2%	5.3%	14.1%	3.8%	4.1%	1.6%	2.5%	7.2%	11.7%	5.4%	8.3%
STMB7	0.4%	0.7%	2.1%	19.1%	0.3%	3.9%	3.1%	18.3%	6.4%	15.3%	11.1%	17.6%	2.6%	3.6%	20.4%	22.9%

Table V
NORMALIZED ADDITIONAL MEAN ABSOLUTE (NAE) ERROR USING OSH AND ENSEMBLE.

As expected, by evaluating a much smaller number of models, OSH and Ensemble achieve striking performance gains, reducing feature selection time up to two orders of magnitude (see Table IV).

On the other hand, table V reports data quantifying to what extent the quality of the predictions deteriorate when using OSH and Ensemble with respect to the case in which FS is used. The accuracy of OSH is extremely close to that of FS, being its average NAE around 2.2%. Concerning Ensemble, the average NAE increase is larger, namely around 10%. We argue that, in practical settings, this (limited) degradation of the prediction accuracy is largely compensated by the significant performance gains it achieves. On the other hand, in these experiments, we relied on Cubist's© estimates of the confidence intervals, whose details are unfortunately not publicly available. An interesting open research question is whether the accuracy of the Ensemble technique could be enhanced by leveraging on alternative techniques, e.g. [14], for the computation of the predictions' confidence intervals.

C. Additional Performance Considerations

In Table VI we report the average overhead (measured in terms of TOB throughput reduction) due to the tracing activities carried out by the Monitoring Layer with a different number of machines/threads. The numbers show that the overhead is in practice very limited, being always less than 5%, and decreasing to 2% in the case of four machines. This can be explained by considering that, as the number of nodes in the system increases, the TO delivery latency also grows accordingly. In a closed model, such as the one characterizing both the RBTree and the STMBench7 benchmarks, this leads to a reduction of the frequency of TO broadcast issued by each node and, consequently, of the frequency of messages traced by the Monitoring Layer.

In Figure 2 we analyze to what extent the size of the training data set affects the model's prediction accuracy and building time. To this end, we considered the model using the features selected by the FS technique for the STMBench7 and Synth benchmarks, and progressively reduced the size of the training data set. The data shown in the plots is obtained by averaging the model's accuracy and building time across the whole set of configurations (number of machines/threads and considered TOB protocol) evaluated for these benchmarks (see Section III).

# Machines	# Threads	Overhead (%)
2	1	5.71
2	3	5.21
4	1	2.63
4	3	2.23

Table VI
AVERAGE OVERHEAD DUE TO MONITORING LAYER

The plots highlight a somewhat expectable, but relevant trade-off: the model building time can be significantly reduced by using smaller training data sets, at the cost of a degradation of the predictions' quality. Specifically, our experimental data show that a very similar prediction accuracy (1%, resp. 10%, higher RAE for the Synth, resp. STMBench7) could have been achieved using 50% smaller data sets, boosting the training phase by a factor larger than 2. The selection of the training data set size represents, in fact, a key tuning knob that can be used to further reduce the time required to derive the TOB performance prediction model. Unfortunately, as also confirmed by our experimental data, the optimal choice of the training data set size is highly workload dependent and is a time consuming process which is typically performed offline. Extending our system to automatize this process is an important research direction that will be pursued in our future work.

D. Alternative Machine Learners

Finally, we compare the performance and accuracy of the models obtained by using Cubist© with those generated by two other machine learning algorithms available in Weka, namely a Multilayer Neural Nework (Neural) and a Support Vector Machine regression (SMO) method. The reported results are obtained using the same set of features as input for all the machine learners, namely those selected by running the FS scheme with Cubist©.

Figure 3 reports the NAE between Neural and Cubist©, and between SMO and Cubist©, averaged across all the three considered workload. By the plot, we see that Cubist© significantly outperforms both Neural and SMO across almost every workload with the exception of the scenarios where sequencer-based protocol is evaluated using 2 machines. In these cases, the Weka's machine learners in fact achieve a lower Minimum Absolute Error with respect to Cubist©, determining an inversion of the trend highlighted

