Tuning Paxos for high-throughput with batching and pipelining

Nuno Santos and André Schiper

Ecole Polytechnique Fédérale de Lausanne (EPFL)
{nuno.santos,andre.schiper}@epfl.ch

Abstract. Paxos is probably the most popular state machine replication protocol. Two optimizations that can greatly improve its performance are batching and pipelining. Nevertheless, tuning these two optimizations to achieve high-throughput can be challenging, as their effectiveness depends on many parameters like the network latency and bandwidth, the speed of the nodes, and the properties of the application. We address this question, by first presenting an analytical model of the performance of Paxos that can be used to obtain values for tuning batching and pipelining. We then present results of experiments validating the model and investigating how these two optimizations interact in a WAN. Results for LAN are also mentioned. The results show that although batching by itself is usually sufficient to maximize the throughput in a LAN environment, in a WAN it must be complemented with pipelining.

1 Introduction

State machine replication is a technique commonly used by fault tolerant systems. This technique allows the replication of any service that can be implemented as a deterministic state machine, *i.e.*, where the state of the service is determined only by the initial state and the sequence of commands executed. Given such a service, we need a protocol ensuring that each replica of the service executes the requests received from the clients in the same order.

Paxos is probably the most popular of such protocols. It is designed for partially synchronous systems with benign faults. In Paxos, a distinguished process, the leader, receives the requests from the clients and establishes a total order, using a series of instances of an ordering protocol.

In the simplest Paxos variant, the leader orders one client request at a time. In general, this is very inefficient for two reasons. First, since ordering one request takes at least one network round-trip between the leader and the replicas, the throughput is bounded by $\frac{1}{2L}$ where L is the network latency. This dependency between throughput and latency is undesirable, as it severely limits the throughput in moderate to high latency networks. Second, if the request size is small, the fixed costs of executing an instance of the ordering protocol can become the dominant factor and quickly overload the CPU of the replicas.

In this paper, we study two well-known optimizations to the basic Paxos protocol that address these limitations: batching and pipelining. *Batching* consists

of packing several requests in a single instance of the ordering protocol. The main benefit is amortizing the fixed per-instance costs over several requests, which results in a smaller per-request overhead and, usually, in higher throughput. *Pipelining* [8] is an extension of the basic Paxos protocol where the leader initiates new instances of the ordering protocol before the previous ones have completed. This optimization is particularly effective when the network latency is high, as it allows the leader to pipeline several instances on the slow link.

Batching and pipelining are used by most replicated state machine implementations, as they usually provide performance gains of one to two orders of magnitude. Nevertheless, to achieve the highest throughput, they must be carefully tuned. With batching, the batch size controls the trade-off between throughput and response latency. With pipelining, the number of instances that can be in execution simultaneously must be limited to avoid overloading the CPU, which could significantly degrade the performance. Moreover, the optimal choice for the bounds on the batch size and number of parallel instances depends on the properties of the system and of the application, mainly on process speed, bandwidth, latency, and size of client requests.

We begin by studying analytically what are the combinations of batch size and number of parallel instances that maximize throughput for a given system and workload (Section 4). This relationship is expressed as a function $w = f(S_{batch})$, where S_{batch} is a batch size and w is a number of parallel instances (also denoted by window size). This result can be used to tune batching and pipelining, for instance, by setting the bounds on the batch and window size to one of the optimal combinations, so that given enough load the system will reach maximum throughput. To obtain the relation above, we developed an analytical model for Paxos, which predicts several performance metrics, including the throughput of the system, the CPU and network utilization of an instance, as well as its wall-clock duration. We then present the results of an experimental study comparing batching and pipelining in two settings, one representing a WAN and the other a cluster (Section 5). We show which gains are to be expected by using either of the optimizations alone or combined, the results showing that although in some situations batching by itself is enough, in many others it must be combined with parallel instances. We contrast these results with the prediction of our model, showing that the model is effective at predicting several performance metrics, including the throughput and optimal window size for a given batch size.

2 Related Work

The two optimizations to Paxos studied in this paper are particular cases of general techniques widely used in distributed systems. Batching is an example of message aggregation, which has been previously studied as a way of reducing the fixed per-packet overhead by spreading it over a large number of data or messages, see [2, 3, 5, 6]. It is also widely deployed, with TCP's Nagle algorithm [10] being a notable example. Pipelining is a general optimization technique, where

several requests are executed in parallel to improve the utilization of resources that are only partially used by each request. One of the main examples of this technique is HTTP pipelining [11]. The work in this paper looks at these two optimizations in the context of state machine replication protocols, studying how to combine them in Paxos. Most implementations of replicated state machines use batching and pipelining to improve performance, but as far as we are aware, there is no detailed study on combining these two optimizations.

