Self-tuning Batching in Total Order Broadcast Protocols via Analytical Modelling and Reinforcement Learning

Paolo Romano
INESC-ID, Lisbon, Portugal

Matteo Leonetti
Sapienza University of Rome, Italy

Abstract—**Batching is a well known technique to boost the throughput of Total Order Broadcast (TOB) protocols. Unfortunately, its manual configuration is not only a time consuming process, but also a very delicate one, as incorrect settings of the batching parameter can lead to severe performance degradation.**

In this paper we address precisely this issue, by presenting an innovative mechanism for self-tuning the batching level in TOB protocols. Our solution combines analytical modeling and reinforcement learning techniques, taking the best of these two worlds: drastic reductions of the learning time and the ability to correct inaccurate predictions by accumulating feedback from the operation of the system.

I. Introduction

Total order broadcast [1] (TOB) represents a fundamental problem in distributed systems. Informally, the TOB problem requires that a group of distributed processes reach agreement on a common order of delivery of messages, in presence of concurrent broadcasts by any process of the group.

The relevance of this problem, and of the associated algorithmic solutions, is due to the fact that it represents an essential building block that is largely employed to simplify the solution of a wide range of distributed problems.TOB, in fact, represents the cornerstone of the classic active replication technique [2], which has been employed in a range of heterogeneous application domains [3]–[6]. It also represents an incarnation of consensus [7], [8], another well-known fundamental problem of distributed systems.

The design space for TOB protocols is quite large, and an abundant number of different algorithmic solutions have been published in the literature (Defago et al. [1] provide a comprehensive survey). Existing TOB algorithms usually optimize specific performance metrics (e.g. latency vs throughput) and target systems with different degrees of synchrony [1]. Among TOB protocols, Sequencer-based TOB (STOB) algorithms are particularly attractive for their low latency. This class of algorithms relies on a single node, the sequencer, to impose a total order on the stream of messages broadcast by the group of processes. STOB algorithms have their key strength point in that they are optimal in terms of the number of communication steps necessary to establish the total order. On the down side, their main limitation is that their maximum throughput is upper bounded by the capacity of the sequencer to generate sequencing messages. In LANs, or in typical data centers, which represent the focus of this paper, the bottleneck is typically represented by the sequencer's CPU.

In order to cope with this issue, a simple, although very effective technique consists in delaying the generation of the sequencing message, so to "batch" together multiple incoming messages at the sequencer side. The sequencer then sends a single message to specify the order in which all of the other messages should be delivered by the processes in the group. By amortizing the per-message overhead, batching allows to reduce the consumption of resources, thereby boosting the throughput of the system. On the other hand, at low load, waiting for additional messages to form a batch induces unnecessary stalls that hamper the performance of the total order service. The problem is exacerbated in the presence of dynamic, fluctuating workloads. In this (in practice very common) case, the optimal batching factor actually varies over time, making static configuration policies largely suboptimal.

At current date, however, the problem of how to self-tune the batching level in STOB protocols is largely unexplored. The only solutions we are aware of, in fact, are far from being fully satisfactory as they depend on the accurate, manual, tuning of different kinds of system parameters [9]. Hence, rather than solving the problem of tuning the batching level, they actually replace it with the problem of manually configuring some different system parameter.

In this paper, we present an innovative mechanism for self-tuning the batching level of STOB protocols, that combines analytical modeling and Reinforcement Learning (RL) techniques. The joint use of these two techniques allows to take the best of the two worlds.

By exploiting the knowledge of a queuing-theory based mathematical model, we can drastically abate the training time required by standalone RL techniques. This has a fundamental impact not only on the time required to achieve optimal performance, but also, and perhaps more importantly, on the stability of the system at high loads. In these scenarios, the lack of initial knowledge on the system's performance would force solutions based on plain RL to explore, with equal probability, the whole range of possible batching configurations. Unfortunately, at medium-high throughput, the usage of excessively small batching configurations has the effect of overloading the sequencer, and destabilizing the whole group communication system, even for short periods of time.

In addition, by complementing the analytical model with a RL mechanism, we can rely on a computationally efficient, analytical model. The unavoidable prediction errors of the model can be corrected over time, by accumulating feedback from the operation of the system. We cast the learning problem in the context of *regret minimization* for *multi-armed bandit problems* [10]. This is a fundamental problem of the RL area, in which an agent is faced with a *bandit* (gambling machine) with multiple levers, each one associated with an unknown stochastic return. We break the optimization task into several sub-problems by discretizing the possible loads (incoming messages per second) into a number of ranges. The decision of choosing a batching level is faced separately for each load range. Specifically, given a load range, we see each batching level as a lever of a *bandit* (gambling machine). We leverage recent results on bandit problems to face the traditional exploration-exploitation trade-off, in a robust and efficient manner. Such a trade-off is determined by the necessary balance between using the best batching level determined at any given time, and the need to try other ones to assess its optimality.

