Theoretical Computer Science 496 (2013) 170–183

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Optimizing Paxos with batching and pipelining

Nuno Santos [*], André Schiper

Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland

A R T I C L E I N F O

Keywords:
Fault-tolerance
Paxos
Batching
Pipelining
Analytical study
Experimental evaluation

A B S T R A C T

Paxos is probably the most popular state machine replication protocol. Two optimizations that can greatly improve its performance are batching and pipelining. However, 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 presenting an analytical model of the performance of Paxos, and showing how it can be used to determine configuration values for the batching and pipelining optimizations that result in high-throughput. We then validate the model, by presenting the results of experiments where we investigate the interaction of these two optimizations both in LAN and WAN environments, and comparing these results with the prediction from the model. The results confirm the accuracy of the model, with the predicted values being usually very close to the ones that provide the highest performance in the experiments. Furthermore, both the model and the experiments give useful insights into the relative effectiveness of batching and pipelining. They show that although batching by itself provides the largest gains in all scenarios, in most cases combining it with pipelining provides a very significant additional increase in throughput, with this being true not only on high-latency WAN but also in many LAN scenarios.

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, establishes a total order among the client requests using a series of instances of an ordering protocol.

In the simplest Paxos variant, henceforth called baseline Paxos, 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, especially 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 baseline Paxos 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

[*] Corresponding author. Tel.: +41 21 69 35354.
E-mail addresses: nuno.santos@epfl.ch (N. Santos), andre.schiper@epfl.ch (A. Schiper).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.10.002

amortizing the fixed per-instance costs over several requests, which results in a smaller per-request overhead and, usually, in higher throughput. *Pipelining* [1] is an extension of baseline Paxos 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 compared to the speed of the nodes, 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. Although the algorithms behind these optimizations are straightforward, tuning their parameters in order to achieve the highest throughput is challenging. With batching, the batch size controls the trade-off between throughput and response latency. With pipelining, the number of instances executed in parallel must be limited to avoid overloading the system resources, either the CPU of the replicas or the network, 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. 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 reaches 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. 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 compare 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.

The rest of the paper is organized as follows. Section 2 presents the related work, and Section 3 provides the background for our work, describing in more detail the batching and pipelining optimizations in Paxos, Section 4 presents our analytical model of Paxos and shows how it can be used to tune these two optimizations, Section 5 presents the experimental evaluation of these two optimizations on LAN and WAN environments, and Section 6 discusses our results and concludes the paper.

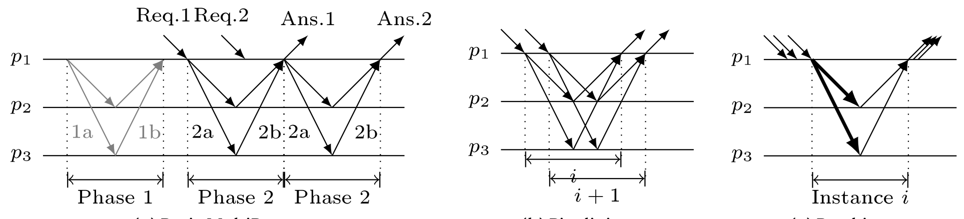
2. Related work

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 [2–5]. It is also widely deployed, with TCP's Nagle algorithm [6] 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 [7]. Most implementations of replicated state machines use batching and pipelining, as they are easy to implement and provide large gains in performance. Although their implementation is easy, tuning them is challenging. As mentioned below, there has been some work on understanding and tuning the batching optimization. But as far as we are aware, there is no detailed study on tuning these two optimizations when they are deployed in combination in the specific context of state machine replication.

In [2], 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 [3] 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 [4,5]. In [4] 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 [8], 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 [9] is an atomic broadcast protocol based on a ring topology and vector clocks that is optimized for high throughput. Ring-Paxos [10] 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.



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

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

Protocols like Mencius [11] take a different approach at improving the throughput of baseline Paxos, by allowing every replica to coordinate different instances of the ordering phase concurrently. These so-called multi-coordinated protocols can be seen as an logical extension of the pipelining optimization discussed in this paper, where instead of overlapping instances in the network links of a single replica (the leader), they overlap instances across the links of all replicas. The two approaches are orthogonal and, in principle, can be combined as individual replicas in a multi-coordinated protocol system could achieve higher throughput by pipelining the instances they coordinate.

3. Background

Paxos [1] 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* atomic broadcast protocol [12], where the sequencer orders requests received from the clients. In the Paxos terminology, the sequencer is called the *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 Fig. 1(a). 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: it sends a Phase 2a message to all replicas with the request and a proposal for its order (a sequence number), to which the replicas answer with a Phase 2b messages accepting the order (in the common case). The final order is established once the leader receives a majority of Phase 2b messages. 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 baseline Paxos, the leader proposes one request per instance and executes one instance at a time (Fig. 1(a)).

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

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 [2]. It can also be applied to Paxos, as illustrated by Fig. 1(c). 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 [4], 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

[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.

Table 1
Notation.