In [6], the authors use simulations to study the impact of batching on several group communication protocols. The authors conclude that batching provides one to two orders of magnitude gains both on latency and throughput. A more recent work [2] proposes an adaptive batching policy also for group communication systems. In both cases the authors look only at batching. In this paper, we show that pipelining should also be considered, as in some scenarios batching by itself is not enough for optimal performance.

Batching has been studied as a general technique by [3] and [5]. In [3] the authors present a detailed analytical study, quantifying the effects of batching on reliable message transmission protocols. One of the main difficulties in batching is deciding when to stop waiting for additional data and form a batch. This problem was studied in [5], where the authors propose two adaptive batching policies.

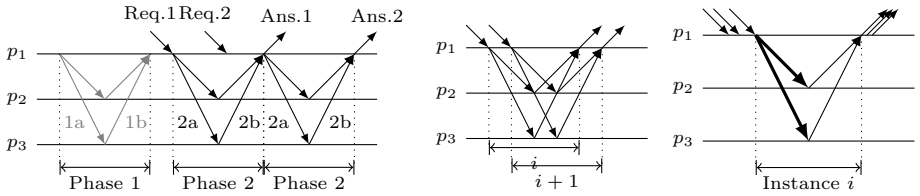
The techniques proposed in these papers can easily be adapted to improve the batching policy used in our work, which was kept simple on purpose as it was not our main focus. There are a few experimental studies showing the gains of batching in replicated state machines. One such example is [1], which describes an implementation of Paxos that uses batching to minimize the overhead of stable storage.

There has been much work on other optimizations for improving the performance of Paxos-based protocols. LCR [7] is an atomic broadcast protocol based on a ring topology and vector clocks that is optimized for high throughput. Ring Paxos [9] combines several techniques, like IP multicast, ring topology, and using a minimal quorum of acceptors, to maximize network utilization. These two papers consider only a LAN environment and, therefore, use techniques that are only available on a LAN (IP multicast) or that are effective only if network latency is low (ring-like organization). We make no such assumptions in our work, so it applies both to WAN and LAN environments. In particular, pipelining is especially effective in medium to high-latency networks, so it is important to understand its behavior.

3 Background

Paxos [8] is a protocol for state machine replication[1] which requires $n \geq 2f + 1$ replicas to tolerate f crash failures. Paxos can be seen as a *sequencer-based*

[1] Formally, Paxos is a consensus protocol and MultiPaxos its extension to state machine replication. As commonly done in the literature, we will use Paxos to denote also the state machine replication protocol.



(a) Basic MultiPaxos (b) Pipelining (c) Batching.

Fig. 1. Paxos: basic message pattern (a) and optimizations (b and c).

atomic broadcast protocol [4], where the sequencer orders requests received from the clients. In the Paxos terminology, the sequencer is called *leader*. Although Paxos is usually described in terms of the roles of proposer, acceptor and learner, this distinction is not relevant for the work in this paper so we ignore it and assume that every process is at the same time proposer, acceptor and learner.

For the purpose of the paper we describe only the relevant details of the Paxos protocol. Paxos is structured in two phases, as shown in Figure 1a. Phase 1 is executed by a newly elected leader as a preparation to order requests. Afterwards, the leader orders a series of client requests by executing several instances of Phase 2, establishing an order once a majority of Phase 2b messages are received. Since Phase 1 is executed only when a leader is elected, it has a minimal impact on performance when faults are rare. Therefore we ignore Phase 1 in our analysis, and use the term *instance* as an abbreviation for *one instance of Phase 2*.

In the simplest version of Paxos, the leader proposes one request per instance and executes one instance at a time (Figure 1a).

Pipelining: Paxos can be extended to allow the leader to execute several instances in parallel [8]. In this case, when the leader receives a new request, it can start a new instance at once, even if other instances are still undecided, as shown in Figure 1b.

Executing parallel instances *improves the utilization of resources* by pipelining the different instances. This optimization is especially effective in high-latency networks, as the leader might have to wait a long time to receive the Phase 2b messages. The main drawback is that each instance requires additional resources from the system. If too many instances are started in parallel, they may overload the system, either by maxing out the leader's CPU or by causing network congestion, resulting in a more or less severe performance degradation. For this reason, the number of parallel instances that the leader is allowed to start is usually bounded. Choosing a good bound requires some careful analysis. If set too low, the network will be underutilized. If set too high, the system might become overloaded resulting in a severe performance degradation, as shown by the experiments in Section 5. The best value depends on many factors, including the network latency, the size of the requests, the speed of the replicas, and the expected workload.

Batching: Batching is a common optimization in communication systems, which generally provides large gains in performance [6]. It can also be applied to Paxos, as illustrated by Figure 1c. Instead of proposing one request per instance, the leader packs several requests in a single instance. Once the order of a batch is established, the order of the individual requests is decided by a deterministic rule applied to the request identifiers.

