Tuning Paxos for high-throughput with batching and pipelining

Nuno Santos, André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
Email: firstname.lastname@epfl.ch

***Abstract*—Paxos is probably the most known state machine replication protocol. Two optimizations that can greatly improve its performance are batching and pipelining. Their effectiveness depends significantly on the system properties, mainly network latency and bandwidth, but also on the CPU speed and properties of the application. This makes it hard to know when and how to use each optimization to achieve maximum throughput with Paxos. We address this question, first analytically, and then experimentally. The results show that although batching by itself is often sufficient to achieve maximum throughput in low latency networks, on a WAN setting it must be complemented with pipelining.**

***Keywords*-Paxos; Batching; Pipelining; Analytical study; Experimental evaluation**

I. 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 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 $1/2L$ 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 optimizations to the basic Paxos protocol that address these limitations: batching and pipelining.

Submitted to Dependable Systems and Networks (DSN 2011)

Batching consists of packing several requests in a single instance. The main benefit is spreading the fixed per-instance costs over several requests, which results in a smaller per-request overhead and potentially in a higher throughput. Batching can easily be implemented on top of Paxos, as it does not require any changes to the ordering protocol.

Pipelining of instances is described in the original Paxos paper [1]. It is an extension of the basic protocol, allowing the leader to initiate 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 the instances on the slow link.

Batching and pipelining are used by most replicated state machine implementations, as they usually provide performance gains between one and two orders of magnitude. Nevertheless, in order to achieve the best throughput, they must be carefully tuned. For batching, it is necessary to define a policy for choosing the size of batches and deciding when to form a batch. This question has been studied previously, see [2], [3], as a general optimization technique for communication protocols. For pipelining, it is necessary to set a limit on the number of instances that can be in execution simultaneously. Setting a value that is too high may lead to a significant degradation in performance, as the system spends an increasing fraction of its resources juggling multiple instances. Moreover, the optimal choice for the size of batches and the number of parallel instances depends on the properties of the system and of the application, mainly on bandwidth, latency, and size of client requests.

In this paper we study these two optimizations in the context of Paxos. We begin by studying analytically what are the combinations of batch size and number of parallel instances that maximize network utilization in any given network. We express this relationship in terms of a function $w = f(S_{batch})$, where S_{batch} is a batch size and w is a number of parallel instances. 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 and the two optimizations combined. The results suggest that batching is usually enough when the latency is low, but on higher latency networks, it must be combined with parallel instances in order to achieve the best

performance.

The rest of the paper is organized as follows. Section II provides the background for our work, describing in more detail the batching and pipelining optimizations in Paxos. Section III presents the analytical model relating batch size and number of parallel instances with network utilization. Section IV presents the experimental evaluation of these two optimizations on a LAN (a cluster) and on a WAN (emulated with Emulab). Section V discusses the results of the paper. Section VI presents the related work, and Section VII concludes the paper.

II. Background

A. The Paxos Protocol

Paxos, or more precisely MultiPaxos, is a state machine replication protocol, which at its core uses the Synod consensus algorithm [1]. If the replication degree is n, and f out of the n replicas may fail by crashing, the protocol requires $n \geq 2f + 1$. MultiPaxos can be seen as a *sequencer-based* atomic broadcast protocol [4], where the sequencer orders requests received from the clients. In the Paxos terminology, the sequencer is called *leader*. Paxos is usually described in terms of proposers, acceptors and learners, which are the roles each process can play. Here we ignore the different roles by assuming that every node plays all three roles.

For the purpose of the paper we describe only the relevant details of the Paxos protocol. Figure 1 shows the message pattern of Paxos for the case $n = 3$, $f = 1$. Once a process becomes leader (p_1 in Figure 1), it executes Phase 1 only once for all future instances. Afterwards, for each new request received from the clients, it only needs to execute Phase 2 (a request is ordered at the leader upon reception of enough Phase 2b messages), thereby saving two message delays per consensus instance. Therefore, in our analysis we ignore Phase 1 messages.

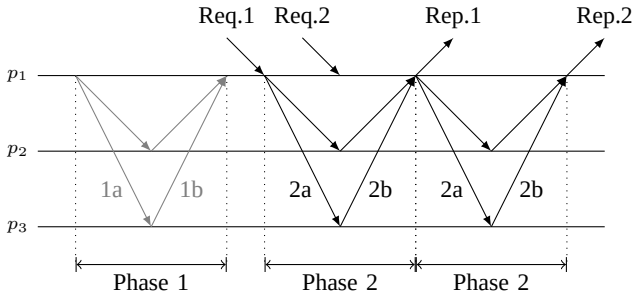


Figure 1. Basic MultiPaxos ($n = 3$, $f = 1$)

In the simplest version of MultiPaxos, the leader proposes one request per instance and executes one instance at a time. Next we describe the two optimizations that are the focus of this work: pipelining and batching.

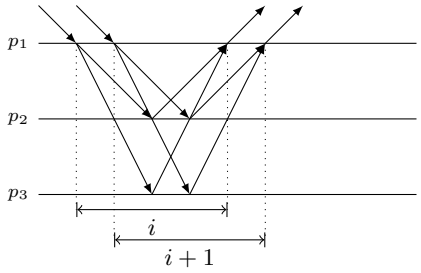


Figure 2. Paxos with pipelining.

B. Pipelining

MultiPaxos can be extended to allow the leader to execute Phase 2 for several instances in parallel [1]. In this case, when the leader receives a new request, it can start a new instance to order the request without waiting for the end of the previous instances, as shown in Figure 2.

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 additional instance requires more resources from the system. If too many instances are started in parallel, they may overload the system 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 IV. The best value depends on many factors, including the network latency, the size of the requests, the speed of the replicas, and expected workload.

C. Batching

Batching is a common optimization in communication systems which generally provides large gains in performance [5]. It can also be applied to Paxos, as illustrated by Figure 3. Instead of proposing one request per consensus instance, the leader packs several requests in the value of a single consensus 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 a consensus instance over several requests, thereby decreasing the average per-request overhead. For each consensus instance, the system performs several tasks that take a constant time regardless of the size of the value being ordered, 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

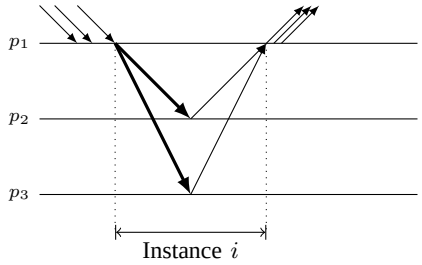


Figure 3. Paxos with batching.

Symbol	Description
B	Bandwidth
L	One way delay (latency)
S_{req}	Size of request
S_{ack}	Size of ack
S_{ans}	Size of answer sent to client
S_{batch}	Batch size
w	Number of parallel instances

Table I
NOTATION.

data to the network card, allocating buffers, updating the replicated log and the internal data structures, and executing the protocol logic. In [2], 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. For a consensus instance, the fixed costs are an even larger fraction of the total costs because, in addition to processing individual messages, processes also have to execute the consensus algorithm. Additionally, batching decreases dramatically the cost of using stable storage, 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 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 value 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], [6]. In the rest of the paper, we study it in the context of Paxos, and analyze its interaction with the pipelining optimization.

III. PAXOS WITH OPTIMAL NETWORK UTILIZATION

In this section we answer the question of how to configure Paxos to achieve maximum network utilization. This is a similar problem as the one of tuning TCP to maximize link utilization, where the sender must be able to send enough unacknowledged data to keep the link full at all time. For TCP the solution is given by the bandwidth-delay product: the sender should be able to send up to $2BD$ unacknowledged data, where B and D are, respectively, the bandwidth and the delay of the link. In the context of Paxos, the problem is more complex. First, there are more processes and more links involved, therefore resulting in a more complex communication pattern and more potential bottlenecks. Second, the amount of unacknowledged data in Paxos is determined by two variables: the size of the batches (S_{batch}) and the number of parallel instances in execution (w). As we show below, there are several combinations of S_{batch} and w that maximize network utilization.

We start by determining the bottleneck link for Paxos and, from this, we compute the maximum throughput in requests per second. We then determine analytically S_{batch} and w that enable Paxos to reach this maximum network throughput.

We consider the Paxos variant described in Section II 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 [7]. We chose the basic variant for generality and simplicity, but this analysis can be easily adapted to other variants.

We also assume full duplex links and that no other application is competing for bandwidth.[1] 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. Table I summarizes the notation used in the rest of the paper.

A. Network bottleneck: outgoing link of the leader

In this subsection we argue that the outgoing link of the leader is the network bottleneck of the system. This allows us to focus the analysis in the following sections on this link.