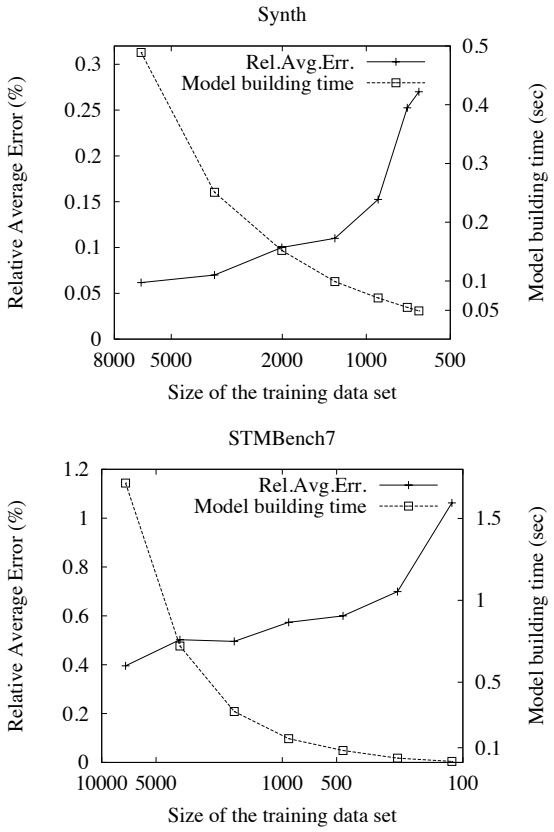

Figure 2. Evaluating model's building time and accuracy while varying the training data set's size.

by the plot. It is interesting to note, however, that in these scenarios the correlation of Neural and SMO (not shown in the plots) is significantly worse (around the half) than that of Cubist©. These differences can be motivated by considering that different machine learning approaches are known to optimize distinct metrics [17] and by referring to the well-known "No free lunch theorem" [28], which states that the performance of no single machine learner can be optimal across all possible scenarios.

As a final remark, we compared the training time of the various machine learning tools when using data set containing approximately 7.200 training cases. The results are strongly in favour of Cubist©, which on average takes 0.64 seconds to build a model, whereas the average time for completing the training phase for Neural and SMO was, respectively, 243 and 575 seconds. Albeit the performance difference is quite striking, it is not completely surprising considering that Cubist© is a commercial, and highly optimized performance tool (written entirely in C and parallelized to take advantage of multi-core CPUs), whereas Weka is an open-source framework designed to simplify development and testing of novel machine learning methods rather than fine-tuned for performance purposes.

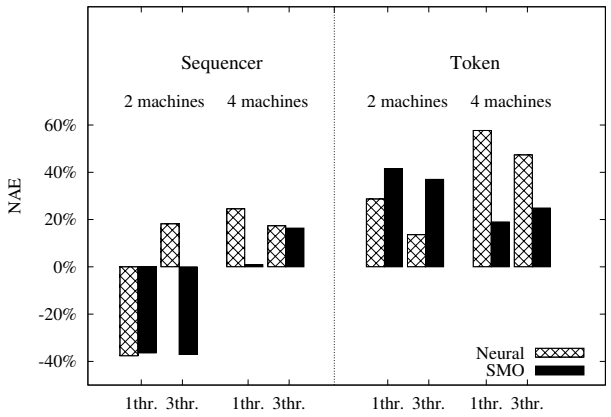

Figure 3. NAE of Neural and SMO with respect to Cubist©.

V. Related Work

Existing performance evaluation and modelling studies of TOB [4], [7] (and related agreement problems, consensus in primis [2]) have been aimed at providing a steady state estimate of the average performance of several TOB protocols in presence of simple synthetic workloads. Typically, the purpose of these approaches is to identify the most favourable settings for each of the considered algorithmic alternatives. Also, due to the inherent complexity of TOB protocols, the only analytical models of TOB we are aware of [2], [7] make rather stringent assumptions on the workload, e.g. symmetric Poissonian traffic sources generating messages at the same rate. In these works, the system model is also simplified, using synthetic (constant or exponential) communication latency distributions, that neglect important factors such as the impact of the message size on the observed latency. To the best of our knowledge, our work represents the first attempt to leverage on machine learning methods for assessing the performance of TOB protocols. Unlike existing analytical/simulation models, we leverage on statistical methods to automatically build fine-grained TOB performance models capable of forecasting in real-time the delivery latency perceived by user level applications on a per message basis.

Our work is clearly related to the machine learning literature addressing performance prediction of computer systems. These include works aiming at forecasting the throughput of TCP flows [16] and Pub-Sub systems [9], solutions aimed at automatizing the allocation of resources in cloud-computing infrastructures [29], and at generating software aging models to be used in the context of rejuvenation frameworks [1].

The idea of ensembling different machine learning models to enhance the performance of single predictors has been widely investigated in general contexts [17], as well as applied to predict performance failures of complex systems [31]. The latter work ensembles models based on the same

set of features but representative of different phases of the life cycle of applications, and dynamically selects the model which better matches the current load scenario by ranking them based on the Brier score. Conversely, our ensemble technique ensemble models built using largely disjoint sets of features capturing different aspects of the application workload with different degrees of accuracy.