The gains of batching come from spreading the fixed costs of an instance over several requests, thereby decreasing the average per-request overhead. For each instance, the system performs several tasks that take a constant time regardless of the size of the proposal, or whose time increases only residually as the size of the proposal increases. These include interrupt handling and context switching as a result of reading and writing data to the network card, allocating buffers, updating the replicated log and the internal data structures, and executing the protocol logic. In [3], the authors show that the fixed costs of sending a packet over a Ethernet network are dominant for small packet sizes, and that for larger packets the total processing time grows significantly slower than the packet size. In the case of Paxos, the fixed costs of an instance are an even larger fraction of the total costs because, in addition to processing individual messages, processes also have to execute the ordering algorithm.

Batching is fairly simple to implement in Paxos: the leader waits until having "enough" client requests and proposes them as a single proposal. The difficulty is deciding what is "enough". In general, the larger the batches, the bigger the gains in throughput. But in practice, there are several reasons to limit the size of a batch. First, the system may have physical limits on the maximum packet size (for instance, the maximum UDP packet size is 64KB). Second, larger batches take longer to build because the leader has to wait for more requests, possibly delaying the ones that are already waiting and increasing the average time to order each request. This is especially problematic with low load, as it may take a long time to form a large batch. Finally, a large batch takes longer to transfer and process, further increasing the latency. Therefore, a batching policy must strike a balance between creating large batches (to improve throughput) and deciding when to stop waiting for additional requests and send the batch (to keep latency within acceptable bounds). This problem has been studied in the general context of communication protocols by [2,3,5]. In the rest of the paper, we study it in the context of Paxos, and analyze its interaction with the pipelining optimization.

4 Analytical model of Paxos performance

We consider the Paxos variant described in Section 3 with point-to-point communication. There are other variants of Paxos that use different communication schemes, like IP multicast and chained transmission in a ring [9]. We chose the basic variant for generality and simplicity, but this analysis can be easily adapted to other variants. We further assume full duplex links and that no other applica-

Symbol	Description
n	Number of replicas
B	Bandwidth
L	One way delay (latency)
S_{req}	Size of request
k	Number of requests in a batch
w	Number of parallel instances
S_{2a}	Size of a Phase 2a message (batch)
S_{2b}	Size of ack
S_{ans}	Size of answer sent to client
ϕ_{exec}	CPU-time used to execute a request
WND	Bound on maximum number of parallel instances (Configuration parameter)
BSZ	Bound on batch size (Configuration parameter)

Table 1. Notation.

tion is competing for bandwidth or CPU time[2]. Also for simplicity, we focus on the best case, that is, we do not consider message loss or failures. We also ignore mechanisms internal to a full implementation of Paxos, like failure detection. On a finely tuned system, these mechanisms should have a minimal impact on throughput. Finally, we assume that execution within each process is sequential. The model can be extended to account for multi-core or SMP machines, but this is a non-trivial extension which, for the sake of simplicity, we do not explore here.

4.1 Quantitative analysis of Phase 2 of Paxos

Table 1 shows the parameters and the notation used in the rest of the paper. We focus on the two resources that are typically the bottleneck in a Paxos deployment, *i.e.*, the leader's CPU and its outgoing channel.

Our model takes as input the system parameters (n, B, L, and four constants defined later that model the speed of the nodes), the workload parameters (S_{req}, S_{ans} and ϕ_{exec}), and the batching level (k). From these parameters, the model characterizes how an instance utilizes the two critical resources, by determining the duration of an instance (wall-clock time), and the busy time of each resource, *i.e.*, the total time during which the resource is effectively used. With these two values, we can then determine the fraction of idle time of a resource, and predict how many additional parallel instances are needed to reach maximum utilization. The resource that reaches saturation with the lowest number of parallel instances is effectively the bottleneck: this resource determines the maximum number of parallel instances that can be executed in the system.

The model also provides estimations of the throughput and latency for a given configuration, which we use to study how different batch sizes affect the performance and the optimal number of parallel instances for each batch size.

For simplicity, we assume that all requests are of similar size. Since the bulk of the Phase 2a message is the batch being proposed, in the following we use $S_{2a} = kS_{req} + c$ to denote the batch size, where c represents the protocol headers.

[2] The presence of other applications can be modeled by adjusting the model parameters to reflect the competition for the resources.

Network busy time: The outgoing network channel of the leader is busy for the time necessary to send all the data related to an instance, which consists of $n-1$ Phase 2a messages, one to every other replica, and k answers to the clients.

Because of differences in topology, we consider the cases of a LAN and a WAN separately. On a LAN, the replicas are typically on the same network, so the effective bandwidth available between them is the bandwidth of the network. Therefore, the leader has a total bandwidth of B to use for sending messages, and we can compute the time the network is used for an instance as $\phi_{inst}^{\text{LAN}} = ((n-1)S_{2a} + kS_{ans})/B$.