It is obvious that both the incoming and outgoing links of the leader carry more data than the links of the other replicas, since the leader sends the value to all replicas and receives the requests from the client, while the replicas only receive

[1]The presence of other applications can be modeled by adjusting the bandwidth and latency parameters, to reflect the competition for network resources.

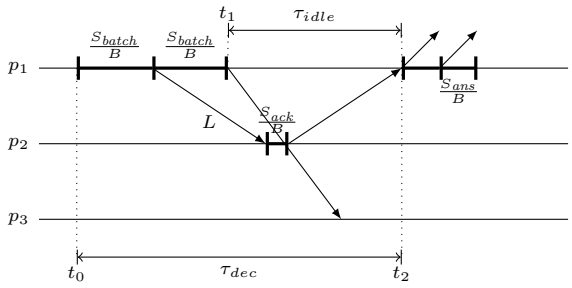


Figure 4. Decision and idle time for an instance

the value from the leader and send an acknowledgment to the leader.

We show that, under the reasonable assumption $S_{req} \geq S_{ack}$ and $S_{ans} \geq S_{ack}$, the outgoing link of the leader carries more data than the incoming link, and therefore is the network bottleneck. These assumptions hold for most applications, because Phase 2b messages (acks) contain only a view and an instance number, while client requests and respective answers contain at least a request id and an application payload. Let T_{in} and T_{out} be the total data received, respectively transmitted, by the leader for each instance of Phase 2, including the requests from the clients and the replies sent to the clients. For each such instance, the leader receives requests from k clients to fill a batch, sends this information to the $n-1$ replicas (Phase 2a), receives $n-1$ ack messages (Phase 2b), and finally sends k replies back to the clients. So we have:

$$T_{in} = kS_{req} + (n-1)S_{ack}$$
$$T_{out} = (n-1)kS_{req} + kS_{ans}$$

If $S_{req} \geq S_{ack}$ and $S_{ans} \geq S_{ack}$, then we can derive directly from the equations above that $T_{out} \geq T_{in}$.

B. Maximum throughput of the network bottleneck

Since the outgoing link of the leader is the bottleneck, we can easily compute the theoretical maximum throughput of Paxos. This maximum is obtained when the outgoing link (which for each request must carry $n-1$ requests (Phase 2a) and the response) is used at 100%. Therefore, the maximum throughput is

$$\frac{B}{(n-1)S_{req} + S_{ans}} \tag{1}$$

In Section IV we compare the value obtained by this expression with the actual maximum throughput obtained experimentally.

C. Computing the duration of one instance of Phase 2

In order to compute the values of w and S_{batch} that keep the outgoing link fully utilized, we compute τ_{dec}, which is the time required to execute one instance of Phase 2. The time for one instance of Phase 2 is the time from the sending of the first Phase 2a message to the reception of enough Phase 2b messages to commit the order. To commit an instance, the leader must receive a majority of Phase 2b messages. This happens only if the Phase 2a message was received by a majority of replicas. Sending the Phase 2a message to a majority requires the following time:

$$\tau_{maj} = \lfloor n/2 \rfloor \frac{S_{batch}}{B} \tag{2}$$

After the leader finishes sending the Phase 2a message to a majority of processes, it takes an additional L time until the last process forming a majority receives the message. This last process then answers to the leader, sending a Phase 2b message of size S_{ack}, which takes an additional L time to be received by the leader. Assuming there is no delay from the time a process receives a message until it answers, the time required to execute one instance of Phase 2 is

$$\begin{aligned} \tau_{dec} &= \tau_{maj} + 2L + \frac{S_{ack}}{B} \\ &= 2L + \frac{\lfloor n/2 \rfloor S_{batch} + S_{ack}}{B} \end{aligned} \tag{3}$$

The instance latency has a fixed lower bound of $2L$ and a variable part that increases with the size of the batch and the number of replicas, and decreases with the bandwidth. Out of these parameters, the only that typically can be tuned is the size of the batch. The formula shows that changing the batch size has greater effect when the bandwidth is low and the number of replicas high.

D. Keeping the outgoing link of the leader fully utilized

The maximum network throughput is reached when the network bottleneck, *i.e.*, the outgoing link of the leader, is fully utilized. We compute here w as a function of S_{batch} in order to satisfy this constraint.

We have just computed τ_{dec}, the duration of Phase 2. During τ_{dec} time, the leader can send $\tau_{dec} \cdot B$ bytes. Since each instance requires the leader to send $(n-1)S_{batch}$ bytes, we have

$$w = \left\lceil \frac{\tau_{dec} \cdot B}{(n-1)S_{batch}} \right\rceil$$

In other words, w is inversely proportional to S_{batch}. Indeed, if S_{batch} is large enough, one instance of Phase 2 (i.e., $w = 1$) is enough to keep the outgoing link of the leader fully utilized. If S_{batch} is not large enough to keep the outgoing link of the leader fully utilized, we must have $w > 1$. Using (3) we get:

$$w = \left\lceil \frac{2LB + S_{ack}}{(n-1)S_{batch}} + \frac{\lfloor n/2 \rfloor}{(n-1)} \right\rceil$$

For n odd, we have $\frac{\lfloor n/2 \rfloor}{(n-1)} = 1/2$. We ignore even values of n as they do not provide any advantage in resiliency

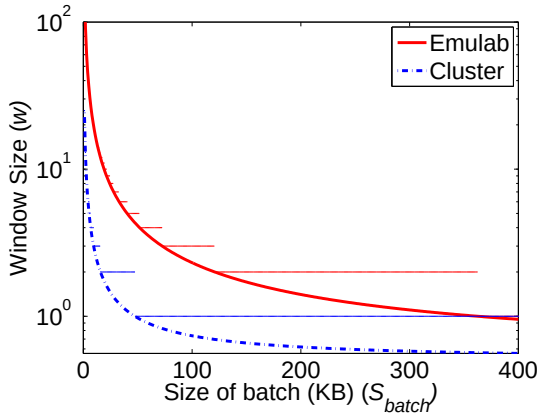


Figure 5. Function (4) for the settings used in Section IV.

over the immediate lower odd number. Therefore, we have the following dependency of w on S_{batch} for maximal throughput:

$$w = f(S_{batch}) = \left\lceil \frac{2LB + S_{ack}}{(n-1)S_{batch}} + 1/2 \right\rceil \quad (4)$$

The function $w = f(S_{batch})$ is represented by the discrete horizontal lines in Figure 5, for the two settings used in Section IV (a cluster on a LAN and a WAN-like topology emulated in Emulab). The continuous lines represents (4) without $\lceil\ \rceil$. We can reverse this function to define S_{batch} in terms of $w \in \mathbb{N}$:

$$S_{batch} = g(w) = \frac{2LB + S_{ack}}{(n-1)(w - 1/2)} \quad (5)$$

E. Detailed timing analysis of an instance

Up to now, we have focused only on determining the relation between S_{batch} and w that maximize network utilization, as this result has practical utility. Formulas (4) and (5) provide the answer to this question, but by themselves do not explain what happens during an instance of the ordering protocol that leads to the results obtained. In this section we explore this question, by looking in detail at the different phases of an instance of the ordering protocol, and breaking down how each stage is affected by the system and application parameters (*i.e.*, network latency, bandwidth, and size of requests). Although these results are not directly useful in practice, they provide additional insight into the network behavior of Paxos, thus complementing the results above.

We start by determining how long the leader takes to send the phase 2a message to all, which is the time to write (n-1) messages to a link with bandwidth B:

$$\tau_{all} = (n-1)\frac{S_{batch}}{B}$$

S_{batch}	1KB	64KB	128KB	362KB
τ_{all}	1.7	100.9	203.5	600
τ_{idle}	299.2	249.5	198.2	0
τ_{dec}	300.8 ms	350.5 ms	401.8 ms	600 ms
$w = f(S_{batch})$	$\lceil 181.8 \rceil = 182$	$\lceil 3.5 \rceil = 4$	$\lceil 1.97 \rceil = 2$	1

Table II
EMULAB TOPOLOGY: TIMING ANALYSIS OF AN INSTANCE.
($n = 3, L = 150ms, B = 9.59$Mbit/s, $S_{req} = 1$KB, $reqOverhead = 16B, S_{ans} = 1$KB, $S_{ack} = 0$)

From this formula, we can compute the idle time of an instance as:

$$\tau_{idle} = \tau_{dec} - \tau_{all} = 2L + \frac{S_{ack} - \lfloor n/2 \rfloor S_{batch}}{B}$$

There are three main factors that control how many additional instances the leader can execute in a given network: the network latency, the bandwidth and the size of the batch. After finishing sending to a majority of replicas, the leader has to wait for a round trip, which is $2L$, thus higher network latency increase the idle time. A higher bandwidth also leads to more idle time, as messages take less time to be written. On the other hand, larger batches take longer to be written to the network, thus decreasing the idle time. Note that τ_{idle} may be negative, in which case the leader does not have time to execute more than one instance simultaneously. We show an example of this situation below.