II. Related Work

Packing small messages into larger ones to maximize performance is a well known optimization that is commonly employed in several domains [9], [11], [12]. TCP Nagle's algorithm [13] represents a noteworthy, widely deployed example of such a technique.

The effects of batching on the performance of TOB protocols was first studied empirically in [12] and later mathematically in [11]. To the best of our knowledge, the work in [9] is the only one to have investigated the issue of designing self-tuning mechanisms for TOB protocols. Unfortunately, the techniques proposed in this work require the explicit setting of additional parameters, e.g. the duration of timers used to wait for messages to be batched, thus failing to fully automatize the tuning of the batching mechanism. The self-tuning mechanism presented in this paper, conversely, is entirely parameter free. Further, it relies on a unique combination of analytical modelling and reinforcement learning techniques, which, to the best of our knowledge, has never been explored up to date.

Our work is also related to performance evaluation and modelling studies of TOB [14], [15] (and related agreement problems, consensus in primis [16], [17]). Instead of deriving a full performance model of the (S)TOB algorithm, the analytical model presented in this paper is restricted to capturing exclusively the effects of batching on the CPU utilization of the sequencer node, being designed to serve a different, and more specific purpose.

Machine learning techniques have already been used to predict the performance of computer systems in several contexts. These include solutions aiming at forecasting the throughput of TCP flows [18], Pub-Sub systems [19], and Atomic Broadcast protocols [20], at automatically classifying traffic based on semi-supervised learning techniques [21], at automatizing the allocation of resources in cloud-computing infrastructures [22], and at generating software aging models [23].

III. Overview of the STOB Algorithm

As already discussed in the Introduction section, STOB algorithms are probably among the most widely deployed TOB protocols [24], as they achieve the minimum bound on message latency for the TOB problem. Various variants of STOB protocols have been proposed in the literature, e.g. with fixed vs dynamic leader [12] vs with vs without uniform delivery guarantees [25]. In this work we focus on the simplest of the STOB algorithms, namely a STOB algorithm which does not guarantee message uniformity, and in which the sequencer role is statically assigned (unless in presence of group membership changes). This choice is made essentially for the sake of simplicity. Nevertheless, all of the aforementioned variants can exploit the batching optimization without additional difficulties, and the self-tuning techniques described in this paper could be adapted to be integrated in more complex variants of this family of TOB algorithms. In the remainder of the paper we shall refer, for simplicity, to the non-uniform, static STOB algorithm described in the following simply as to STOB.

In failure-free runs of the STOB algorithm, if no processes leave or join the group, the processes agree on the identity of a single process, before starting totally order broadcasting messages. Such a process, called sequencer, has the role to impose a common total order of delivery of messages to all processes in the group. If a process wants to totally order broadcast (TO-Bcast) a message, it executes a plain broadcast of the message. When a process receives a message from the network, however, it cannot immediately totally order delivery (TO-deliver) it to the application. In order to guarantee group-wide agreement on the final delivery order, in fact, it has first to wait to receive from the sequencer the corresponding *sequencing* message, and to ensure that all previously ordered messages have been delivered.

The batching level, denoted in the remainder as b, defines how many messages the sequencer waits to receive before generating a sequencing message. As already discussed, setting b to 1 ensures minimal latency at low load. At high loads, however, higher values of batching allow to amortize the cost of sequencing each message, and the sequencer to sustain much higher throughput rates.

IV. System Overview

Our self-tuning system has been developed as a layer for Appia [24], a popular Group Communication System (GCS) fully implemented in Java. Appia follows an architectural design that allows to compose layered stacks of micro-protocols according to the application needs. The self-tuning layer sits between the Sequencer TOB layer and the interface towards the application, thus achieving total transparency for the application.