On a WAN environment, however, the replicas are in different data centers, so the connection between them is composed of a fast segment inside the replica's data center (bandwidth B_L), and of another comparatively slow segment between the different data centers (bandwidth B_W). Since usually $B_W \ll B_L$, in the following analysis we consider B_W to be the effective bandwidth between the replicas, ignoring B_L, *i.e.*, we take $B = B_W$. Moreover, while in LAN a replica has a total bandwidth of B to share among all other replicas, on a typical WAN topology each replica has a total of B_W bandwidth to every other replica. The reason is that the inter-data center section of the connection between the replicas will likely be different for each pair of replicas, so that after leaving the data center, the messages from a replica will follow independent paths to each other replica. Thus, contrary to the case of a LAN, every message sent by the leader uses a separate logical channel of bandwidth B. By the same reasoning, the messages from the leader to the clients also use separate channels. Since sending the answers to the client does not delay executing additional instances, the network bottleneck are the channels between the leader and the other replicas. Therefore, we get $\phi_{inst}^{\text{WAN}} = S_{2a}/B$.

In both cases, the per request time is given by $\phi_{req}^{\text{NET}} = \phi_{inst}^{\text{NET}}/k$, where NET stands for either LAN or WAN. The maximum network throughput of instances and requests is given by $1/\phi_{inst}^{\text{NET}}$ and $1/\phi_{req}^{\text{NET}}$, respectively.

CPU time: During each instance, the leader uses the CPU to perform the following tasks: read the requests from the clients, prepare a batch containing k requests, serialize and send $n-1$ Phase 2a message, receive $n-1$ Phase 2b messages, execute the requests and send the answers to the clients (in addition to executing the protocol logic whenever it receives a message).

These tasks can be divided in two categories: interaction with clients and with other replicas. The CPU time required to interact with clients depends mainly on the size of the requests (S_{req}) and the number of requests that must be read to fill a batch (k), while the interaction with replicas depends on the number of replicas (n) and the size of the batch (S_{2a}). Since these two interactions have distinct parameters, we model them by two functions: $\phi_{cli}(x)$ and $\phi_{rep}(x)$. The function $\phi_{cli}(x)$ represents the CPU time used by the leader to receive a request from a client and send back the corresponding answer, with x being the sum of the sizes of the request and the answer. Similarly, $\phi_{req}(x)$ is the CPU time used by the leader to interact with another replica, where x is the sum of the sizes of the Phase 2a and 2b messages. Both functions are linear,

which models the well-known [3] behavior where the time to process a message consists of a constant plus a variable part, the later increasing linearly with the size of message[3]. The values of the parameters of these two functions must be determined experimentally for each system, as they depend both on the hardware used to run the replicas and on the implementation of Paxos. We show how to do so in Section 5.

Based on the previous discussion, we get the following expression for the CPU time of an instance: $\phi_{inst}^{\text{CPU}} = k\phi_{cli}(S_{req} + S_{ans}) + (n-1)\phi_{rep}(S_{2a} + S_{2b}) + k\phi_{exec}$. The first term models the cost of receiving k requests from the clients and sending back the corresponding answers, the second term represents the cost of processing $n-1$ Phase 2a and 2b messages, finally, the last term is the cost of executing the k requests. The time per request is given by $\phi_{req}^{\text{CPU}} = \phi_{inst}^{\text{CPU}}/k$. and the throughput in instances and request per seconds by $1/\phi_{inst}^{\text{CPU}}$ and $1/\phi_{req}^{\text{CPU}}$, respectively.

Wall-clock time: Estimating the wall-clock duration of an instance is more challenging than estimating the network and CPU utilization, because some operations that must complete for the instance to terminate are done in parallel. As an example, once the leader finishes sending $\lfloor n/2 \rfloor$ messages to the other replicas, the execution splits into two separate sequence of events. In one of them, the leader sends the remaining phase 2a messages. On the other, it waits for enough phase 2b messages to decide and start executing the requests. If after executing the first request in the batch, the leader did not finish sending all the Phase 2a messages, it may have to wait for the outgoing link to be free before sending the answers to the clients. Thus, the exact sequence of events that leads to completion depends on the workload and the characteristics of the system. In a fast LAN the wall-clock duration is likely to be limited by the CPU speed, while in a high-latency WAN the latency is likely the dominant factor. Similarly, if the workload consists of large requests and answers, the bandwidth is more likely to be the bottleneck than the CPU or the latency.

Therefore we model the wall-clock time by considering three different cases, each corresponding to a different bottleneck: CPU, bandwidth or latency. For each case, we compute the duration of an instance, which gives us three formulas: T_{inst}^{CPU}, T_{inst}^{band} and T_{inst}^{lat}. The instance time is the maximum of the three, *e.g.*, $T_{inst} = \max(T_{inst}^{\text{CPU}}, T_{inst}^{band}, T_{inst}^{lat})$.