We can apply the formulas above to the scenarios used for the experiments in Section IV.

In the Emulab topology (Table II), with 1KB batches, the leader takes 1.7ms to send the Phase 2a messages to all replicas and 300.8ms to decide. In this case, the decision time is dominated by the network latency, because 300ms corresponds to two message delays. The outgoing link of the leader is free for most of this time, more precisely, for 299ms out of 300ms, which gives the leader enough time to send up to 181 additional instances, hence $w = 182$. As the batch size increases, the leader takes longer to send all messages and there is less time available for parallel instances. With batches of 128KB, the leader takes 402ms to decide, 203 of which are spent sending the phase 2a messages. Therefore, the leader's outgoing link is free only for 198ms, which is enough to send most of an additional instance[2]. With the optimal batch size, the leader takes 600ms to send the phase 2a messages to all, which is exactly how long it takes to decide. Hence, there is no idle time and $w = 1$. These results show that when the latency is high, there is a considerable amount of idle time available to execute parallel instances. The fraction of idle time decreases with the size of the batch, as for larger batches the leader takes longer to send to all, thus using the outgoing link for longer.

[2]More precisely, 0.97 of an instance. The value of w already includes the first instance.

S_{batch}	1KB	47KB	64KB
τ_{all}	0.02	0.8	1.03
τ_{idle}	0.39	0	-0.12
τ_{dec}	0.41 ms	0.8 ms	0.92 ms
$w = f(S_{batch})$	$\lceil 16.5 \rceil = 17$	1	$\lceil 0.93 \rceil = 1$

Table III
CLUSTER TOPOLOGY: TIMING ANALYSIS OF AN INSTANCE. ($n = 3, L = 0.2ms, B = 940\text{Mbit/s}, S_{req} = 1\text{KB}, S_{ans} = 1\text{KB}, S_{ack} = 0\text{KB}$)

Table III shows a similar analysis for the cluster topology. With a 1Gbit/s of bandwidth, it takes only 0.02ms to send all phase 2a messages for batches of 1KB. Nevertheless, because of the 0.2ms latency, the leader decides 0.41ms after having started the instance, leaving a total of 0.39ms of idle time. This time is enough to send the phase 2a messages for 16 additional instances. With larger batches, the τ_{idle} decreases, reaching 0 with 47KB batches. Further increasing the batch leads to negative idle time, which means that the leader decides before finishing sending all phase 2a messages. That is, sending the messages to the last $\lfloor n/2 \rfloor$ replicas, takes longer than receiving the phase 2b messages from the first $\lfloor n/2 \rfloor$ replicas.

IV. EXPERIMENTAL STUDY

In this section we study the batching and pipelining optimizations from an experimental perspective. We evaluate the gains provided by these two optimizations in two settings, WAN and cluster, as well as their performance when used alone and when used in combination. Our goal is to characterize the gains provided by these optimizations in different scenarios, and to understand how to best tune them for optimal performance in such scenarios.

The experiments complement the analytical results in Section III. Section III looks at the system in a steady state, assuming that the every batch has size S_{batch} and that at any moment there are w instances executing. The model tells us that if a system reaches a steady state where S_{batch} and w satisfy the equations (4) and (5), then the system is at the maximum network throughput. But the model leaves open the question of how to achieve such a steady state. This depends in many factors, like the load offered by the clients, the speed of the system, and the policy used to create batches. In some cases it may not be possible at all to reach an optimal steady state. This is the case if clients are not sending enough requests to reach the maximum network throughput or if the nodes are not fast enough to sustain it. More interestingly, in other cases there are combinations of optimal S_{batch} and w pairs that may be reachable while others are not (see Emulab experiments). For instance, a system with slow CPUs might more easily sustain a large batch size with few instances than small batches with many instances. In this case, the system should be tuned to try to reach one of the optimal steady states that are within range of the physical capabilities, by using the analytical model to determine all the optimal steady states. Tuning the system in this way can be done mainly through the batching/pipelining policy, which we discuss next.

A. Defining a policy to control pipelining and batching in Paxos

Implementing batching and pipelining in Paxos are fairly straightforward tasks: batching has a trivial implementation and pipelining was described in the original Paxos paper [1]. But these implementations leave open the task of defining a policy to control these optimizations, that is, to decide when and how to form a new batch and start a new instance. This policy is the key to control the tradeoffs involved in batching and pipelining that were discussed in Section II.

The algorithm we used in our experiments appears in Figure 6. It can be used as part of any Paxos implementation that has support for batching and pipelining. The algorithm has 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, and Δ_B is used as a timeout on building a batch: upon expiration, the batch is proposed even if its size is less than BSZ.

The parameter WND is strict, in the sense that the system will never exceed it, but BSZ and Δ_B are indicative and can be exceeded in some situations that are mentioned below.

```
1:  Initialization:
2:    reqQueue {queue with client requests waiting to be ordered}
3:    w ← 0 {number of active instances}
4:    fork main() task

5:  upon instance decided
6:    w ← w − 1

7:  procedure buildBatch()
8:    Batch ← ∅
9:    while true do
10:     wait until reqQueue not empty or age(Batch) + Δ_B ≥ C_t()
11:     while reqQueue not empty do
12:       r ← reqQueue.first()
13:       if size(Batch) + |r| ≤ BSZ then
14:         Batch ← Batch ∪ {r}
15:         reqQueue.removeFirst()
16:       else
17:         return Batch
18:     if size(Batch) ≥ BSZ or age(Batch) + Δ_B ≥ C_t() then
19:       return Batch

20: procedure main()
21:   while true do
22:     wait as long as w = WND
23:     Batch ← buildBatch()
24:     start new instance with Batch
25:     w ← w + 1
```

Figure 6. Algorithm used to batch requests and start consensus instances. $size(Batch) = \sum_{r' \in Batch} |r'|$, $age(Batch) = \min\{r'.ts : r' \in Batch\}$ where $r.ts$ is the local time when the request was received from the client, and $C_t()$ returns the local time.

The queue $reqQueue$ (Line 2) is used to store the client requests waiting to be ordered. We assume that a dedicated

thread receives the requests from the clients and places them in this queue. The variable w (Line 3) keeps track of the number of active instances.

The heart of the algorithm is the loop in the procedure *main()*. Whenever the number of active instances is lower than the maximum WND (Line 22), a new instance may be started using the function *buildBatch()*. This function uses a combination of time and size-based strategies. A new instance is started either when there are enough requests available or when some request has been waiting for longer than Δ_B.

The bounds Δ_B and WND can be exceeded in some situations, which are hard to avoid in a general way. When the load generated by the clients is higher than the capacity of the system, a request may be delayed for more than Δ_B (Line 22). Additionally, if some request is larger than BSZ, the algorithm creates a batch larger than BSZ. There are other ways of handling large requests, like refusing the request, splitting it into several parts or transferring it using a parallel channel. We chose to allow exceeding the batch size for simplicity.

In our experiments we study BSZ and WND, while keeping Δ_B set to 10ms. Under moderate to high load, the leader always forms a new batch before $\Delta_B = 10ms$ expires, so the parameter had no impact on our experiments.

B. Experimental settings

We performed the experiments using JPaxos [8], a full implementation of Paxos in Java.

We consider two typical deployment scenarios: a WAN (emulated over Emulab [9]) and a cluster. In both cases, we evaluate three configurations: batching alone, pipelining alone, and the two optimizations combined.

We consider a system with three replicas and a variable number of clients. The replicated service used in these experiments receives requests containing an array of bytes and replies back with the same array. It keeps no state. We chose a simple service as this puts the most stress on the replication mechanisms. All the experiments were done with a request size of 1KB. JPaxos adds a header of 16 bytes per request and 4 bytes per batch of requests. The analytical results reported in tables IV and V take the protocol overhead in consideration.

Clients send requests synchronously to the leader, waiting for the reply before submitting the next request. All the experiments were done with the system under medium to high load (*i.e.*, 100 or more clients), which is when batching and pipelining have the most impact on the performance.

All communication is done over TCP. We did not use IP multicast because it is not generally available on 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 having flow and congestion control, and of having no limits on message size, therefore saving us the tedious work of reimplementing these features. 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 6 minutes run, excluding the first 10%. For clarity, the plots shown here do not include error bars for the 95% confidence interval. The errors in most cases are very small, so the error bars would be hard to distinguish from the data points.