The self-tuning layer traces TO broadcast/delivery events in order to collect the following two performance metrics at

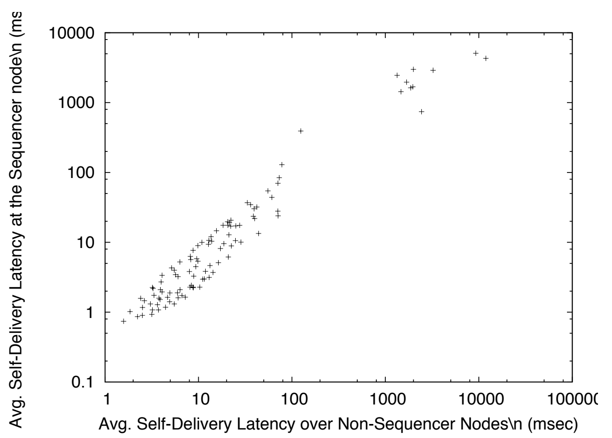

Figure 1. Self-delivery latency at the sequencer and at the non-sequencer nodes.

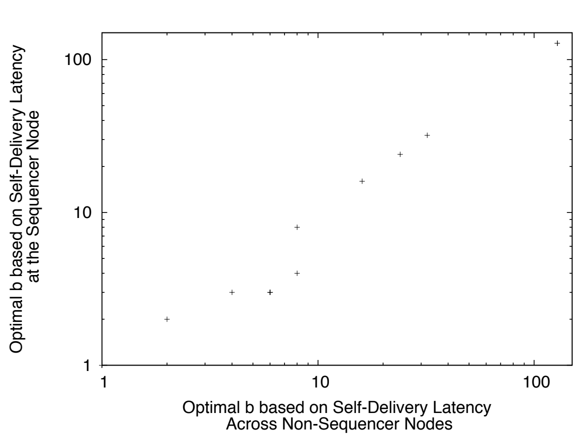

Figure 2. Optimal batching value computed on the basis of the sequencer's vs non-sequencer nodes' self-delivery latency.

the sequencer process: the message arrival rate, and the self-delivery latency, i.e., the latency experienced by the sequencer to TO-deliver messages whose TO-broadcast was activated locally.

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 inadequate for collecting measurements of the TO broadcast inter-nodes delivery latency in LAN environments, being the latter frequently around or less than a millisecond.

Further, we experimentally verified that, at least in typical LAN settings, there is a very high correlation between the self-delivery latency at the sequencer and the self-delivery latency experienced by the other nodes in the group. This claim is supported by the data in Figure 1 and Figure 2. The data in the plots was obtained using a plain (not self-tuning) STOB protocol in a cluster of 10 machines equipped with two Intel Quad-Core XEON at 2.0 GHz, 8 GB of RAM, running Linux 2.6.32-26-server and interconnected via a private Gigabit Ethernet. All the experimental data reported in the remainder of this paper have been collected using this cluster. In this experiment, we let the batching level b vary in the set $\{1,2,3,4,6,8,16,32,64,128\}$ and, for each batching level, we injected 512 bytes messages at an arrival rate ranging from 1 msg/sec up to saturating the GCS. In all of the experiments, independently of the batching level used, it was verified that the bottleneck has always resulted to be the sequencer CPU, and the available network bandwidth was always far from being saturated. In Figure 1 we report on the x axis the self-delivery (in msec) experienced on average by the non-sequencer nodes, and on the y axis the self-delivery latency experienced by the sequencer node.

To generate the plot in Figure 2 we manually computed the optimal batching value that, given the current load, minimized i) the self-delivery latency of the sequencer node, and ii) the average self-delivery latency over all the non-sequencer nodes. The correlations of the datasets in Figure 1 and Figure 2 are, respectively, 0.85 and 0.99. This experimental evidence confirms the fact that, in a LAN, by relying exclusively on local measures of self-delivery latency taken at the sequencer node, it is possible to obtain an accurate picture of the performance perceived, on average, by any node of the system.

V. The Analytical Performance Model

As discussed in the Introduction section, our self-tuning relies on an analytical model which is used to initialize a reinforcement learning system. In this section we first present the analytical model. Next we discuss how it is possible to determine its parameters' values. Finally we validate its accuracy and effectiveness in driving the self-tuning process without additional feedback from the reinforcement learning subsystem.

A. The Mathematical Model

The design of the analytical performance model used in our system has been driven by two key requirements: i) high computational efficiency, thus allowing to compute its solution also in real-time without significantly overloading the CPU of the sequencer node; ii) possibility to identify its parameters via an automatized and extremely fast procedure, thus making it easily employable in practical settings also by non-specialists.

These two requirements led us to opt for a relatively simple model that captures the impact of batching on self-delivery latency focusing on two main aspects: the additional delay introduced by forcing the sequencer to wait for the completion of a batch of messages, and the alleviation of the load pressure on the CPU of the sequencer due to the generation of a reduced number of sequencing messages. We choose to explicitly model the possible effects of contention on the network, as typical applications that use TOB services in a LAN [5], [26] generate messages large at most a few KBs. In these settings, the sequencer's CPU remains the bottleneck even at very high batching values.