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 phase 2b message
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)

even larger fraction of the total costs because, in addition to processing individual messages, processes also have to execute the ordering algorithm. Additionally, when stable storage is used, batching dramatically decreases its overhead, because a single stable storage access is enough to log the state of all requests in a batch.

Batching is fairly simple to implement in Paxos: the leader waits until having "enough" client requests and proposes them as a single value. 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, UDP packets can be at most 64 KB, and in most cases the underlying network infrastructure imposes an even smaller limit. 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 [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

4.1. Assumptions

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 [10]. We chose the basic variant for generality and simplicity, but our analysis can be adapted to other variants. We further assume full duplex links and that no other application 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.

4.2. 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 outgoing network channel.

Our model takes as input the system parameters (n, B, L, and four constants defined below 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 time during which each of the resources is effectively used (the resource busy time). With these three values, we can then determine the percentage of idle time of each resource, and predict how many additional parallel instances are needed to reach its maximum utilization. The resource reaching saturation with the lowest number of parallel instances is the bottleneck of the system, and determines the maximum number of parallel instances that can be executed.

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.

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

For simplicity, we assume that all requests are of similar size, although our techniques could be extended to consider the case of different requests sizes. Note that with this assumption, we have $S_{2a} = kS_{req} + c$, where c is the size of the protocol headers. For readability, we use simply S_{2a} in the formulas below.

4.2.1. Network busy time

The outgoing 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.

In a LAN scenario, the replicas are typically in 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 available for sending the Phase 2a messages, which leads to the following formula for the per-instance network busy time:

$$\phi^{\text{LAN}}_{inst} = ((n-1)S_{2a} + kS_{ans})/B.$$

In a WAN scenario, however, the replicas are in different data centers, so the connection between them is composed of a fast segment inside each replica 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

$$\phi^{\text{WAN}}_{inst} = S_{2a}/B.$$

From the above formulas, we can compute the maximum network throughput both in terms of instances and requests. The *throughput in instances* is given by $1/\phi^{\text{NET}}_{inst}$, where NET stands for either LAN or WAN. For the throughput in requests, we first derive the network time used for each request, which is $\phi^{\text{NET}}_{req} = \phi^{\text{NET}}_{inst}/k$. With this new formula, the *throughput in requests* is $1/\phi^{\text{NET}}_{req}$.

4.2.2. 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 and behaviors, 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 [4] 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 5.2.2.

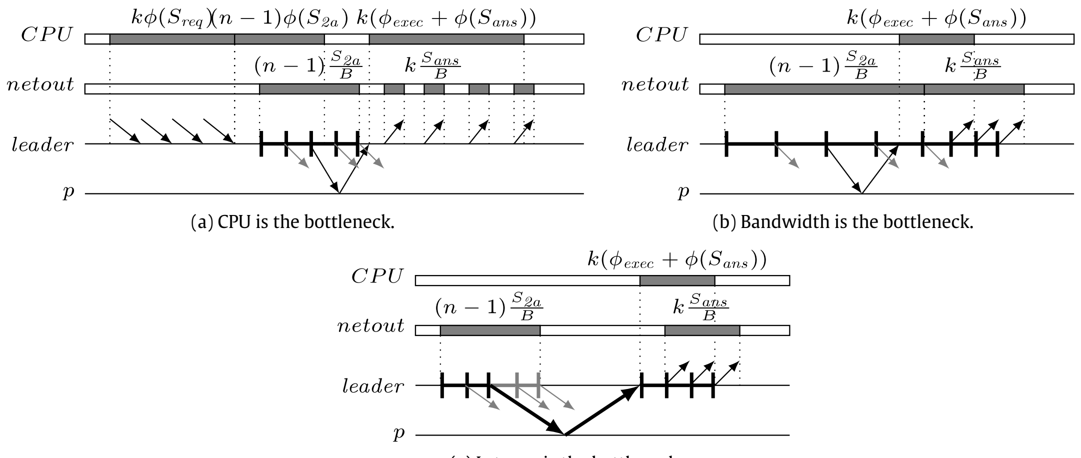
Based on the previous discussion, we get the following expression for the CPU time of an instance:

$$\phi^{\text{CPU}}_{inst} = k\phi_{cli}(S_{req} + S_{ans}) + (n-1)\phi_{rep}(S_{2a} + S_{2b}) + k\phi_{exec}. \tag{1}$$

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 and, finally, the last term is the cost of executing the k requests.

From the previous formula, we can compute the time per request as $\phi^{\text{CPU}}_{req} = \phi^{\text{CPU}}_{inst}/k$, and the throughput in instances and requests as $1/\phi^{\text{CPU}}_{inst}$ and $1/\phi^{\text{CPU}}_{req}$, respectively.

[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.

4.2.3. 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 and 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:

$$T_{inst} = \max(T_{inst}^{\mathrm{CPU}}, T_{inst}^{band}, T_{inst}^{lat}). \tag{2}$$

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

$$T_{inst}^{\mathrm{CPU}} = \phi_{inst}^{\mathrm{CPU}} + \lfloor n/2 \rfloor S_{2a}/2B \tag{3}$$
$$T_{inst}^{band} = ((n-1)S_{2a} + kS_{ans})/B \tag{4}$$
$$T_{inst}^{lat} = \lfloor n/2 \rfloor S_{2a}/B + 2L + k\phi_{exec} + kS_{ans}/B. \tag{5}$$

Fig. 2 illustrates these 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 time-lines are not represented) and the arrows below are messages exchanged with the other replicas.