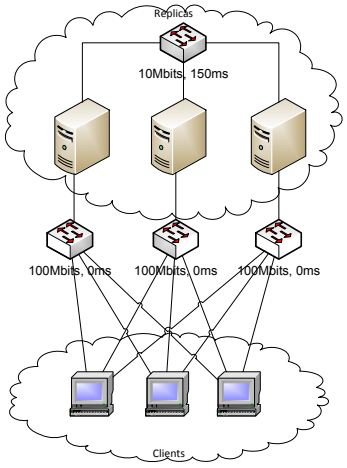


Figure 7. Topology used for Emulab experiments

The metrics analyzed are the latency per instance, latency per request, throughput of instances and throughput of requests. The *latency per instance* corresponds to τ_{dec}, see Figure 4: it is the time elapsed since the leader sends a Phase 2a message until it receives a majority of Phase 2b messages. The *latency per request* is the time the client waits for the reply to one 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. The *throughput of instances* is the number of Phase 2 executed per second, and the *throughput of requests* is the number of requests ordered per second (corresponds to Formula (1)). Note that the throughput of requests is equal to the throughput of instances multiplied by the average number of requests per instance. In some of the experiments we also show the average size of the batches formed by the batching/pipelining algorithm and the average number of instances open at any given moment (computed as a time series average), when they significantly differ from their bounds, BSZ and WND.

C. Emulab experiments

Figure IV-C shows the topology used for the Emulab experiments. This network is an example of a WAN, where the bottleneck of the system is usually the network latency, while the nodes are comparatively fast. The replicas are connected by an emulated network with $150ms$ latency and 10Mbit/s of bandwidth. The effective bandwidth between

S_{batch}	1KB	64KB	128KB	362KB
$w = f(S_{batch})$	$\lceil 181.8 \rceil = 182$	$\lceil 3.5 \rceil = 4$	$\lceil 1.97 \rceil = 2$	1
τ_{dec}	300.8 ms	350.5 ms	401.8 ms	600 ms

Table IV
FORMULA (4) FOR THE EMULAB TOPOLOGY AND THE CORRESPONDING INSTANCE DURATION (τ_{dec}),
($n = 3, L = 150ms, B = 9.59\text{Mbit/s}, S_{req} = 1\text{KB}, reqOverhead = 16B, S_{ans} = 1\text{KB}, S_{ack} = 0$)

replicas, as reported by the `netperf`[3] diagnostic tool, is 9.59 Mbit/s. Since the clients are used only to generate load and are not the focus of the experiment, they are connected directly to every replica over a link with no emulated latency and 100Mbit/s bandwidth[4]. The routing is configured so that replicas do not use the fast links to/from the clients to bypass the slow links when communicating with other replicas.

Using the results in Section III, Table IV shows several combinations of values for S_{batch} and w that maximize network utilization. We start with experiments showing that batching without pipelining and pipelining without batching, do not lead to optimal performance.

1) Batching without pipelining: Figure 8 presents the results of the experiments using only the batching optimization. Figure 8d shows that the gains from batching are substantial, improving the throughput from 3 requests/s with $BSZ = 1\text{KB}$ (one request/instance) to around 250 requests/s with $BSZ = 128\text{KB}$. For larger batches the performance degrades slightly.

To understand this behavior, note that the throughput of requests is equal to the product of the throughput of instances (Figure 8c) and the number of requests in each instance (Figure 8e). For 500 and 900 clients, as the maximum batch size increases, the number of requests per instance increases linearly while the throughput of instances decreases. Up to 128KB, the additional number of requests per instance is enough to offset the degradation in instance throughput, but between 128KB and 256KB the instance throughput degrades faster than the increase in batch size, thereby resulting in worst overall throughput. With 100 clients the maximum throughput is not reached because the offered load is not enough to fill up batches larger than 50KB (Figure 8e).

In Figure 8a we can see that with $BSZ = 1\text{KB}$ (one request/instance), the client waits a long time for the reply, in some cases more than 1 minute. The reason why the request latency is much higher than the instance latency is the time a requests spends queuing at the leader before being proposed, which increases dramatically as the system reaches saturation. With small batch sizes, the wait time is very long. Increasing the size of the batches significantly decreases this wait time, reducing the client latency to values close to 1 second.

In this experimental scenario, the maximum throughput is reached with a batch size smaller than the one required for network-optimality (128KB versus 362KB, see last column of Table IV), and the maximum throughput is lower than the network maximum (250 versus 405 requests/s, computed using Formula (1)). This happens even though the CPU usage is low, at around 10%. The reason is that when the system executes only one instance at a time, *i.e.*, $w = 1$, then whenever a process is busy processing a message that is on the critical path (*i.e.*, serializing, deserializing, context switches, executing the protocol, etc), the network is idle. This can be avoided with pipelining.

2) Pipelining without batching: Here we look at the effects of pipelining without batching, *i.e.*, $BSZ = 1KB$ (one request/instance). The results are shown in Figure 9.

The throughput increases linearly with WND, up to a maximum of around 100 requests per second with $WND = 30$ (Figure 9d). Further increasing WND results in worse performance, with throughput dropping to around 60. This is reflected on the decision time (Figure 9b). Up to $WND = 30$, the decision time is stable at around 300ms (which is optimal as predicted by the analytical model, see Table IV). For $WND > 30$, the decision time jumps to between 500 to $900ms$, indicating that the system is overloaded. The additional delay is likely the result of network contention caused by a large number of small messages, as in this particular experiment the CPU usage never exceeds 10%.

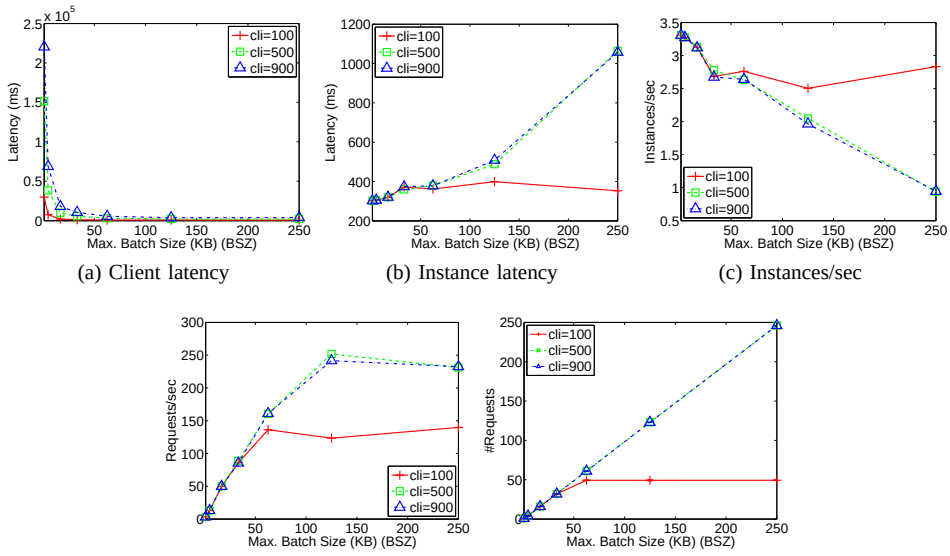
Once again, the system is not able to reach the expected number of parallel instances ($w = 182$) to fully utilize the bandwidth available. Hence, just like when using batching by itself, pipelining alone is not enough to reach the optimal performance in this scenario.

3) Combining batching with pipelining: For this experiment, we varied the maximum window size while keeping the maximum batch size at 128KB. We chose 128KB because it produced the best performance in the tests with batching alone, see Sect. IV-D1. Figure 10 shows the results. The system reaches the maximum throughput of more than 400 requests/s with $WND = 4$, see Figure 10d (compared to 405, the throughput given by (1)). As shown by Figures 10a to 10d, performance does not improve for higher values of WND. The reason can be understood by looking at the number of requests per instance (Figure 10e) and the number of active instances (Figure 10f): as WND increases, either both metrics remain constant (cli=900), or one increases as the other decreases (cli=500 and 100). In both case, the performance remains the same.