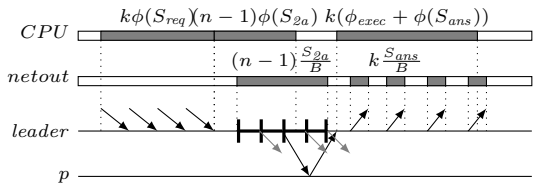
Once again, due to the differences in topology, we model the LAN and the WAN cases differently. For the LAN case, we have:

$$T_{inst}^{\text{CPU}} = \phi_{inst}^{\text{CPU}} + \lfloor n/2 \rfloor S_{2a}/2B \tag{1}$$

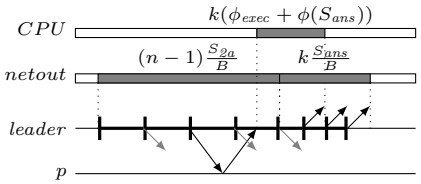
$$T_{inst}^{band} = ((n-1)S_{2a} + kS_{ans})/B \tag{2}$$

$$T_{inst}^{lat} = \lfloor n/2 \rfloor S_{2a}/B + 2L + k\phi_{exec} + kS_{ans}/B \tag{3}$$

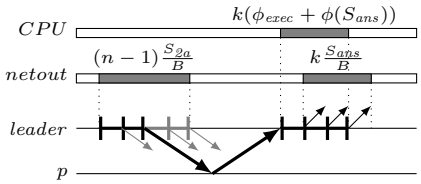
[3] We chose to use a single function to represent sending and receiving a pair of related messages, instead of one function per message type. Since the model is linear, this reduces the number of parameters that have to be estimated to half without losing any expressiveness.



(a) CPU is the bottleneck



(b) Bandwidth is the bottleneck



(c) Latency is the bottleneck

Fig. 2. Utilization of the CPU and outgoing link of the leader during an instance.

Figure 2 illustrates the three cases. Each sub-figure represents one instance. The two lines at the bottom represent the leader and the replica whose Phase 2b message triggers the decision at the leader. The two bars at the top represent the busy/idle periods of the CPU and of the outgoing link of the leader. The arrows above the leader line represent messages exchanged with the clients (their timelines are not represented) and the arrows below are messages exchanged with the other replicas.

If the CPU is the bottleneck (Equation (1) and Figure 2a), the wall-clock time of an instance is dominated by its CPU time (Formula ϕ^{CPU}_{inst} in Section 4.1/*CPU Time*). Additionally, the wall-clock time must also include the time during which the leader is sending the Phase 2a messages to other replicas, because its CPU will be partially idle as it waits for the answers. This difference between CPU and wall-clock time increases with the size of the batch (confirmed experimentally in Section 5, see Figure 4). This idle time is represented by $\lfloor n/2 \rfloor S_{2a}/2B$. If the bandwidth is the bottleneck (Equation (2) and Figure 2b), the wall-clock time of an instance is the total time needed by the leader to send all the messages of that instance through the outgoing channel, *i.e.*, $n-1$ Phase 2a messages and k answers. Finally, if the latency is the bottleneck (Equation (3) and Figure 2c), the wall-clock time of an instance is the time needed to send the first $\lfloor n/2 \rfloor$ phase 2a messages to the replicas, plus the round-trip time required to receive enough Phase 2b messages from the replicas, followed by the execution time of the requests and the time to send the answers back to the clients.

For the WAN case, the formulas are as follow:

$$T^{\text{CPU}}_{inst} = \phi^{\text{CPU}}_{inst} + S_{2a}/B \tag{4}$$

$$T^{band}_{inst} = S_{2a}/B \tag{5}$$

$$T^{lat}_{inst} = S_{2a}/B + 2L + k\phi_{exec} \tag{6}$$

The difference is that messages can be sent in parallel, because of the assumption that each pair of processes has exclusive bandwidth. Therefore, the time to send a message to the other replicas does not depend on n and sending the answers to the clients does not affect the duration of an instance (separate client-leader and leader-replica channels).

4.2 Maximizing resource utilization

If the leader's CPU and outgoing channel are not completely busy during an instance, then the leader can execute additional instances in parallel. The idle time of a resource R (CPU or outgoing link) is given by $T_{inst} - \phi^R_{inst}$ and the number of instances that a resource can sustain, w^R, is T_{inst}/ϕ^R_{inst}. From these, we can compute the maximum number of parallel instances that the system can sustain as $w = \lceil \min(w^{\text{CPU}}, w^{\text{NET}}) \rceil$.

This value can be used as a guideline to configure batching and pipelining. In theory, setting the window size to any value equal to or higher than this lower bound results in optimal throughput, but as shown by the experiments in Section 5, increasing the window size too much may result in congestion of the network or saturation of the CPU, and reduce performance. Therefore, setting the window size to w should provide the best results.