Finally, our work is related with the body of literature addressing the issue of identifying the most informative attributes to be used by machine learners. In addition to the already mentioned greedy search techniques, such as Backward Elimination or Forward Selection [11], we can mention also techniques, e.g. [30], aimed at clustering highly correlated attributes for a preliminary screening of redundant metrics, or at identifying the set of attributes accounting for the greater variability in the output variable, such as Primary Component Analysis [18] and Projection Pursuit [27]. Unfortunately, these methods do not provide a direct indication of the actual accuracy achievable by the machine learner and rely on a set of input parameters (e.g. the fraction of the total variability of the output variable should be accounted when evaluating the feature set) whose optimal settings may be not trivial to determine.

VI. Conclusion

In this paper we propose and evaluate a machine learning based approach to performance modelling of Total Order Broadcast protocols. The ability of our technique to provide fine-grained prediction on a per-message basis makes it an extremely useful building block for architecting self-optimizing replication schemes. An extensive experimental study comparing different machine learning methodologies and feature selection approaches has been presented. We also introduced two novel heuristics that drastically reduce the execution time of the feature selection phase at the cost of a very limited loss of accuracy.

References

[1] A. Andrzejak and L. Silva. Using machine learning for non-intrusive modeling and prediction of software aging. In *Proc. NOMS'08*, IEEE Press, 2008.

[2] A. Coccoli, P. Urban, A. Bondavalli, and A. Schiper. Performance Analysis of a Consensus Algorithm Combining Stochastic Activity Networks and Measurements. In *Proc. DSN '02*, IEEE Computer Society Press, 2002.

[3] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D^{2}STM: Dependable distributed software transactional memory. In *Proc. PRDC'09*, IEEE Computer Society Press, 2009.

[4] F. Cristian, R. D. Beijer, and S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering*, 1:177–201, 1994.

[5] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[6] T. Dietterich. Overfitting and undercomputing in machine learning. *ACM Comput. Surv.*, 27(3):326–327, 1995.

[7] R. Ekwall and A. Schiper. Modeling and validating the performance of atomic broadcast algorithms in high-latency networks. In *Proc. Euro-Par '07*, Lecture Notes in Computer Science, pages 574–586. Springer, 2007.

[8] E. Frank, M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, and I. H. Witten. *Weka: A machine learning workbench for data mining.*, pages 1305–1314. Springer, Berlin, 2005.

[9] L. Garces-Erice. Admission control for distributed complex responsive systems. In *Proc ISPDC '09*, pages 226–233, Washington, DC, USA, 2009. IEEE Computer Society Press.

[10] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Operating Systems Review*, 41(3):315–324, 2007.

[11] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, 2003.

[12] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1994.

[13] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.

[14] B. Jiang, X. Zhang, and T. Cai. Estimating the confidence interval for prediction errors of support vector machine classifiers. *J. Mach. Learn. Res.*, 9:521–540, 2008.

[15] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. ICDCS'01*, pages 707–710, IEEE Computer Society Press, 2001.

[16] M. Mirza, J. Sommers, P. Barford, and X. Zhu. A machine learning approach to tcp throughput prediction. In *Proc. SIGMETRICS '07*, pages 97–108, ACM Press, 2007.

[17] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[18] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572, 1901.

[19] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine Approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[20] J. R. Quinlan. Cubist. http://www.rulequest.com/cubist-info.html.

[21] J. R. Quinlan. Learning with continuous classes. pages 343–348. World Scientific, 1992.

[22] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[23] P. Romano, N. Carvalho, and L. Rodrigues. Towards distributed software transactional memory systems. In *Proc. LADIS'08*, ACM Press, 2008.

[24] D. E. Rumelhart, R. Durbin, R. Golden, and Y. Chauvin. Backpropagation: the basic theory. pages 1–34, 1995.

[25] F. B. Schneider. *Replication management using the state-machine approach*. ACM Press/Addison-Wesley Publishing Co., 1993.

[26] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, and K. R. K. Murthy. Improvements to the SMO algorithm for SVM regression. *IEEE-NN*, 11(5), IEEE Computer Society Press, 2000.

[27] J. S. Vetter and D. A. Reed. Managing performance analysis with dynamic statistical projection pursuit. In *Proc. Supercomputing '99*, ACM Press, 1999.

[28] D. H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Comput.*, 8(7):1341–1390, 1996.

[29] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing*, 11(3):213–227, 2008.

[30] L. Yang, J. M. Schopf, C. L. Dumitrescu, and I. Foster. Statistical data reduction for efficient application performance monitoring. In *Proc. CCGRID '06*, pages 327–334, IEEE Computer Society, 2006.

[31] S. Zhang, I. Cohen, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *Proc. DSN '05*, pages 644–653, IEEE Computer Society Press, 2005.