According to the analytical model, the optimal throughput is achieved with $w = 2$, but in the experiments the peak throughput is achieved only with $WND = 4$. With $WND = 2$ the throughput is 340 requests/sec, 60 below the theoretical maximum. To understand this difference, recall that the analytical model assumes that batches have

[3]http://www.netperf.org/netperf/

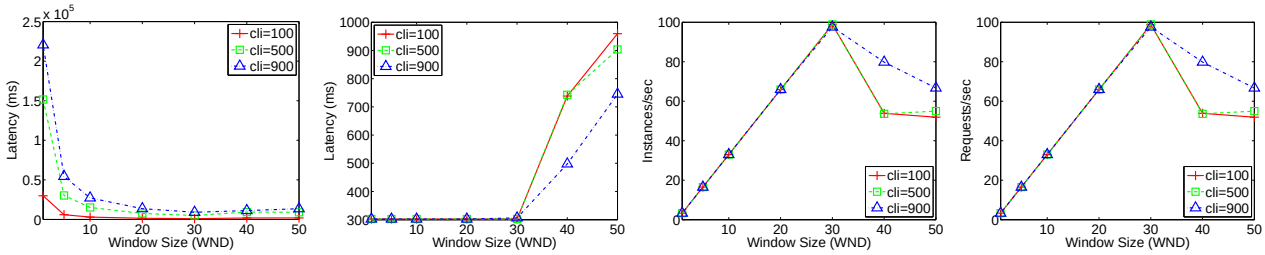
[4]This is the real speed of the physical network used to run the tests.



(a) Client latency (b) Instance latency (c) Instances/sec

(d) Requests/sec (e) # Requests per instance

Figure 8. Emulab: Batching without pipelining. $S_{req} = 1\text{KB}$, $WND = 1$



(a) Client latency (b) Instance latency (c) Instances/sec (d) Requests/sec

Figure 9. Emulab: pipelining without batching. $S_{req} = 1\text{KB}$, $BSZ = 1\text{KB}$

maximum size and the per-instance decision time is the one given by τ_{dec}. If these assumptions do not hold, then the model will underestimate w. In this case, the batches are indeed of maximum size for $WND = 2$ (Figure 10e), but the effective decision time (≈600ms) exceeds τ_{dec} (402ms, Table IV). This is likely caused by small queuing delays at the system level, both in the network and in the TCP stack of the nodes.

D. Cluster experiments

While in a WAN the bottleneck is typically the network, in a cluster the bottleneck is more likely to be the node's CPU since the network has usually high bandwidth and low latency. This changes significantly the relative advantages of batching and pipelining, as the experiments in this section show.

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

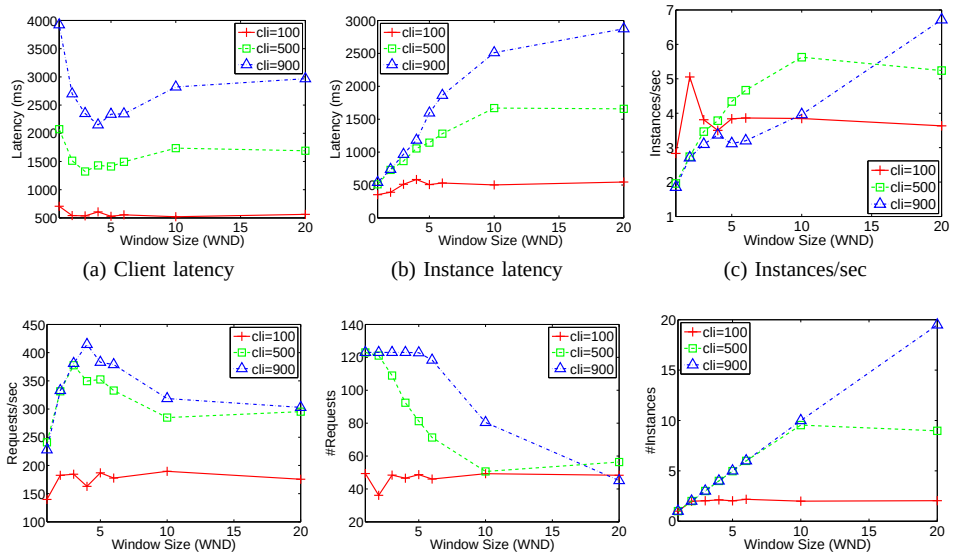
S_{batch}	1KB	47KB	64KB
$w = f(S_{batch})$	$\lceil 16.5 \rceil = 17$	1	$\lceil 0.93 \rceil = 1$
τ_{dec}	0.41 ms	0.8 ms	0.92 ms

Table V
FORMULA (4) FOR THE CLUSTER TOPOLOGY AND THE CORRESPONDING INSTANCE DURATION (τ_{dec}), ($n = 3, L = 0.2ms, B = 940\text{Mbit/s}, S_{req} = 1\text{KB}, S_{ans} = 1\text{KB}, S_{ack} = 0\text{KB}$)

Using the results in Section III, Table V shows several combinations of values for S_{batch} and w that maximize network utilization.

1) Batching without pipelining: For these experiments we set $WND = 1$ and varied BSZ. Figure 11 shows the results.

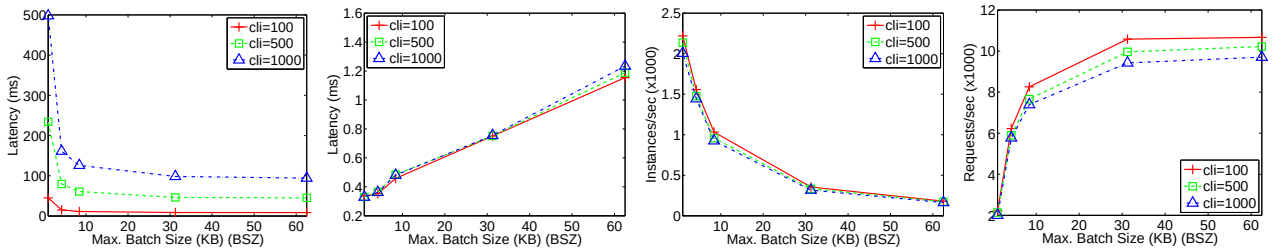
Increasing BSZ provides large gains in request throughput (Figure 11d). With $BSZ = 1\text{KB}$ the throughput is under 3000 requests per second. As BSZ increases, the throughput also increases up to a maximum of just under 11000 requests per second with $BSZ = 32\text{KB}$. Further increasing BSZ does not provide any gains. At this point, the bottleneck is the leader's CPU.



(a) Client latency (b) Instance latency (c) Instances/sec

(d) Requests/sec (e) # Requests per instance (f) # Active instances

Figure 10. Emulab: Batching and parallel instances. $S_{req} = 1\text{KB}$, $BSZ = 128\text{KB}$



(a) Client latency (b) Instance latency (c) Instances/sec (d) Requests/sec

Figure 11. Cluster: Batching without pipelining. $S_{req} = 1\text{KB}$, $WND = 1$

Looking at instance throughput (Figure 11c), we see the opposite effect, that is, the throughput decreases as the batch size increases. It reaches a maximum of 2800 with $BSZ = 1\text{KB}$ and drops to less than 500 for BSZ larger than 32KB. Up to $BSZ = 32\text{KB}$, the increase in the number of requests per instance more than offsets the drop in the number of instances executed, resulting in improved request throughput[5]. For $BSZ > 32\text{KB}$, the two tendencies balance out, and the throughput remains stable.

According to the analytical predictions, for $w = 1$ the maximum throughput is reached for batches of $47KB$. However, in the experiments the maximum throughout is achieved when the limit on the batch size is set to $BSZ = 32KB$. As we mentioned previously, the analytical model assumes that the network is the bottleneck in the system. This is not the case in this experiment, where we measured a CPU usage of close to 100% at the leader process for $BSZ \geq 32\text{KB}$. Therefore, the system is limited by the leader's CPU at 11K requests/sec, which prevents it from utilizing the full capacity of the network.

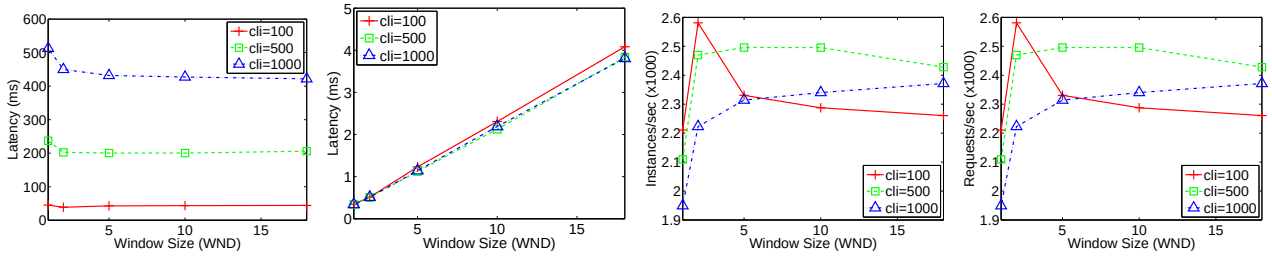
2) Pipelining without batching: For these experiments we set $BSZ = 1\text{KB}$ and varied WND. Figure 12 shows the results.