If the CPU is the bottleneck (Eq. (3) and Fig. 2(a)), the wall-clock time of an instance is dominated by its CPU time ($\phi_{inst}^{\mathrm{CPU}}$, Formula (1)). However, we must also account for the time during which the leader is sending the Phase 2a messages to other replicas, because its CPU will be partially idle during this time, while waiting for the Phase 2b messages from the replicas. This difference between CPU and wall-clock time increases with the size of the batch (confirmed experimentally in Section 5, see Fig. 10). This idle time is represented by $\lfloor n/2 \rfloor S_{2a}/2B$.

If the bandwidth is the bottleneck (Eq. (4) and Fig. 2(b)), 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 (Eq. (5) and Fig. 2(c), 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:

$$T_{inst}^{\text{CPU}} = \phi_{inst}^{\text{CPU}} + S_{2a}/B \tag{6}$$

$$T_{inst}^{band} = S_{2a}/B \tag{7}$$

$$T_{inst}^{lat} = S_{2a}/B + 2L + k\phi_{exec}. \tag{8}$$

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).

4.3. Maximizing resource utilization with parallel instances

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 $T_{inst} - \phi_{inst}^{R}$ and the number of instances that a resource can sustain, w^R, is T_{inst}/ϕ_{inst}^{R}, for $R \in \{\text{CPU}, \text{NET}\}$. From these, we can compute the maximum number of parallel instances that the system can sustain as:

$$w = \lceil \min(w^{\text{CPU}}, w^{\text{NET}}) \rceil. \tag{9}$$

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 5.2, 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 (Section 5.1) and in a WAN environment emulated using Emulab [13] (Section 5.2).

For each scenario, we start by presenting the experimental results, then we determine the parameters of the model that characterize 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 [14], a full-feature implementation of Paxos in Java, which supports both batching and pipelining.

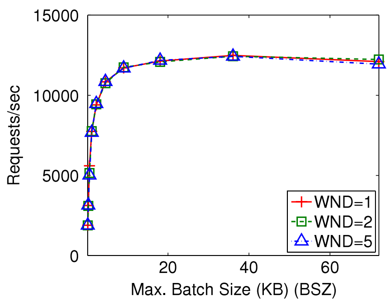
Implementing batching and pipelining in Paxos is fairly straightforward: batching has a trivial implementation and pipelining was described in the original Paxos paper [1]. To control these optimizations, *i.e.*, decide when to create a new batch and initiate a new instance, we use a simple algorithm with three parameters, *WND*, *BSZ* and Δ_B. The parameter *WND* is the maximum number of instances that can be executed in parallel, *BSZ* is the maximum batch size (in bytes), and Δ_B is the batch timeout. The timeout Δ_B is reset whenever the leader opens a new batch, which happens when it receives the first request that is assigned to the batch. The leader then waits until either it has enough requests to fill the batch or the timeout Δ_B expires. It then proposes the batch by starting a new instance as soon as the number of active instances is under *WND*. In the experiments we vary *BSZ* and *WND* while keeping Δ_B set to 50 ms. This timeout has no impact on the results, because as explained below, all experiments were performed with the system under high load, so that in the common case the leader is able to fill a batch before the timeout expires.

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 in a closed loop (*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.

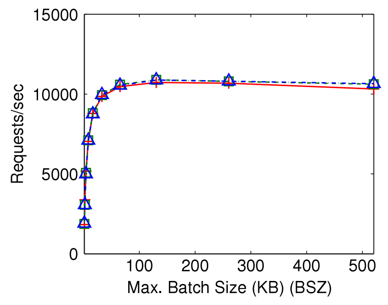
The replicated service keeps no state. It receives requests containing an array of S_{req} bytes and returns an 8 byte array. We chose a simple service as this puts the most stress on the replication mechanisms. JPaxos adds a header of 16 bytes per request and 4 bytes per batch of requests. The analytical 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 min run, excluding the first 10%. For clarity, we omit the error bars with the confidence intervals, as they are very small.

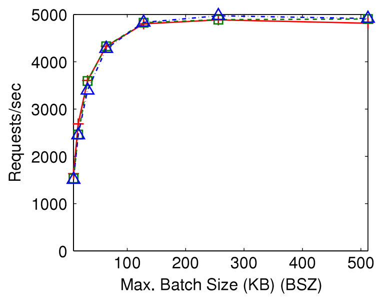
We report our results using the following metrics. The *throughput of instances* is the number of Phase 2 instances executed per second, and the *throughput of requests* is the number of requests ordered per second. These two metrics correspond to the throughput formulas of the analytical model given at the end of Sections 4.2.1 and 4.2.2. The *latency per instance* is the time elapsed at the leader from proposal to decision of an instance, *i.e.*, from sending the Phase 2a message to receiving a majority of Phase 2b messages. It corresponds to T_{inst} (Formula (2)) in the analytical model. The *client latency* is the time the



(a) $S_{req} = 128$.