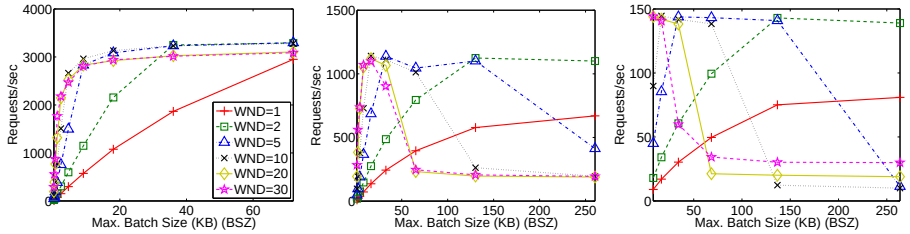
5 Experimental Study

In this section we study the batching and pipelining optimizations from an experimental perspective, and validate the analytical model. We have performed experiments both in a cluster environment and in a WAN environment emulated using Emulab [14], but in the interest of space we include below only the most representative results from the Emulab experiments. The full set of results, both for the cluster and Emulab experiments, is available in [13].

We start by presenting the experimental results, then we determine the parameters of the model that represent the process speed (parameters of $\phi_{cli}(x)$ and $\phi_{rep}(x)$), and finally compare the predictions for the throughput and optimal window size of the model with the values obtained experimentally. We performed the experiments using JPaxos [12], a full-feature implementation of Paxos in Java, which supports both batching and pipelining.

Implementing batching and pipelining in Paxos is fairly straightforward: batching has a trivial implementation and pipelining was described in the original Paxos paper [8]. To control these optimizations, *i.e.*, decide when to create a new batch and initiate a new instance, we use a simple algorithm with two parameters, *WND* and *BSZ*: *WND* is the maximum number of instances that can be executed in parallel, and *BSZ* is the maximum batch size (in bytes).

We consider a system with three replicas. In order to stress the batching and pipelining mechanisms, all the experiments were performed with the system under high load. More precisely, we used a total of 1200 clients spread over three nodes, each running in a separate thread and sending requests synchronously



(a) $S_{req} = 128$ (b) $S_{req} = 1$KB (c) $S_{req} = 8$KB

Fig. 3. Experimental results in Emulab: throughput with increasing batch size.

(*i.e.*, waiting for the previous reply before sending the next request). During the experiments, the nodes running the clients were far from being saturated, which implies that the bottleneck of the system was on the replicas.

The replicated service keeps no state. It receives requests containing an array of S_{req} bytes and answers with an 8 bytes array. We chose a simple service as this puts the most stress on the replication mechanisms. JPaxos adds a header of 16 bytes per request and 4 bytes per batch of requests. The analytical results reported below take the protocol overhead in consideration.

All communication is done over TCP. We did not use IP multicast because it is not generally available in WAN-like topologies. Initially we considered UDP, but rejected it because in our tests it did not provide any performance advantage over TCP. TCP has the advantage of providing flow and congestion control, and of having no limits on message size. The replicas open the connections at startup and keep them open until the end of the run. Each data point in the plots corresponds to a 3 minutes run, excluding the first 10%. For clarity, we omit the error bars with the confidence intervals, as they are very small.

Experimental results: The topology used for the Emulab experiments represents a typical WAN environment with geographically distributed nodes. The replicas are connected point-to-point by a 10Mbits link with 50ms latency. Since the goal is to keep the system under high load, the clients are connected directly to each replica and communicate at the speed of the physical network. The physical cluster used to run the experiments consisted of nodes of Pentium III at 850MHz with 512MB of memory, connected by a 100Mbps Ethernet.

Figure 3 shows the throughput in requests per second for increasing values of the maximum batch size. The series represent various values for the maximum window size. The results show that batching alone (*i.e.*, $WND = 1$) does not suffice to achieve maximum throughput. Although larger batches improve performance significantly, batching falls short of the maximum that is achieved with larger window sizes. The difference is greater with large request sizes (1KB and 8KB), where it achieves only half of the maximum, than for small sizes (128 bytes), where batching on its own reaches almost the maximum. The reason is that with small request sizes the leader is CPU-bound, so it cannot execute more than one parallel instance, while with larger requests the bottleneck is the

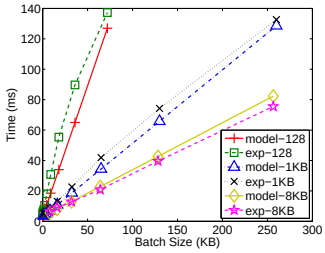


Fig. 4. Experimental versus model results for the CPU time of an instance. Fit values: $\phi_{cli}(x) = 0.28x + 0.2$, $\phi_{rep}(x) = 0.002x + 1.5$.

network latency. Increasing the window size to 2 is enough for the system to reach maximum throughput in all scenarios if the batch size is large enough (40KB with $S_{req} = 128$ and around 140KB with $S_{req} = 1$KB and $S_{req} = 8$KB). If the window size is further increased, the maximum throughput is achieved with smaller batch sizes.