The throughput increases slightly when WND is increased from 1 to 2, but remains stable afterwards. The maximum request throughput is reached at 2500 requests/s, which is well below the maximum reached when using batching, and of the theoretical maximum of the network (≈40K requests/s, see Formula (1)). Once again, the bottleneck is the CPU of the nodes, in particular the CPU of the leader which in these tests was close to 100% utilization.

3) Combining batching with pipelining: For these tests, we set $BSZ = 64\text{KB}$ and varied WND. Figure 13 shows the results.

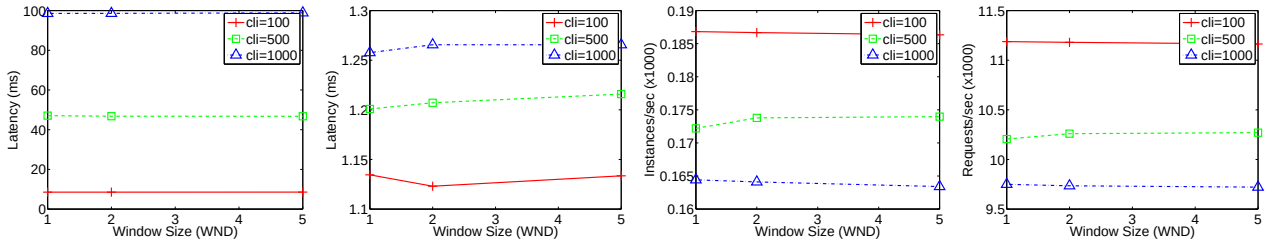
The maximum throughput is similar to the best result when using only batching, around 11K requests/s. In fact, Figure 13d shows clearly that in this scenario, executing

[5]In this experiment the batches were always of maximum size (not shown here). So the number of requests per instance increases linearly.



(a) Client latency (b) Instance latency (c) Instances/sec (d) Requests/sec

Figure 12. Cluster: pipelining without batching. $S_{req} = 1\text{KB}$, $BSZ = 1\text{KB}$



(a) Client latency (b) Instance latency (c) Instances/sec (d) Requests/sec

Figure 13. Cluster: Batching and pipelining combined. $S_{req} = 1\text{KB}$, $BSZ = 64\text{KB}$

parallel instances does not improve performance at all. Once again the reason is that the leader's CPU is at maximum usage, becoming the bottleneck of the system.

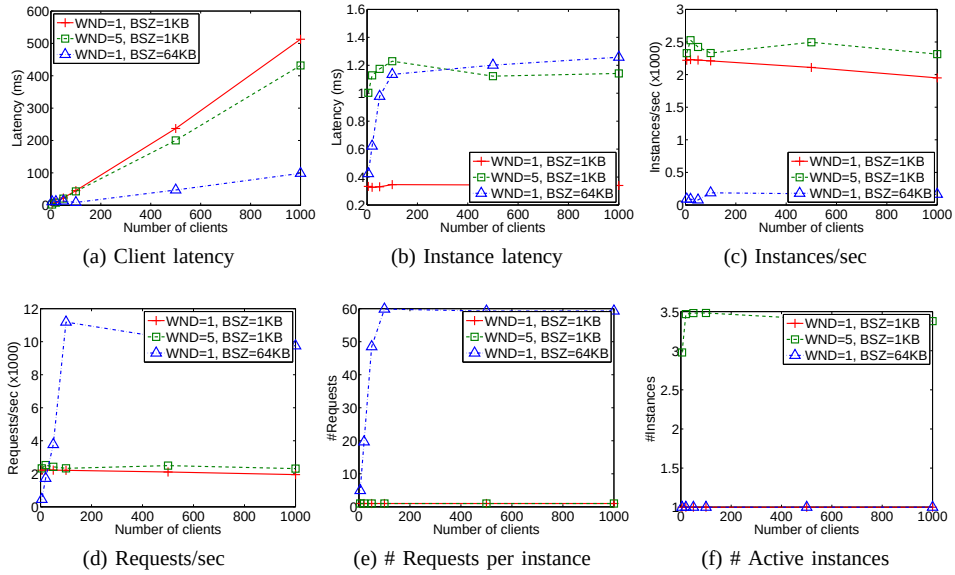
E. Scalability with Number of Clients

In this section we study how batching and pipelining settings affect the scalability with the number of clients. We chose three typical combinations for BSZ and WND ($\{1\text{KB}, 1\}$, $\{1\text{KB}, 5\}$ and $\{64\text{KB}, 1\}$), and vary the number of clients from 3 to 1000. We perform these experiments and all the others shown in the rest of the paper on the cluster, because in Emulab the system bottleneck is the network which prevent us from properly study any aspect of system performance other than the network itself. On the cluster the bottleneck is clearly the nodes CPU, as seen in the previous experiments. Figure 14 shows the results.

With $BSZ = 1KB$ the system is limited to a throughput of 2000 requests/sec, reached with 20 clients (Figure 14d). As the number of clients increases, the throughput remains similar to the peak of 2000 requests/sec. Increasing WND to 5 improves the throughput only slightly. Although the system is able to increase the parallelism from an average of 1 to 3.5 simultaneous instances (Figure 14f), the time per instance increases also by a factor of around 3 (Figure 14b), therefore negating most of the gains of the increased parallelism as can be seen by the instance throughput which is only slightly higher with $WND = 5$ (2.5 instead of 2, Figure 14c). In these tests, the bottleneck is the CPU of the leader, which is reached with just 20 clients.

With $BSZ = 64\text{KB}$, the greater efficiency of the larger batches allows the system to scale up to 100 clients, with a peak throughput of 11K requests/sec (Figure 14d). With less than 50 clients, the situation reverses, with the system performing worse with $BSZ = 64\text{KB}$ than with $BSZ = 1\text{KB}$. This is a consequence of the configuration of the batching policy chosen for the tests, which is not tuned for low load scenarios. The next section explores this topic in more detail.

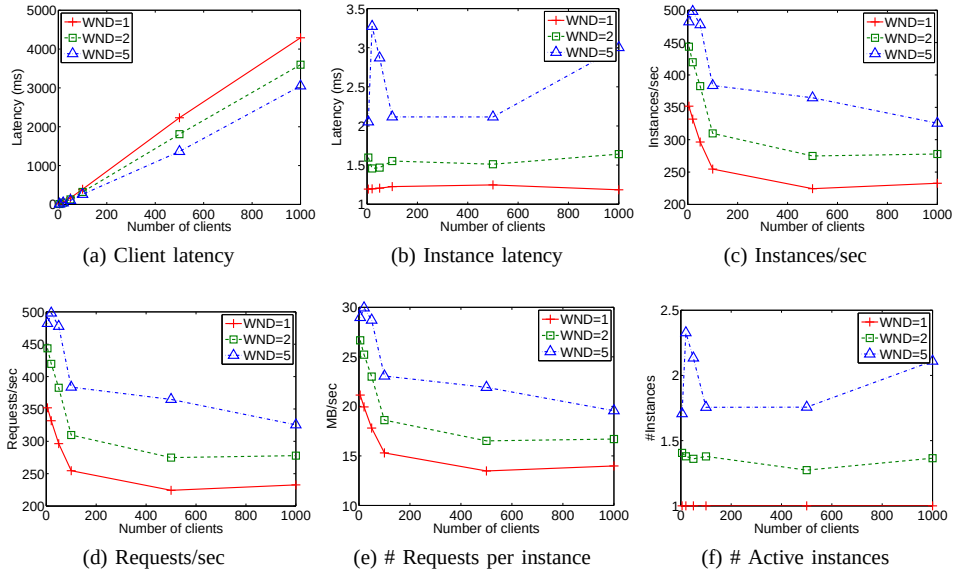
The consensus latency remains fairly constant between 0.35 and 1.2ms (Figure 14b). On the other hand, the latency experienced by the clients increases linearly with the number of clients (Figure 14a). This is caused by the leader holding the client requests on an input queue until it is able to include them on a batch. This delay is orders of magnitude larger than the consensus latency for higher number of client. The client latency is significantly lower for $BSZ = 64\text{KB}$ than for $BSZ = 1\text{KB}$. This is due to the higher throughput achieved with larger batch sizes, which significantly reduces the queuing time of requests. This suggests that for services expected to have moderate to high load, optimizing throughput will actually provide gains in latency, even if these same optimization increase the latency with low load. The additional increase with low load will likely be much smaller than the gains with high load. In the case of these experiments, a larger batch size increases the consensus latency from $0.35ms$ to $1.2ms$, but at high loads it decreases the latency perceived by the client from $500ms$ to less than $100ms$.