We model the CPU of the sequencer as a $M/M/1$ queue [27] for which each job corresponds to a batch of messages of size b. We denote with $\lambda(b,m)$ the arrival rate of a batch of b messages given that the TO-Bcast rate is equal to m, and with $\mu(b,m)$ the average rate at which a batch of size b is sequenced given that the TO-Bcast rate is m. We can then express, by using well known queuing theory results, the STOB self-delivery latency as the response time of the queue:

$$T(b,m) = \frac{1}{\mu(b,m) - \lambda(b,m)} \tag{1}$$

subject to the stability constraint: $0 \leq \lambda(b,m) < \mu(b,m)$. We take into account the effects of batching by expressing $\lambda(b,m)$ and $\mu(b,m)$ in Equation 1 as follows:

$$\lambda(b,m) = \frac{m}{b} \tag{2}$$

where we denoted with m the average message arrival rate to the sequencer, and accounted for the fact that, when using a batching value b, the sequencer processes batches at a rate inversely proportional to b.

$$\mu(b,m) = \frac{1}{T_{1st} + \frac{(b-1)}{2m} + T_{add}(b-1)} \tag{3}$$

which expresses $\mu(b)$ as the inverse of the sum of three components:i) the CPU time, denoted as T_{1st}, required to generate a sequencing message for a single incoming message, or, equivalently, for a batch containing a single message; ii) the time required to receive the $b-1$ remaining messages of the batch; iii) the CPU time required to process the $b-1$ remaining messages of the batch. This typically corresponds to unmarshalling each incoming message and buffering it in memory, but it does not include the actual sending of the sequencing message (which is accounted for by T_{1st} in our model). Note that, in practice, the CPU time required to include an additional message into an existing batch (T_{add}) is much lower than the one associated with sending the sequencing message for a message (T_{1st}). In the following we will therefore assume that $T_{add} < T_{1st}$.

By merging Equations (1-3) we can derive the average self-delivery latency as a function of the batching level and of the average message arrival rate:

$$T(b,m) = \frac{1}{\frac{1}{T_{1st}+\frac{(b-1)}{2m}+T_{add}(b-1)} - \frac{m}{b}} \tag{4}$$

subject to the constraint:

$$\frac{1}{T_{1st} + \frac{(b-1)}{2m} + T_{add}(b-1)} - \frac{m}{b} > 0 \tag{5}$$

By letting b tend to infinity in the above constraint, we can easily determine an upper bound on the maximum throughput, m^* sustainable by the sequencer:

$$m^* = lim_{b\to\infty} \frac{b+1}{2T_{1st} + 2T_{add}(b-1)} = \frac{1}{2T_{add}} \tag{6}$$

Finally, by imposing $\frac{\partial T(b,m)}{\partial m} = 0$ and $\frac{\partial^2 T(b,m)}{\partial^2 m} > 0$ we can derive the optimal batching value as a function of the message arrival rate m, as reported in Equation 7, where we used the shorthand $\sigma = \frac{1}{T_{1st}}$. The optimal batching value is predicted via a piece-wise function. For low values of the arrival rate, as expectable, the model predicts that batching harms performance, rather than improving it. As the load increases, as long as it remains lower than the maximum sustainable throughput, the optimal batching value grows non-linearly and has a vertical asymptote at $m = m^*$.

$$b^*(m) = \begin{cases} 1, \quad \text{if } m < \frac{T_{add}\sigma^2}{2} + \frac{1}{2}\sqrt{\frac{4\sigma^2+2T_{add}^2\sigma^4}{2}} \\ \frac{2m-\sigma-2mT_{add}\sigma}{\sigma-2mT_{add}\sigma+\sqrt{\frac{2(\sigma+2m(T_{add}\sigma-1))^2}{(2\sigma T_{add}-1)^2(1+2\sigma T_{add})\sigma^2}}}, \\ \quad \text{if } \frac{T_{add}\sigma^2}{2} + \frac{1}{2}\sqrt{\frac{4\sigma^2+2T_{add}^2\sigma^4}{2}} < m < m^* \end{cases} \tag{7}$$

B. Determining the Model's Parameters

In order to solve our analytical model, it is first necessary to determine the value of the parameters T_{1st} and T_{add}. Even though the semantics of T_{1st} and T_{add} might not appear immediately manifest, it is in practice possible to estimate their values via the following, very quick, training phase. To compute T_{1st} it suffices to observes that, for $b = 1$, Equation 4 reduces to:

$$T(1,m) = \frac{1}{\frac{1}{T_{1st}} - m}, \text{ where } m < \frac{1}{T_{1st}}$$

In other words, T_{1st} is simply the inverse of the maximum throughput sustainable by the system when $b = 1$, and its value can therefore be easily computed (in an approximated manner of course) by setting the batching level to 1 and injecting traffic at an increasing rate, until saturating the GCS.