The experiments also show that increasing the window size too much results in a performance collapse, with the system throughput dropping to around 10% of the maximum. This happens when the leader tries to send more data than the capacity of the network, resulting in packet loss and retransmissions. The point where it happens depends on the combination of S_{req}, WND and BSZ, which indirectly control how much data is sent by the leader; larger values increase the chance of performance collapse. With $S_{req} = 128$ there is no performance degradation, because the CPU is the bottleneck limiting the throughput. With larger request sizes, the network becomes the bottleneck and there are several cases of performance collapse. With $WND = 5$, there is a sharp drop at $BSZ =$ 256KB (Figure 3b). For larger WND, the performance collapse happens with smaller values of BSZ: with $WND = 10$ at 130KB, and at less than 64KB for larger window sizes. Similarly, as the batch size increases performance collapse occurs at smaller and smaller window sizes.

These results show that CPU and network may react to saturation very differently. In this particular system, the CPU deals gracefully with saturation, showing almost no degradation, while the network saturation results in a performance collapse. The behavior may differ significantly in other implementations, because the behavior of the CPU or network when under load (graceful degradation or performance collapse) depends on the implementation of the different layers of the system, mainly application and replication framework (threading model, flow-control) but also operating system and network stack.

A note on the cluster results: In the experiments performed in a cluster environment [13], batching by itself is enough to achieve the maximum throughput, with pipelining having minimal impact on the results. The reason for this difference is that the latency in a cluster is very low so the leader does not have time to start new instances while waiting for the results of previous instances.

S_{2a}	Model (predictions) w^{CPU}	w^{NET}	Max Thrp	Experiments w	Max Thrp
128	**30.88**	833.48	308	30-35	≈ 330
256	**28.77**	422.94	574	25-30	≈ 550
1KB	**20.45**	107.58	1620	20-25	≈ 1800
16KB	**3.38**	7.68	3765	2-5	≈ 3100
32KB	**1.47**	3.12	4032	1-2	≈ 3300

(a) $S_{req} = 128$

S_{2a}	Model (predictions) w^{CPU}	w^{NET}	Max Thrp	Experiments w	Max Thrp
1KB	**28.89**	119.01	286	30-40	≈ 310
2KB	**25.54**	60.12	502	30-40	≈ 600
8KB	**15.42**	15.8	1155	15-20	≈ 1030
128KB	3.16	**1.93**	1184	1-2	≈ 1120
256KB	2.68	**1.6**	1184	1-2	≈ 1100

(b) $S_{req} = 1KB$

S_{2a}	Model (predictions) w^{CPU}	w^{NET}	Max Thrp	Experiments w	Max Thrp
8KB	19.47	**16**	150	15-20	≈ 144
16KB	14.24	**8.5**	150	5-10	≈ 144
64KB	6.72	**2.88**	150	2-5	≈ 144
128KB	4.84	**1.94**	150	1-2	≈ 144
256KB	3.8	**1.47**	150	1-2	≈ 144

(c) $S_{req} = 8KB$

Table 2. Emulab: comparison of analytical and experimental results. Prediction of optimal w is in bold.

Setting model parameters: To estimate the parameters ϕ_{cli} and ϕ_{rep} we used the Java Management interfaces (`ThreadMXBean`) to measure the total CPU time used by the leader process during a run. Dividing this value by the total number of instances executed during the run gives the average per-instance CPU time. To prevent the JVM warm-up period from skewing the results, we ignore the first 30 seconds of a run (for a total duration of 3 minutes). We repeat the measurements for several request and batch sizes, and then adjust the parameters of the model manually until the model's estimation for the CPU time (ϕ^{CPU}_{inst}) fits the training data. Figure 4 shows the training data together with the results of the model, for the final fit of $\phi_{cli}(x) = 0.28x + 0.2$ and $\phi_{rep}(x) = 0.002x + 1.5$. The figure shows that the CPU time measured experimentally increases roughly linearly with the size of the batch, which validates our choice of a linear model.

Comparison of analytical and experimental results: Table 2 shows the results of the model for the optimal window size of the CPU and network for several batch sizes, and compares them with the experimental results.

The analytical results show that the bottleneck with 128 bytes requests is the CPU (w^{CPU} is smaller than w^{NET}) while for 8KB requests it is the network. With 1KB requests, the behavior is mixed, with the CPU being the bottleneck with small batch sizes and the network with larger batch sizes. These results quantify the common sense knowledge that smaller requests and batches put a greater load on the CPU in comparison to the network. Moreover, as the request size or batch size increase, the optimal window size decreases, because if each instance contains more data, the network will be idle for less time.

The experimental results in Table 2 are obtained by determining for each batch size the maximum throughput and the smallest w where this maximum is first achieved.

In all cases the prediction for w is inside the range where the experiments first achieve maximum throughput, showing that the model provides a good

approximation. Concerning the throughput, the model is accurate with S_{req} = 8KB across all batch sizes. With S_{req} = 128, it is accurate for the smallest batches but overestimates the throughput for the larger batches. The reason is that the network can be modeled more accurately than the CPU, as it tends to behave in a more deterministic way[4]. The CPU exhibits a more non-linear behavior, especially when under high load as is the case when the number of requests in a single batch increase to more than hundreds.