(a) Client latency (b) Instance latency (c) Instances/sec

(d) Requests/sec (e) # Requests per instance (f) # Active instances

Figure 14. Cluster: Scalability with number of clients. $S_{req} = 1\text{KB}$



(a) Client latency (b) Instance latency (c) Instances/sec

(d) Requests/sec (e) # Requests per instance (f) # Active instances

Figure 15. Cluster: Performance with large requests. $S_{req} = 64\text{KB}$, $BSZ = 64\text{KB}$

F. Large requests

So far, the throughput in the cluster experiments has been limited by the leader's CPU, which has always been the system bottleneck at 100% usage. The network, on the other hand, was far from the maximum: a maximum throughput of around 11K requests/sec corresponds to 33MB/sec on the outgoing link of the leader, since each request is sent to the two other replicas and the answer, which is the same size as the request, is sent to the client. But the capacity of the link was measured at 940Mbit/s, which corresponds to 117.5MB/sec, considerably higher than the 33MB/sec of effective usage.

In this section we look at the effect of the size of the client request in the throughput of the system. Each client request has a fixed processing cost. In the case of the previous cluster experiments, a request size of 1KB is relatively small for the bandwidth available. In fact, to maximize network utilization, the system should process 44K requests/sec, which represents a large amount of processing time devoted to the fixed per-request costs. To investigate the effect of the request size on throughput, we performed experiments with a 64KB requests. We fix $WND = 1$ so that each batch

contains a single request, and vary the limit on the number of parallel instances from 1 to 5. Figure 15 shows the results.

Figure 15e shows that with 64KB requests, the data rate is substantially higher than with 1KB requests, reaching a maximum of $30MB/s$ worth of requests, which corresponds to a total data rate of $90MB/s$ on the outgoing link of the leader. This peak of 30MB/s of ordered requests is reached with around 50 clients and then drops as the number of clients increases, stabilizing between 20 and 25MB/s. The reason is that the communication between the clients and the replicas is using the same network as replica-to-replica communication. As the number of clients increases, so does the total number of TCP connections competing for bandwidth over the same network, which leads to congestion and a degradation on throughput.

The results also show that increasing WND improves the throughput significantly, reaching a maximum with $WND = 5$ (Figure 15d). a maximum with $WND = 5$ (Figure 15d). The system is able to execute an average of 1.5 concurrent instances with $WND = 2$, and of 2 concurrent instances with $WND = 5$(See figure 15f) [6] . The additional concurrency increases the per-instance latency (See Figure 15b), but not enough to offset the gains of the increased parallelism. These results differ from the experiments with $S_{req} = 1KB$, where increasing WND did not improve performance at all. The reason for this different behavior is that a larger request size decreases the percentage of CPU time that the nodes must devote to the fixed costs of processing a request, therefore leaving more time for executing the ordering protocol, which can be used to execute several instances of the ordering protocol in parallel.

G. Batching Policy

So far all the experiments were performed with the parameter Δ_B (Figure 6) set to $10ms$. If the client load is high enough to allow the leader to fill up a new batch whenever $w < WND$, then this timeout never expires and its value has no impact in the results. This has been the case of most experiments presented so far. But as seen in subsection IV-E, when the load is low the performance can suffer significantly, as the leader delays new instances while waiting for additional client requests. We explore how Δ_B affects performance in low to moderate load scenarios with a set of experiments where this parameter is set to values ranging from 1ms to 50ms. We also tested a variation of the algorithm in Figure 6 that differs only in the following: if there are requests waiting and no instance is executing, then the leader starts a new instance immediately. Otherwise, if there is at least one instance in execution, it behaves like the basic algorithm. The rationale behind this policy is that if the system is idle, then starting a new instance, even if it is of a small size, will be a better use of the resources than to leave them idle.

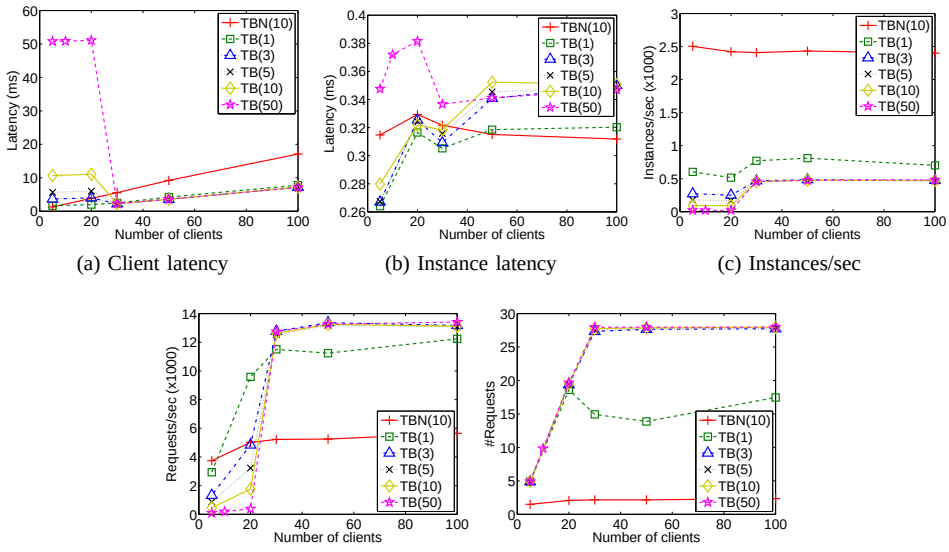
In the following, we denote the tests done with the unmodified algorithm (Figure 6) by Time-Based(x), or TB(x) for short, where x is the value of Δ_B, and the policy based on the variant of this algorithm by TBN(x). The N stand for Nagle because the basic idea of the modification is inspired by TCP's Nagle algorithm.

Figure 16 shows the results. With the exception of TB(1), all TB(x) policies exhibit a two-phase behavior. With less than 30 clients, the client latency is constant, being just above the timeout Δ_B (Figure 16a). With 30 clients, the latency drops to around 3ms, and then grows linearly with the number of clients. The other metrics reflect the same behavior. The cause for this dual behavior is that with less than 30 clients, the leader does not receive enough requests to fill up the batches, so it waits until the expiration of the timeout Δ_B before proposing a batch. Therefore, the performance is limited by the expiration of the timeout. On the other hand, with 30 or more clients the leader is able to fill up the batches and propose them before Δ_B expires, so the service performs at the speed of the system.

TB(1) is a special case. With less than 30 clients, it performs better than the other TB policies, but it does worse with more than 30 clients. The Figures 16c and 16e show that while the instance throughput does increase with TB(1), the average size of each instance is lower than with the other policies, which results in lower throughput. Recall that with TB(1), the leader starts an instance at least every 1 ms, so it has only 1 ms to receive client requests, which in this particular system is not long enough to receive enough requests to fill a batch.

The $TBN(10)$ is the best performer with 5 clients, but it is significantly worse for any higher loads (Figure 16d). In particular, with 30 or more clients, the request throughput of $TBN(10)$ is only 5K compared to 13K for the best performing policies. The reason is that $TBN(10)$ produces very small batches, with an average size of 3 as compared to 29 for most of the $TB(x)$ policies. Therefore, even though it achieves the highest instance throughput (2500 versus 500), it has a very low request throughput. The results would be similar with other timeouts for TBN. Recall that the leader is allowed to start an instance as soon as no instance is running. Since each instance takes $0.35ms$ to execute, with TBN the leader starts an instance every 0.35 ms. Therefore, any timeout larger than this value will never expire. The reason for the poor performance of $TBN(x)$ is likely because the leader prioritizes executing consensus, leaving little time to receive and process client requests. The results above were taken on single core machines. Since the protocol stack used for the experiments uses separate threads to execute the ordering protocol and to process client requests, it is likely that on a multi-core machine the results

[6] We omit the results with higher values for WND, which did not improve performance.



(a) Client latency (b) Instance latency (c) Instances/sec

(d) Requests/sec (e) # Requests per instance

Figure 16. Cluster: Batching delays. $S_{req} = 128B$, $BSZ = 4\text{KB}$, $WND = 2$

of the TBN policy would be substantially better.

H. TCP versus UDP

The Paxos protocol is designed to tolerate message loss, so it does not need reliable. Nevertheless, so far all experiments were done with TCP. It is reasonable to expect that this additional reliability comes at a cost of performance as compared to using an unreliable communication mechanism, like UDP. In this section, we look at this question, comparing our Paxos stack using TCP and UDP.

Figure 17 shows the results. With UDP, the average instance latency is smaller than with TCP by 0.05ms (Figure 17b), which is to be expected since UDP has less per-packet overhead. On the other hand, with UDP the instance throughput (Figure 17c) is worse by around 30 instances per second, while the size of the batches is more or less the same (Figure 17e). The overall result is that UDP has a worst request throughput than TCP, by around 1K requests/sec. Even though the difference in percentage is small, these results are somehow unexpected. For some reason we were not able to fully explain, TCP is more efficient and executes a higher number of instances, even though the time per instance is greater. One possible justification that we might explore in the future is that the difference in performance comes from the fact that when using TCP, our implementation reuses buffers between messages, while with UDP it creates new buffers for each message.