In order to derive the value of the T_{add} parameter we exploit Equation 6. To this end, we set b to the maximum value that we intend to use in our self-tuning system, which we denote with b_{max} and identify the maximum throughput sustainable by the system, denoted as $m^*_{b_{max}}$. In all our experiments we found that the throughput's gains achievable by increasing the batching value over 128 become negligible, thus we used $b_{max} = 128$. Using Equation 6, we can then set $T_{add} = \frac{1}{2m^*_{b_{max}}}$.

C. Model accuracy and efficiency

In order to determine the accuracy of the presented analytical performance model, we used the data collected for the experiment described in Section IV to manually identify the optimum batching value as a function of the message arrival rate. In Figure 3 we compare the optimal batching value predicted by the analytical model with the one manually found via exhaustive exploration of the parameters' space. The two plots report the same data, but the one on the bottom uses a log scale on the y-axis. By looking at the top plot, we observe that globally the model captures quite closely the dynamics of

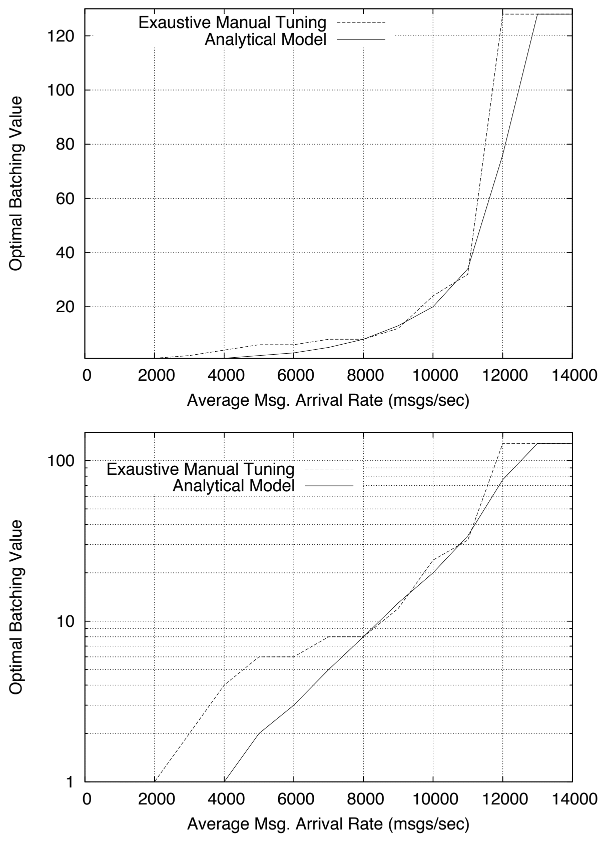

Figure 3. Validating the accuracy of the analytical model.

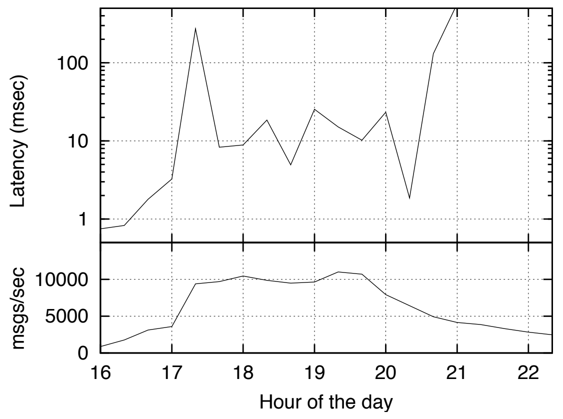

Figure 4. Self-tuning based on analytical model

the real system. The logarithmic scale plot, on the other hand, allows to better visualize that the analytical model tends to underestimate the optimal settings of the batching value at medium loads (at approx. 3000-8000 msgs./sec).The fact that the proposed model fails under some circumstances comes indeed with no surprise, as real systems are characterized by some degree of uncertainty that no model can escape, and the proposed model relies on assumptions whose correctness can only be hypothesized.