6 Discussion

In this paper we have studied two important optimizations to Paxos, batching and pipelining. The analytical model presented in the paper is effective at predicting the combinations of batch size and number of parallel instances that result in optimal throughput in a given system.

The experiments show clearly that batching by itself provides the largest gains both in high and low latency networks. Since it is fairly simple to implement, it should be one of the first optimizations considered in Paxos and, more generally, in any implementation of a replicated state machine.

Pipelining is useful only in some systems, as its potential for throughput gains depends on the ratio between the speed of the nodes and the network latency: the more time the leader spends idle waiting for messages from other replicas, the greater the potential for gains of executing instances in parallel. Thus, in general, it will provide minimal performance gains over batching alone in low latency networks, but it provides substantial gains when latency is high.

While batching decreases the CPU overhead of the replication stack, executing parallel instances has the opposite effect because of the overhead associated with switching between many small tasks. This reduces the CPU time available for the service running on top of the replication task and, in the worst case, can lead to a performance collapse if too many instances are started simultaneously (see Emulab experiments).

The model can be used in the following way to tune batching and pipelining: (i) choose the largest batch size that for a given workload satisfies the response time requirements, then (ii) use the model to determine the corresponding number of parallel instances that maximize throughput. The rationale for this heuristic is the following. As batching provides larger gains than pipelining, the batch size should be the first parameter to be maximized. However, there is a limit on how much it can be increased, because large batches take longer to fill up with requests leading to an higher response time. Given the expected request rate and the desired response time, we can easily compute the largest batch size that satisfies the response time. The model then provides the corresponding window size that maximizes throughput. As an example, consider the Emulab environment. If the average request size is 1KB and we have determined that the batch size should be 8KB, then the window size should be set to 16 (Table 2b).

[4] This is true only until reaching a level of saturation where packets are dropped, after which it becomes difficult to model

The paper has focused on throughput rather than latency because as long as latency is kept within an acceptable range, optimizing throughput provides greater gains in overall performance. A system tuned for high-throughput will have higher capacity, therefore being able to serve a higher number of clients with an acceptable latency, whereas a system tuned for latency will usually reach congestion with fewer clients, at which point its performance risks collapsing to values well below the optimal.

Acknowledgment

The authors would like to thank Paweł T. Wojciechowski, Jan Kończak and Tomasz Żurkowski for their work on JPaxos.

References

1. Amir, Y., Kirsch, J.: Paxos for system builders. Tech. Rep. CNDS-2008-2, Johns Hopkins University (2008)
2. Bartoli, A., Calabrese, C., Prica, M., Di Muro, E., Montresor, A.: Adaptive message packing for group communication systems. In: OTM 2003 Workshops. LNCS, Springer (2003)
3. Carmeli, B., Gershinsky, G., Harpaz, A., Naaman, N., Nelken, H., Satran, J., Vortman, P.: High throughput reliable message dissemination. In: Proceedings of the 2004 ACM Symposium on Applied Computing. NY, USA (2004)
4. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. 36 (Dec 2004)
5. Friedman, R., Hadad, E.: Adaptive batching for replicated servers. In: Symposium on Reliable Distributed Systems, SRDS'06 (Oct 2006)
6. Friedman, R., Renesse, R.: Packing messages as a tool for boosting the performance of total ordering protocols. Tech. Rep. TR95-1527, Department of Computer Science, Cornell University (1995)
7. Guerraoui, R., Levy, R.R., Pochon, B., Quéma, V.: Throughput optimal total order broadcast for cluster environments. ACM Trans. Comput. Syst. 28(2) (2010)
8. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems 16(2) (May 1998)
9. Marandi, P., Primi, M., Schiper, N., Pedone, F.: Ring Paxos: A high-throughput atomic broadcast protocol. In: Dependable Systems and Networks (DSN'10) (Jun 2010)
10. Nagle, J.: Congestion control in IP/TCP internetworks. Tech. Rep. RFC 896, IETF (Jan 1984)
11. Padmanabhan, V.N., Mogul, J.C.: Improving HTTP latency. Computer Networks and ISDN Systems 28(1-2) (1995)
12. Santos, N., Kończak, J., Żurkowski, T., Wojciechowski, P., Schiper, A.: JPaxos - State machine replication in Java. Tech. Rep. 167765, EPFL (Jul 2011)
13. Santos, N., Schiper, A.: Tuning Paxos for high-throughput with batching and pipelining. Tech. Rep. 165372, EPFL (Jul 2011)
14. White, B., et al.: An integrated experimental environment for distributed systems and networks. In: Proc. of the Fifth Symposium on Operating Systems Design and Implementation. Boston, MA (Dec 2002)