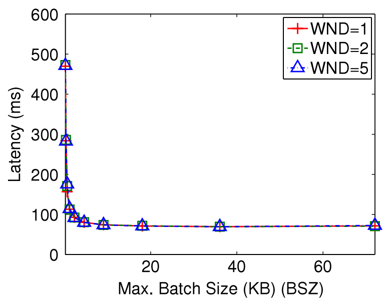


(b) $S_{req} = 1$ KB.

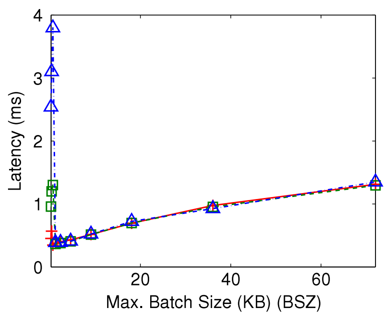


(c) $S_{req} = 8$ KB.

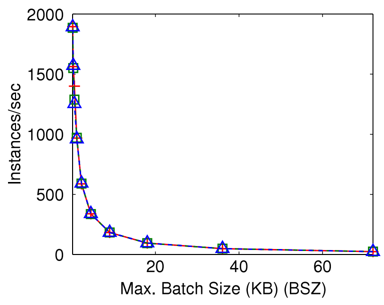
Fig. 3. Cluster. Experimental results: throughput as a function of batch size.



(a) Client latency.



(b) Instance latency.



(c) Instances/sec.

Fig. 4. Cluster, experimental results with $S_{req} = 128$.

client waits for the reply to a request, which includes the transmission time from the client to the leader, the queuing time of the request at the leader, the time to order the request, and the time to send the answer back to the client.

5.1. *Cluster*

The following experiments were run on a cluster of Pentium 4 at 3 GHz with 1 GB memory connected by a Gigabit Ethernet. The effective bandwidth of a TCP stream between two nodes measured by `netperf` is 940 Mbit/s.

5.1.1. *Experimental results*

Fig. 3 shows the request throughput as a function of batch size, for request sizes of 128 bytes, 1 KB and 8 KB, and for maximum window sizes of 1, 2 and 5.

Batching provides a major improvement in performance in all cases, ranging from an almost ten times improvement with 128 bytes requests to a little over four times with 8 KB requests. The batch size where the system reaches optimal throughput varies depending on the request size: around 10 KB, 64 KB and 128 KB for request sizes of 128 bytes, 1 KB and 8 KB, respectively. On the other hand, increasing *WND* does not improve performance. In fact, except for the smallest batch sizes, the average number of consensus instances open at any time was always one (not shown), which shows that the leader is not able to start additional consensus instances. The cause is the relatively slow CPU by comparison to the network. During each run the average CPU utilization of the leader's CPU is above 90%, suggesting that the leader is CPU-bound and, therefore, is not able to execute additional instances in the time it waits for the answers for the previous instances.

Note that the performance does not drop if *BSZ* or *WND* are increased past their optimal values. This is a desirable behavior, because the system will perform optimally with a wide range of configuration parameters, making it easier to tune. As Section 5.2 shows, this is not always the case.

Fig. 4 shows the effects of batching in several other metrics with request size of 128 bytes. The results show that even small levels of batching have a dramatic effect on performance. The client latency (Fig. 4(a)) goes from almost 500 ms without batching to under 100 ms just by increasing *BSZ* to 2 KB. Further increases in batch size provide only small additional improvements in response time, as the bottleneck shifts from the CPU required to process each instance of the ordering protocol to the cost of handling each individual client request.

The instance latency (Fig. 4(b)) increases linearly with the size of the batch, which shows clearly that its time is composed of a fixed constant time ($\approx$0.35 ms in this case) plus a variable time that depends on the amount of data that has to be transmitted. The only exception is for the smallest batches sizes with window sizes of 2 and 5, where the instance time is up to eight times higher that what would be expected from that rule above. As the batch sizes are too small to handle the

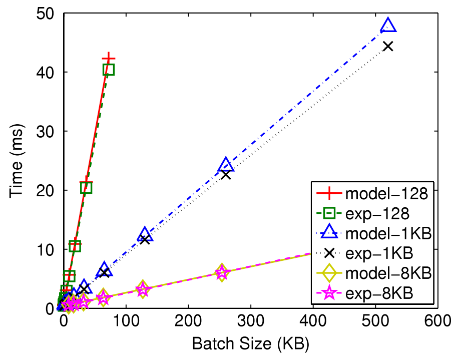


Fig. 5. Experimental versus model results for the CPU time of an instance in the cluster. Fit values: $\phi_{cli}(x) = 0.005x + 0.08$, $\phi_{rep}(x) = 0.0035x + 0.22$.

Table 2
Analytical results for the cluster scenario, with request size of 128 bytes ($S_{req} = 128$). Results for different batch sizes (S_{2a}, see Table 1). Times in ms.