In order to assess the actual impact of the model's error on system's performance we inject traffic using the traces collected by a real system, namely FenixEDU, the Web application that is responsible for the management of the whole campus of one of the main universities in Portugal, the Instituto Superior Técnico of Lisbon. The traces we used report the number of messages in input to the cluster hosting the FenixEDU system during September 3, 2010. In this day, at 18:00, the enrolment of students for the following semester started, and the FenixEDU system was subject to a spike of load lasting several hours.

We injected traffic according to the traces from 16:00 to 22:00, and used the analytical model's prediction to dynamically adapt, with a frequency of 1Hz, the batching level. Figure 4 shows in the bottom plot the message arrival rate, and in the top plot the self-delivery latency at the sequencer node (in log scale). It is possible to note that the self-delivery latency is subject to two spikes: one during the ramp-up front of the traffic surge, and one during the ramp-down front, and both occurring as the average message arrival rate is of around 5000-6000 messages per second. As already discussed, at this load pressure, in fact, the analytical model underestimates the optimal batching level, which led the system to thrashing. Interestingly, since the ramp-up front is faster than the ramp-down, the system is able to recover from the thrashing caused by the incorrect choice of the batching level during the ramp-up. As soon as the ramp-up is completed, at high load, the analytical model predicts correctly the optimal level of batching, avoiding the GCS from crashing. Conversely, since the ramp-down lasts for a much longer time period (several hours vs around 30 minutes), the prolonged permanence in a state in which the batching level is erroneously tuned (namely configured with an excessively low value) causes the GCS to eventually collapse.

Concerning the computation efficiency of the analytical model, we implemented it in Java pre-computing every expression of Equation 7 that is independent of the message arrival rate m. We measured the time required to solve the model by passing as input parameter m the whole set of integer values in the range [1,14000], and repeated the process 100 times. On a machine equipped with an Intel Core 2 Duo at 2.53GHz running Mac OS X 10.6.6, the average time to determine the optimal batching value was in the order of 20 nanoseconds.

As a final note, it is noteworthy to highlight (even though for space constraints it will not be possible to report detailed experimental data) that we have also experimented with the usage of analytical models more complex than the above described one. These include models in which the (average) time to sequence a batch of size b at load m (essentially the denominator of Equation 3) is expressed as a generic polynomial of b. This kind of approaches could in theory allow for capturing more complex non-linear dynamics. However, they require a complex and time consuming phase of identification of the model parameters via non-linear fitting techniques [28]. Also, in our experiments, their accuracy resulted only marginally better than the one achievable by the model presented in Section V-A. These considerations have ultimately led us to opt for adopting a simpler, and consequently easier to instantiate and solve, analytical model, whose errors are dynamically compensated via the usage of a RL technique, as we discuss in the following.

VI. Combining a RL Approach

In order to compensate the errors of the analytical model, our system relies on a RL technique that dynamically updates the initial knowledge provided by the model based on the feedback gathered by observing the consequences of the self-tuning choices. To this end, we cast the problem of deciding the optimal batching level given the current system load to a classical RL problem, namely the multi-armed bandit [29]. In this problem, a gambling agent is faced with a bandit (a slot machine) with k arms, each associated with an unknown reward distribution. The gambler iteratively plays one arm per round and observes the associated reward, adapting its strategy in order to maximize the average reward. Formally, each arm i of the bandit, for $0 \leq i \leq k$, is associated with a sequence of random variables $X_{i,n}$ representing the reward of the arm i, where n is the number of times the lever has been used. The goal of the agent is to learn which arm i maximizes the criterion: $\mu_i = \sum_{n=1}^{\infty} \frac{1}{n} X_{i,n}$, that is, achieves maximum average reward. To this purpose, the learning algorithm needs to try different arms in order to estimate their average reward. On the other hand, each suboptimal choice of an arm i costs, on average, $\mu^* - \mu_i$, where μ^* is the average obtained by the optimal lever. Several algorithms have been studied that minimize the *regret* r, defined as $r = \mu^* n - \sum_{i=1}^{K} \mu_i E[T_i(n)]$, where $T_i(n)$ is the number of times arm i has been chosen. In our system we leverage on a recent result of Auer et al. [10], who introduced an algorithm, UCB, that achieves a logarithmic bound on the number of suboptimal trials not only in the limit, but also for any finite sequence. Building on the idea of *confidence bounds* this algorithms create an overestimation of the reward of each possible decision, and lowers it as more samples are drawn. Implementing the principle of *optimism in the face of uncertainty* the algorithm picks the option with the highest current bound.

