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

Paolo Romano  
INESC-ID Lisbon, Portugal

Matteo Leonetti  
Sapienza Rome University, 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.

1. 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. TOB represents the cornerstone of the classic active replication technique [2] and represents an incarnation of consensus [3], [4], 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 (see [1] for a comprehensive survey). Among the TOB protocols designed for being deployed in asynchronous distributed systems - the most generic, and hence most interesting, class of distributed systems - 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 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 [5]. 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, in fact, 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, even for small periods of time, has the effect of overloading the sequencer and destabilizing the whole group communication system.

On the other hand, by complementing the analytical model with a RL mechanism, we can rely on a relatively simple - and computationally efficient - analytical model whose unavoidable prediction errors 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 bandit problems [6], one of the fundamental problems in the RL area. Our solution leverages on recent results on bandit problems to face, in a robust and efficient manner, the traditional exploration-exploitation trade-off, that is 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.

2. Related Work

Packing small messages into larger ones to maximize performance is well a known optimization that is commonly employed in several domains [5], [7], [13]. TCP Nagle’s algorithm [14] 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 [7] and later mathematically in [13]. To the best of our knowledge, the work in [5] 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 [15], [16] (and related agreement problems, consensus in primis [17]). Rather than deriving a full performance model of the (S)TOB algorithm, the analytical model presented in this paper is restricted to capture exclusive the effects of batching on the CPU utilization of the sequencer node, as it designed explicitly to serve a different, and more specific purpose.

Machine learning techniques have already been used to predict the performance of computer systems. These include works aiming at forecasting the throughput of TCP flows [18] and Pub-Sub systems [19], solutions aimed at automatizing the allocation of resources in cloud-computing infrastructures [20], and at generating software aging models to be used in the context of rejuvenation frameworks [21].

3. Overview of the STOB Algorithm

Various variants of STOB protocols have been proposed in the literature, e.g. with fixed vs dynamic leader [7] vs with vs without uniform delivery guarantees [8]. 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.
4. System Overview

Our self-tuning system has been developed as a layer for Appia [9], 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 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.

Our self-tuning layer sits between the Sequencer TOB layer and the interface towards the application. This allows to achieve total transparency for the application, as well as to intercept any event generated by or delivered to the application, which include TO broadcast/delivery events, and events notifying changes of the group membership.

The self-tuning layer traces TO broadcast/delivery events in order to collect the following two performance metrics at 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 often inadequate for collecting meaningful measurements of the TO broadcast inter-nodes delivery latency, being the latter frequently around or less than a millisecond.

Further, we experimentally verified that, at least in a typical LAN settings, there is a very high correlation between the self-delivery latency at the sequencer and the self-delivery latency experience 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.

5. The Analytical Performance 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 [10], [11] 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 [12] 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:

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

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:

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

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

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 $b$.

We report this expression in Figure 3, for ease of reference, where we used the shorthand $\sigma = \frac{1}{T_{1st}}$. The optimal batching value is predicted by the model 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 tends to infinity with a vertical asymptote at $m = m^*$.

5.1. 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 estimate their values via the following, very quick, training phase. To compute $T_{1st}$ it suffices to observe that, for $b = 1$, Equation 4 reduces to:

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

The identification of the saturation point of the system can be seen as a search problem and, as such, one may rely on the wide range of classic efficient search methods and heuristics. In our system we use a simple variant of classic binary search which operates in two distinct phases. We start by generating traffic with a low rate $m_0$ (e.g. $m_0=100$ msgs/sec) to obtain the reference value of the self-delivery latency at low load, say $T_{1,low}$. Then, at regular intervals of 1 second, we double the message arrival rate (i.e. $m_{t+1} = 2m_t$) until we find that the message rate $m_t$ in which the self-delivery latency becomes at least two orders of magnitude larger than $T_{1,low}$. By aggressively doubling the message generation rate at each interval, this method allows to quickly narrow the search interval for the saturation point. We found, however, that it is prone to lead the GCS to trash by forcing it to work at excessively high loads.