S_{2a}	ϕ_{req}^{CPU}	ϕ_{inst}^{CPU}	ϕ_{inst}^{NET}	T_{inst}	w^{CPU}	w^{NET}
128	0.52	0.52	0.00	0.52	1.00	211.73
256	0.30	0.60	0.00	0.60	1.00	124.18
512	0.19	0.77	0.01	0.77	1.00	79.52
1 KB	0.14	1.09	0.02	1.10	1.00	56.97
2 KB	0.11	1.75	0.04	1.76	1.01	45.63
4 KB	0.10	3.06	0.08	3.07	1.01	39.95
8 KB	0.09	5.67	0.15	5.71	1.01	37.11
16 KB	0.09	10.90	0.31	10.98	1.01	35.68
32 KB	0.08	21.36	0.62	21.51	1.01	34.97
64 KB	0.08	42.28	1.23	42.58	1.01	34.62

incoming client load, the leader attempts to compensate by executing parallel instances. But with a slow CPU (recall that the nodes are single-core, single-CPU Pentium 4 at 3 GHz), the additional instances cause instability resulting in a spike in the consensus execution time.

Finally, Fig. 4(c) shows that, as expected, by using larger batches the leader can execute fewer instances to order higher number of requests. Note that without batching, the leader executes 2000 instances per second, corresponding to 2000 requests (Fig. 3(a)). As the batch size increases, the number of instances drops to under 200, but the throughput in requests increases to 12000, because the number of requests per batch increases faster than the reduction in the number of instances.

5.1.2. *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 s of a run (for a total duration of 3 min). 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_{inst}^{CPU}) fits the training data. Fig. 5 shows the training data together with the results of the model, for the final fit of $\phi_{cli}(x) = 0.005x + 0.08$ and $\phi_{rep}(x) = 0.0035x + 0.22$. 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.

5.1.3. *Comparison of analytical and experimental results*

All the analysis below is done with $\phi_{exec} = 0$, since the request execution time of the service used in the experiments is negligible (recall that the service simply answers with a 8 byte array). Table 2 shows detailed results for the case $S_{req} = 128$, while Table 3 shows a summary of the analytical results for all request sizes and compares them with the experimental results.

With $S_{req} = 128$ (Table 2), the CPU time used by an instance (column ϕ_{inst}^{CPU}) is an order of magnitude larger than the network busy time (column ϕ_{inst}^{NET}). As a result, the wall-clock time of an instance (column T_{inst}) is dominated by the CPU time. Although the network could sustain a large number of parallel instances (column w^{NET}), the CPU cannot sustain more than one (column w^{CPU}) and therefore the system as a whole has no capacity to execute additional instances. The situation is similar for larger requests sizes (columns w^{NET} and w^{CPU} in Table 3(b) and (c)), although the CPU becomes less of a bottleneck as the size of the requests increases. A similar pattern occurs as the batch size increases, with the load shifting from the CPU to the network. But even with the largest messages tested, *i.e.*, $S_{req} = 8$ KB and $S_{2a} = 512$ KB, the CPU is still the bottleneck,

Table 3
Cluster: comparison of analytical and experimental results for different batch sizes (S_{2a}). Prediction of optimal w [= $\min(w^{\text{CPU}}, w^{\text{NET}})$] is in bold. The column w shows the range that was determined experimentally to contain the smallest value of *WND* that produces the maximum throughput.

S_{2a}	Model			Experiments	
	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
$S_{req} = 128$					
128	**1**	211.73	1916	1	≈1895
256	**1**	124.18	3313	1	≈3126
1 KB	**1**	56.97	7313	1	≈7745
32 KB	**1**	34.97	11983	1	≈12488
64 KB	**1**	34.62	12108	1	≈12100
$S_{req} = 1$ KB					
1 KB	**1.01**	31.54	1878	1	≈1850
2 KB	**1.01**	18.64	3202	1	≈3380
8 KB	**1.03**	8.93	6791	1	≈7050
256 KB	**1.04**	5.79	10644	1	≈10680
512 KB	**1.05**	5.74	10742	1	≈10400
$S_{req} = 8$ KB					
8 KB	**1.05**	4.92	1625	1	≈1634
16 KB	**1.08**	3.25	2530	1	≈2687
64 KB	**1.14**	2	4344	1	≈4328
256 KB	**1.17**	1.68	5293	1–2	≈4900
512 KB	**1.17**	1.63	5493	1–2	≈4900

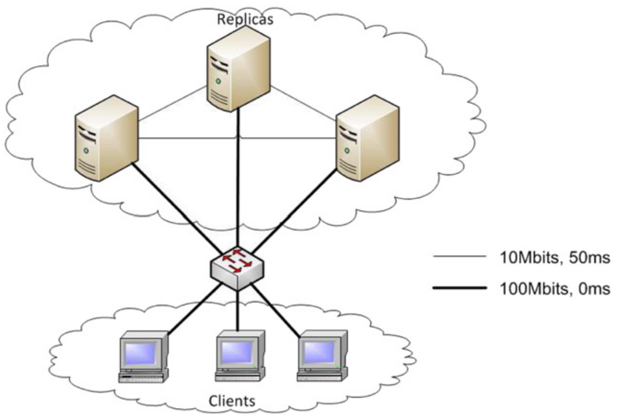


Fig. 6. Topology used for Emulab experiments.

being able to sustain only 1.17 parallel instances as compared to 1.63 of the network. Such a situation is typical of systems where the network is comparatively faster than the process,[4] which is typical in a cluster environment.