In particular, assuming that rewards are limited in [0,1], each arm is associated with a value:

$$\overline{\mu_i} = \overline{x_i} + \sqrt{\frac{\log n}{n_i} min\{1/4, V_i(n_i)\}} \tag{8}$$

where $\overline{x_i}$ is the current estimated reward for arm i, n is the number of the current trial, n_i is the number of times the level i has been tried, and:

$$V_i(s) = |\frac{1}{s}\sum_{\tau=1}^{s} X_{i,\tau}^2 - \overline{x}_i^2| + \sqrt{\frac{2\log n}{s}}$$

The right-hand part of the sum in Eq. 8 is an *upper confidence bound* that decreases as more information on each option is acquired. By choosing, at any time, the option with maximum $\overline{\mu_i}$, the algorithm searches for the option with the highest reward, while minimizing the *regret* along the way.

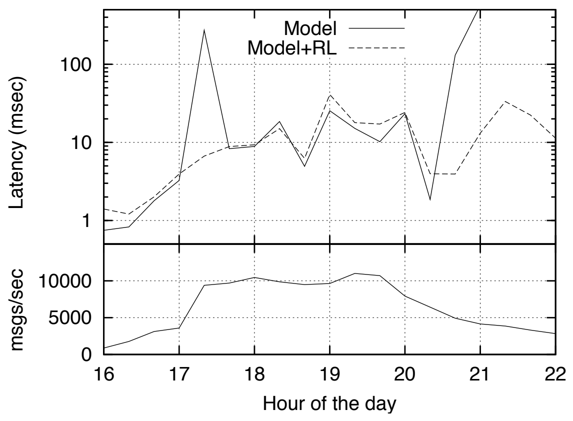

Figure 5. Self-tuning combining RL and analytical model

In order to apply this technique, we discretized the parameters space, defined by the cartesian product $b \times m$, as follows. We considered $k = 8$ different batching levels, denoted as $b_1, .., b_k$ and such that $b_i = 2^i$ for $0 \leq i < 8$. We split the message arrival rate into $l = 15$ intervals, denoted as m_l, having endpoints in $L = \{0, 10, 100\} \cup \{n * 1000 | 1 \leq n \leq 10\} \cup \{12000, 14000, 16000\}$ expressed in messages per second. Each message arrival rate interval m_l is associated with an instance of the bandit problem with k arms, where each arm is associated with a different batching level. Since in UCB rewards are bound in the [0,1] interval, given an observed self delivery latency t, we use the following function to defining its reward $R(t)$:

$$R(t) = \frac{maxLatency - min\{maxLatency, t\}}{maxLatency}$$

where $maxLatency$ is a parameter defining the maximum self-delivery observable by the sequencer that we set, consistently with what we did in Section V-B, to 100 msec (a threshold above which our GCS started thrashing severely).

As already mentioned, the original UCB technique does not rely on the availability of initial knowledge on the arms' reward distribution. In the application domain considered in this paper, however, the blind initial exploratory phase undertaken by UCB has severe consequences on system's stability. At high loads (i.e. at more 8000 messages per second in our cluster), in fact, the GCS starts thrashing after a few seconds if the batching level is not adequately tuned. This makes a plain UCB-based self-tuning technique extremely unstable, and, de facto, unusable in practice.

We tackle this issue by initializing the statistics of every arm of each UCB instance with the self-delivery latency predicted by Equation 4 of the analytical model. Figure 5 reports the performance achieved by the combined usage of the UCB-based RL technique and the analytical model, when considering the same trace-driven workload already used in Section V-C. In the plot we also report the performance achieved by the self-tuning mechanism relying solely on the analytical model. These experimental data allow us to make several interesting considerations.

At high loads, the initial knowledge provided by the analytical model allows the RL method to avoid exploring inadequately low batching values which would rapidly lead the GCS to thrashing: for message rate values larger than 10000 msgs/sec, in fact, the initial rewards for batching values lower or equal than 16 are all identically null. At medium loads, where the analytical model incurs in the biggest errors, destabilizing the system, the RL method is able to rapidly update its initial, incorrect knowledge, ensuring predictable performance and globally enhancing the robustness of the self-tuning mechanism. Finally, at extremely high and low load values, where the analytical model always guesses perfectly the optimal batching value, the exploratory behavior of RL leads to a slight deterioration of performance. This is an unavoidable cost that has to be incurred by any RL based system. However, since the UCB technique ensures that the regret increases at most logarithmically, the performance deterioration imputable to suboptimal exploratory behaviors is expected to become negligible over time.

VII. Conclusions

Analytical modelling and reinforcement learning techniques are traditionally considered as two alternative approaches to build autonomic systems. In this paper we investigated, to the best of our knowledge for the first time in literature, the possibility to combine these two approaches to build self-optimizing distributed systems.