In order to avoid this issue, we let the system idle for one time interval to allow it recover from the overload. Then a second phase begins, during which we inject traffic starting from $m_{t-1}$, and prudently increase it at each time interval at the finer-grained pace of 10% (i.e. $m_{t+1} = m_t \cdot 1.1$), until we find the message rate value, which we denote as $m^*_t$, in which the self-delivery latency is, again, at least two orders of magnitude larger than $T_{1,low}$. Finally, we use $\frac{1}{m^*_t}$ to estimate $T_{1st}$.

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, using a technique analogous to the one just mentioned, the maximum throughput sustainable by the system, denoted as $m_{max}^s$. 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 $T_{add} = \frac{1}{2m_{max}^s}$.

The techniques for finding the values of the model’s parameters described above have been implemented as simple application level benchmarking tool developed in JAVA which, in all of our experiments, required between 20 and 30 seconds to complete its execution.

5.2. 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 4 to manually identify the optimum batching value as a function of the message arrival rate. In Figure 4 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 right-hand side uses a log scale on the y-axis. By looking at the left-hand side plot, we observe that globally the model captures quite closely the dynamics of the real system. The right-hand side 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. 3000-6000 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

$$b^*(m) = \begin{cases} 1 & \text{if } m < \frac{T_{add}\sigma^2}{2} + \frac{1}{2} \sqrt{\frac{4\sigma^2 + 2T_{add}^2\sigma^4}{2}} \\ \frac{2m - \sigma - 2mT_{add}\sigma}{\sigma - 2mT_{add}\sigma + \sqrt{2(\sigma + 2m(T_{add}\sigma - 1))^2 - 4m^2T_{add}^2\sigma^2}} & \text{if } \frac{T_{add}\sigma^2}{2} + \frac{1}{2} \sqrt{\frac{4\sigma^2 + 2T_{add}^2\sigma^4}{2}} < m < m^* \end{cases}$$
to 22:00, and used the analytical model’s prediction to dynamically adapt, with a frequency of 1Hz, the batching level. Figure 5 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 trashing. Interestingly, since the ramp-up front is faster than the ramp-down, the system is able to recover from the trashing 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 is longer, the prolonged permanence in a state in which the batching level is erroneously tuned causes the GCS to eventually collapse.

As a final note, concerning the computation efficiency of the analytical model, we implemented it in Java pre-computing every expression of the Equation in Figure 3 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.

6. 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 [22]. 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:

$$\mu_i = \frac{1}{n} \sum_{n=1}^{\infty} X_{i,n}$$

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 $\mu^*$ is the average obtained by the optimal lever. Several algorithm have been studied that minimize the regret, defined as

$$\mu^* n - \mu_i \sum_{i=1}^{K} E[T_i(n)]$$

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. [6], 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 algorithm creates an overestimation of the reward of each possible decision, and lowers it as more samples are drawn. In particular, assuming that rewards are limited in $[0,1]$, each arm is associated a value:

$$\mu_i = \bar{x}_i + \sqrt{\frac{\log n}{n_i}} \min\{1/4, V(n_i)\}$$  \hspace{1cm} (7)$$

where $\bar{x}_i$ 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:

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

The right-hand part of the sum in Eq. 7 is an upper confidence bound that decreases as more information on each option is acquired. By choosing, at any time, the option with maximum $\bar{x}_i$, the algorithm searches for the option with the highest reward, while minimizing the regret along the way.

In order to apply this technique, we discretize the parameters space, defined by the cartesian product $b \times m$, as follows. We considered $k = 8$ different batching levels, denoted as $b_1, \ldots, b_k$ and such that $b_i = 2^i$ for $0 \leq i < 8$. We split the message arrival rate into $l = 15$ intervals, denoted as $m_i$, having endpoints in $L = \{0, 10, 100\} \cup \{n \times 1000 | 1 \leq n \leq 10\} \cup \{12000, 14000, 16000\}$ expressed in messages per second. Each message arrival rate interval $m_i$ 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 $R(t)$:

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

where maxLatency is a parameter defining the maximum self-delivery observable by the sequencer that we set, consistently with what we did in Section 5.1, to 100 msec (a threshold above which our GCS started trashing 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 trashing 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 6 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 5.2. 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 trashing: 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.

7. Conclusions

Analytical modelling methods, and reinforcement learning (and, more in general, machine 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.

References