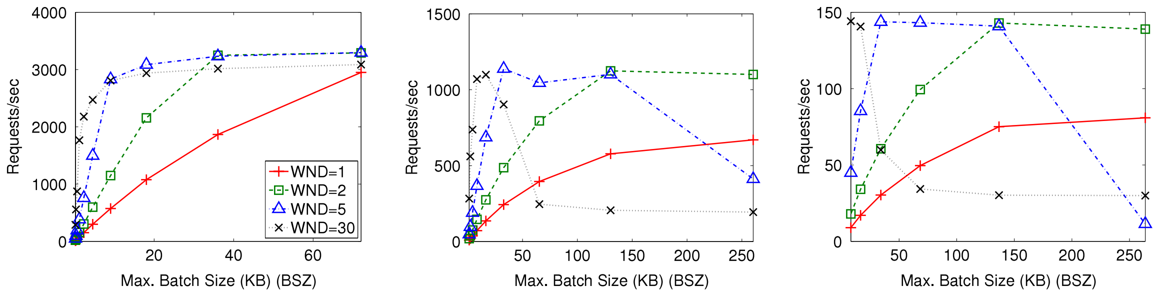
The model also captures the effect of batching on performance. As the size of the batches increases, the total instance time increases reflecting the larger size, but the average time per request (ϕ^{CPU}_{req}) decreases. This is very noticeable for 128-byte requests, with the average time per request dropping from 0.52 ms down to 0.08 ms for the largest batches.

The results also show that the predicted optimal window size matches closely the value obtained in the experiments. Furthermore, the predicted throughput is close to the experimental results, with less than 5% error for 128 bytes and 1 KB requests, and at most 20% for 8 KB requests 512 KB batch size.

5.2. Emulab

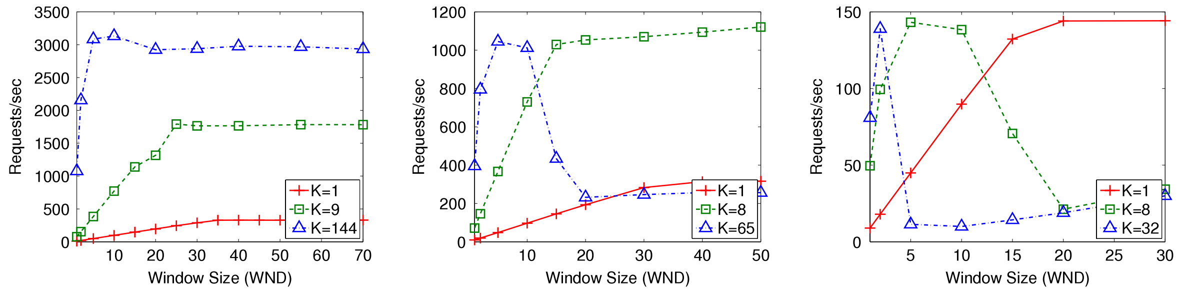
Fig. 6 shows the topology used for the Emulab experiments, which represents a typical WAN environment with the geographically distributed nodes. The replicas are connected point-to-point by a 10-Mbit link with 50 ms latency. Since the

[4] The speed of the process depends not only on the CPU speed but also on the efficiency of the Paxos implementation.



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

Fig. 7. Experimental results in Emulab: throughput with increasing batch size.



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

Fig. 8. Experimental results in Emulab: throughput with increasing window size. K represents the number of requests that fit on a batch, *i.e.*, $K = \lfloor BSZ/reqSize \rfloor$.

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 850 MHz with 512 MB of memory, connected by a 100 Mbps Ethernet.

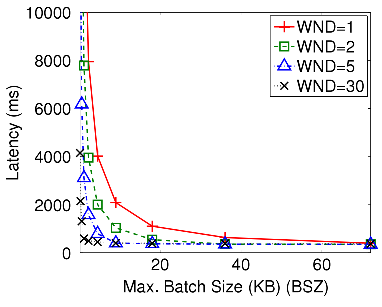
5.2.1. *Experimental results*

Fig. 7 shows the throughput in requests per second for increasing values of *BSZ*, while Fig. 8 shows similar data but plotted against the values of *WND*. Both figures include results for requests sizes of 128, 1 KB and 8 KB. Fig. 9 gives more detail on the tests with request size of 128 bytes, showing the client latency, instance latency and instance throughput.