We applied this concept, specifically, to tackle the problem of self-tuning the batching level of sequencer based total order broadcast protocols. Our experimental results show that, by appropriately combining these two different methodological approaches, it is possible to achieve the best of the two worlds. By exploiting the knowledge of a queuing-theory based mathematical model, we can drastically abate the training time required by standalone RL techniques. On the other hand, by observing the consequences of the self-tuning choices, the RL can progressively correct the unavoidable approximation errors of the analytical model.

Acknowledgments

This work has been partially supported by the project "Cloud-TM" (co-financed by the European Commission through the contract no. 257784), the FCT project ARISTOS (PTDC/EIA- EIA/102496/2008) and by FCT (INESC-ID multiannual funding) through the PIDDAC Program Funds.

References

[1] X. Defago, A. Schiper, and P. Urban, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.

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

[3] K. Ostrowski, K. Birman, and D. Dolev, "Live Distributed Objects: Enabling the Active Web," IEEE Internet Computing, November-December 2007, 2007.

[4] P. Romano, D. Rughetti, F. Quaglia, and B. Ciciani, "Apart: Low cost active replication for multi-tier data acquisition systems," in *Proc. of NCA*. IEEE Computer Society, 2008, pp. 1–8.

[5] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.

[6] N. Carvalho, P. Romano, and L. Rodrigues, "Asynchronous lease-based replication of software transactional memory," in *Proc. Middleware*. Springer Berlin / Heidelberg, 2010, pp. 376–396.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[8] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998.

[9] R. Friedman and E. Hadad, "Adaptive batching for replicated servers," in *Proc. SRDS*, 2006, pp. 311–320.

[10] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

[11] B. Carmeli, G. Gershinsky, A. Harpaz, N. Naaman, H. Nelken, J. Satran, and P. Vortman, "High throughput reliable message dissemination," in *Proc. SAC*, 2004, pp. 322–327.

[12] T. Friedman and R. V. Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," in *Proc. of HPDC*, 1997, pp. 233–.

[13] J. Nagle, "Congestion control in ip/tcp internetworks," *SIGCOMM Comput. Commun. Rev.*, vol. 14, pp. 11–17, October 1984.

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

[15] R. Ekwall and A. Schiper, "Modeling and validating the performance of atomic broadcast algorithms in high-latency networks," in *Proc. Euro-Par*, ser. Lecture Notes in Computer Science. Springer, 2007, pp. 574–586.

[16] A. Coccoli, P. Urban, A. Bondavalli, and A. Schiper, "Performance Analysis of a Consensus Algorithm Combining Stochastic Activity Networks and Measurements," in *Proc. DSN*, 2002, pp. 551–560.

[17] N. Santos and A. Schiper, "Tuning paxos for high-throughput with batching and pipelining," in *Proc. ICDCN*, 2012.

[18] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A machine learning approach to tcp throughput prediction," in *Proc. SIGMETRICS*. New York, NY, USA: ACM, 2007, pp. 97–108.

[19] L. Garces-Erice, "Admission control for distributed complex responsive systems," in *Proc. ISPDC*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 226–233.

[20] M. Couceiro, P. Romano, and L. Rodrigues, "A machine learning approach to performance prediction of total order broadcast protocols," in *Proc. SASO*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 184–193.

[21] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson, "Offline/realtime traffic classification using semi-supervised learning," *Perform. Eval.*, vol. 64, no. 9-12, pp. 1194–1213, 2007.

[22] 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*, vol. 11, no. 3, pp. 213–227, 2008.

[23] A. Andrzejak and L. Silva, "Using machine learning for non-intrusive modeling and prediction of software aging," in *Proc. NOMS*, Apr 7–11 2008.

[24] H. Miranda, A. Pinto, and L. Rodrigues, "Appia, a flexible protocol kernel supporting multiple coordinated channels," in *Proc. ICDCS'01*. IEEE, Apr. 2001, pp. 707–710.

[25] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer, 2006.

[26] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D^2STM: Dependable distributed software transactional memory," in *Proc. PRDC*. IEEE Computer Society, 2009.

[27] L. Kleinrock, *Queueing Systems. Volume 1: Theory*. Wiley-Interscience, Jan. 1975.

[28] D. M. Bates and D. G. Watts, *Nonlinear Regression Analysis and Its Applications*. Wiley, 1988.

[29] H. Robbins, "Some aspects of the sequential design of experiments," *Bulletin of the American Mathematical Society*, vol. 58, no. 5, pp. 527–535, 1952.