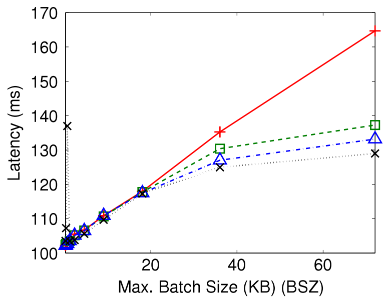
The results show that, contrary to the cluster experiments, batching alone does not suffice to achieve maximum throughput. Although using only batching (Fig. 7, $WND = 1$) improves performance significantly, it falls short of the maximum that is achieved with larger window sizes. The difference is greater with large request sizes (1 KB and 8 KB), where batching alone achieves only half of the maximum throughput, than for small sizes (128 bytes), where it 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 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 (40 KB with $S_{req} = 128$ and around 140 KB 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.

Using pipelining alone (Fig. 8, $K = 1$, where K is the maximum number of requests that fit in a batch, *i.e.*, $K = \lfloor BSZ/reqSize \rfloor$) is not enough to achieve the best performance with small and medium request sizes (128 bytes and 1 KB), but is enough for large request sizes (8 KB). With small request sizes, the performance gains are quite modest, as it does not even reach 500 requests per second, far from the maximum of 3000. With large request sizes, however, with $WND \geq 15$ the system reaches the maximum performance. The difference in behavior is once again due to different bottlenecks; with small request sizes the bottleneck is mainly the CPU, so batching provides the most gains by decreasing the average per request CPU utilization. On the other hand, with large requests the bottleneck is the network, either the latency or the bandwidth, and in this case batching does not help. When the limitation is the network latency, pipelining improves throughput significantly, as seen in Fig. 8(c) until $WND = 15$. After this point, the system is limited by the network bandwidth, so neither batching nor pipelining can improve the results further.

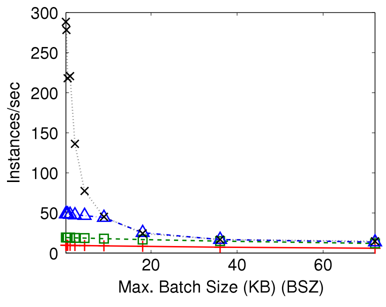
The experiments also show that increasing the window size too much results in a performance collapse, with the system throughput dropping to around 10% of the maximum. This happens when the leader tries to send more data than the capacity of the network, resulting in packet loss and retransmissions. 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



(a) Client latency.



(b) Instance latency.



(c) Instances/sec.

Fig. 9. Emulab, experimental results with $S_{req} = 128$.

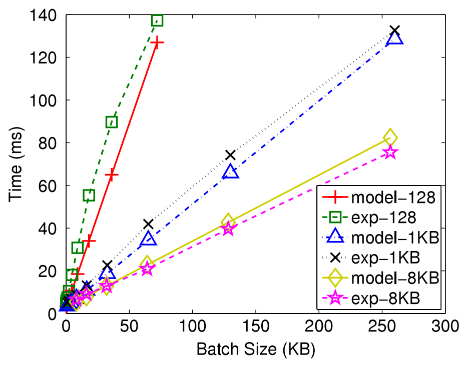


Fig. 10. Experimental versus model results for the CPU time of an instance. Fit values: $\phi_{cli}(x) = 0.28x + 0.2$, $\phi_{rep}(x) = 0.002x + 1.5$.

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 $BSZ = 256$ KB (Fig. 7(b)). For larger *WND*, the performance collapse happens with smaller values of *BSZ*: with $WND = 10$ at 130 KB, and at less than 64 KB 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.

To conclude the analysis of the experimental results, we look in more detail at the experiments with request size of 128 bytes (Fig. 9). As the batch size increases the client latency goes from a maximum of over 10 s down to less than 0.5 s (Fig. 9(a)). This is a indirect consequence of the batch size. Recall that larger batch sizes result in higher throughput (Fig. 7(a)). Since the experiments were performed with a fixed request load, then higher throughput reduces the time that client requests spend waiting in queues, which dramatically decreases the client latency.

The instance latency (Fig. 9(b)) increases with *BSZ*, with the minimum being the round-trip time (100 ms). For values of *WND* other than 1, the increase is not linear. The reason is that the instance latency depends on the effective batch size, which may be smaller than the maximum batch size (*BSZ*). This is the case with values of *WND* greater than one, because as the leader is allowed to execute parallel instances, it does not always fill the batches completely before the batch timeout expires.

The throughput in instances (Fig. 9(c)) is the highest when *WND* is large and *BSZ* small, as the leader is able to fill up batches and start new instances before the previous ones finish. However, since each batch is small, the throughput in requests is low. As *BSZ* increases, the leader will wait more to start a batch, therefore resulting in a smaller throughput in instances. When *WND* is small the throughput in instances is small because the leader is restricted in how many instances it can execute in parallel, so even if it has batches ready, it must wait for the previous instances to finish.

5.2.2. Setting model parameters

Following the same procedure as in the case of the cluster, we have determined the following parameters for the Emulab model: $\phi_{cli}(x) = 0.28x + 0.2$, $\phi_{rep}(x) = 0.002x + 1.5$. Fig. 10 shows the training data and the corresponding model results when parametrized with the values above.

Table 4
Emulab: comparison of analytical and experimental results for different batch sizes (S_{2a}). Prediction of optimal w [$= \min(w^{\text{CPU}}, w^{\text{NET}})$] is in bold. The column w shows the range that was determined experimentally to contain the smallest value of *WND* that produces the maximum throughput.

S_{2a}	Model (predictions)			Experiments	
	w^{CPU}	w^{NET}	Max Thrp	w	Max Thrp
$S_{req} = 128$					
128	**30.88**	833.48	308	30–35	≈330
256	**28.77**	422.94	574	25–30	≈550
1 KB	**20.45**	107.58	1620	20–25	≈1800
16 KB	**3.38**	7.68	3765	2–5	≈3100
32 KB	**1.47**	3.12	4032	1–2	≈3300
$S_{req} = 1$ KB					
1 KB	**28.89**	119.01	286	30–40	≈310
2 KB	**25.54**	60.12	502	30–40	≈600
8 KB	**15.42**	15.8	1155	15–20	≈1030
128 KB	3.16	**1.93**	1184	1–2	≈1120
256 KB	2.68	**1.6**	1184	1–2	≈1100
$S_{req} = 8$ KB					
8 KB	19.47	**16**	150	15–20	≈144
16 KB	14.24	**8.5**	150	5–10	≈144
64 KB	6.72	**2.88**	150	2–5	≈144
128 KB	4.84	**1.94**	150	1–2	≈144
256 KB	3.8	**1.47**	150	1–2	≈144

5.2.3. *Comparison of analytical and experimental results*

Table 4 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-byte requests is the CPU (w^{CPU} is smaller than w^{NET}) while for 8 KB requests it is the network. With 1 KB 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 4 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 $S_{req} = 8$ KB 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.[5] The CPU exhibits a more non-linear behavior, especially when under high load. This in turn requires carefully tuning of the maximum number of parallel instances, as higher values increase CPU utilization.

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 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

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

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 experiments were performed with a null service in order to put more stress on the ordering protocol. With a non-null service, the maximum throughput of the system would be inversely proportional to the execution time of the requests. Hence, as the execution time increases, the impact of the optimizations would become less and less pronounced. The analytical model can be used to predict what would happen in this case, by setting the ϕ_{exec} parameter to the appropriate value.

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 1 KB and we have determined that the batch size should be 8 KB, then the window size should be set to 16 (Table 4(b)).

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.

Acknowledgments

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

References

[1] L. Lamport, The part-time parliament, ACM Transactions on Computer Systems 16 (2) (1998).
[2] R. Friedman, R. Renesse, Packing messages as a tool for boosting the performance of total ordering protocols, Tech. Rep. TR95-1527, Department of Computer Science, Cornell University, 1995.
[3] A. Bartoli, C. Calabrese, M. Prica, E. Di Muro, A. Montresor, Adaptive message packing for group communication systems, in: OTM 2003 Workshops, in: LNCS, Springer, 2003.
[4] B. Carmeli, G. Gershinsky, A. Harpaz, N. Naaman, H. Nelken, J. Satran, P. Vortman, High throughput reliable message dissemination, in: Proceedings of the 2004 ACM Symposium on Applied Computing, NY, USA, 2004. http://doi.acm.org/10.1145/967900.967971.
[5] R. Friedman, E. Hadad, Adaptive batching for replicated servers, in: Symposium on Reliable Distributed Systems, SRDS'06, 2006. http://dx.doi.org/10.1109/SRDS.2006.8.
[6] J. Nagle, Congestion control in IP/TCP internetworks, Tech. Rep. RFC 896, IETF, Jan. 1984.
[7] V.N. Padmanabhan, J.C. Mogul, Improving HTTP latency, Computer Networks and ISDN Systems 28 (1–2) (1995).
[8] Y. Amir, J. Kirsch, Paxos for system builders, Tech. Rep. CNDS-2008-2, Johns Hopkins University, 2008.
[9] R. Guerraoui, R. R. Levy, B. Pochon, V. Quéma, Throughput optimal total order broadcast for cluster environments, ACM Transactions on Computer Systems 28 (2) (2010). http://dx.doi.org/10.1145/1813654.1813656.
[10] P. Marandi, M. Primi, N. Schiper, F. Pedone, Ring Paxos: A high-throughput atomic broadcast protocol, in: Dependable Systems and Networks, DSN'10, 2010. http://dx.doi.org/10.1109/DSN.2010.5544272.
[11] Y. Mao, F. P. Junqueira, K. Marzullo, Mencius: building efficient replicated state machines for WANs, in: Proceedings of the 8th USENIX conference on Operating systems design and implementation, OSDI'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 369–384. http://portal.acm.org/citation.cfm?id=1855741.1855767.
[12] X. Défago, A. Schiper, P. Urbán, Total order broadcast and multicast algorithms: Taxonomy and survey, ACM Computing Surveys, 36.
[13] B. White, 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, 2002.
[14] N. Santos, J. Kończak, T. Zurkowski, P. Wojciechowski, A. Schiper, JPaxos — State machine replication in Java, Tech. Rep. 167765, EPFL, Jul. 2011.