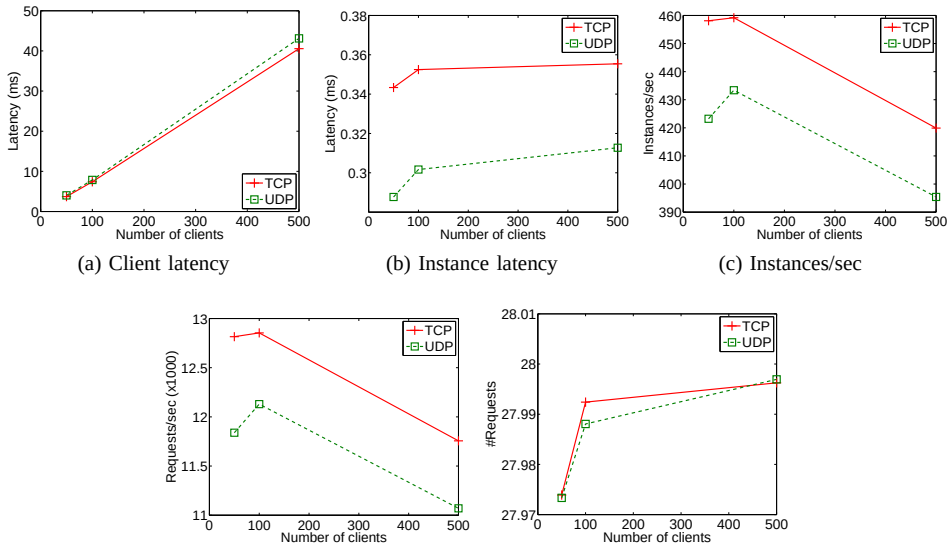
V. Discussion

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. Batching really shines in high load scenarios, when the leader is able to quickly form large batches: the gains in throughput can be of one to two orders of magnitude. When the load is low to moderate, batching must be carefully configured to strike a good balance between batch size and latency, in order to avoid excessive delays as the leader waits for additional requests to fill a batch.

Pipelining only provides benefits 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 the network, the greater the potential for gains of executing additional parallel instances. Thus, in general, it will provide minimal performance gains over batching alone in low latency networks, but may provide 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 cluster experiments). This problem can be avoided by carefully setting the limit on the number of parallel instances, taking in consideration the available CPU time on the leader.

The paper has focused mainly on throughput because as long as latency is kept within an acceptable range, optimizing throughput provides greater gains in performance. A system tuned for high-throughput will have higher capacity,



(a) Client latency (b) Instance latency (c) Instances/sec

(d) Requests/sec (e) # Requests per instance

Figure 17. Cluster: TCP versus UDP. $S_{req} = 128B$, $BSZ = 4$KB, $WND = 1$

therefore being able to serve a higher number of clients with an acceptable latency, whereas a system tuned for latency will usually reach congestion earlier, at which point it stops providing an acceptable service. This effect is clearly seen in the results in Section IV, where large batch sizes clearly provide better performance in terms of throughput and latency when the system is under high load. Nevertheless, for system where latency is more important than throughput, we can use the analytical derivation above to compute the latency of an instance (Formula (3)).

These results are also relevant to the discussion of monolithic (*e.g.*, Paxos) versus modular implementations of replicated state machines. A typical modular implementation uses atomic broadcast as a black box to order the client requests. This leads to better modularity, as it abstracts the implementation of atomic broadcast, but it may make it harder to implement pipelining. As an example, a common way of implementing atomic broadcast is by reduction to consensus [10], where consensus is used as a black box. Batching can be easily implemented in such a scenario, but the same is not true for pipelining. This optimization is simple when there is a leader responsible for choosing an order, like in Paxos, but rather hard if we make no additional assumptions on the internal working of the consensus box. Therefore, we can use the results obtained with batching without parallel instances an example of what to expect from atomic broadcast by reduction to consensus. This suggests that this alternative implementation of replication will perform well in networks with low latency, where batching is enough to achieve optimal throughput, but will under-perform when the latency is increased.

VI. 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 [11] 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 [12]. The work in this paper looks at these two optimizations in the context of state machine replication protocols, studying how to adapt them and 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.

Batching and pipelining optimizations in state machine replication: In [5], 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 [6] proposes an adaptive batching policy also for group communication systems. In both cases the authors look only at batching. In this paper, we have shown 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 [2] and [3]. In [2] 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 [3], where the authors propose two adaptive batching policies. One policy uses a common code path on the source code of the server that must be traversed by messages before they are added to a batch. This policy sends a batch if there is no additional message inside the common code path. The other policy uses a timer that is reset whenever the application asks to send a message. The batch is sent when the timer expires. 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 [13], which describes an implementation of Paxos that uses batching to minimize the overhead of stable storage.

Batching is especially important in Byzantine systems. These protocols are more expensive than the corresponding protocols for benign faults, since they have a higher message cost per consensus instance and sometimes rely on cryptographic operations which have a high fixed per-message cost. Two examples are PBFT [14] and Zyzzyva [15], both of which use batching and pipelining. The corresponding publications contain experimental studies that, among other factors, evaluate the effects of batching. But these studies have a limited scope, focusing only on a narrow range of settings and ignoring the interplay with pipelining.

Other optimizations: There has been a lot of work on other optimizations for improving the performance of Paxos-based protocols. Two of the most recent works are LCR and Ring Paxos. LCR [16] is an atomic broadcast protocol based on a ring topology and vector clocks that is optimized for high throughput. In tests, it is able to use up to 95% of the bandwidth of a Gigabit Ethernet. Ring Paxos [7] is a variant of the Paxos protocol, that combines several techniques to improve performance. It uses IP multicast to send messages to all and organizes replicas on a ring to maximize the utilization of the network links. Additionally, it sends the Phase 2a message only to $f+1$ replicas instead of sending to all. Their prototype achieves a 90% bandwidth utilization on a Gigabit Ethernet. 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 our work applies both to WAN and LAN environments. In particular, pipelining is a especially effective technique in medium to high-latency networks, so it is important to understand its behavior.

VII. Conclusion

In this paper we have studied two important optimizations to Paxos, batching and pipelining. We have shown that batching produces the largest gains, both in a cluster and a WAN environment. Together with its simplicity, these results suggest that batching should be the first optimization considered in such a system. Interestingly, in systems with moderate to high network latency, batching by itself is no longer enough to achieve the best throughput. In this case, the use of pipelining provides a significant improvement in performance. The results show that as the network latency increases, the gains of pipelining become more significant.

Acknowledgment

The authors would like to thank Martin Hutle and Zarco Milosevic for their comments on an early draft of this paper, and Pawel Wojciechowski, Jan Konczak and Tomasz Zurkowski for their work on JPaxos.

References

[1] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, May 1998.

[2] B. Carmeli, G. Gershinsky, A. Harpaz, N. Naaman, H. Nelken, J. Satran, and P. Vortman, "High throughput reliable message dissemination," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, NY, USA, 2004.

[3] R. Friedman and E. Hadad, "Adaptive batching for replicated servers," in *Symposium on Reliable Distributed Systems, SRDS'06*, Oct. 2006.

[4] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, Dec. 2004.

[5] R. Friedman and R. Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," Department of Computer Science, Cornell University, Tech. Rep. TR95-1527, 1995.

[6] A. Bartoli, C. Calabrese, M. Prica, E. Di Muro, and A. Montresor, "Adaptive message packing for group communication systems," in *OTM 2003 Workshops*, ser. LNCS. Springer, 2003.

[7] P. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring paxos: A high-throughput atomic broadcast protocol," in *Dependable Systems and Networks (DSN'10)*, Jun. 2010.

[8] N. Santos, J. Konczak, T. Zurkowski, P. Wojciechowski, and A. Schiper, "Jpaxos - state machine replication in java," EPFL, Tech. Rep. to appear.

[9] B. White and J. L. 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.

[10] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[11] J. Nagle, "Congestion control in ip/tcp internetworks," IETF, Tech. Rep. RFC 896, Jan. 1984.

[12] V. N. Padmanabhan and J. C. Mogul, “Improving http latency,” *Computer Networks and ISDN Systems*, vol. 28, no. 1-2, 1995.

[13] Y. Amir and J. Kirsch, “Paxos for system builders,” Johns Hopkins University, Tech. Rep. CNDS-2008-2, 2008.

[14] M. Castro, “Practical byzantine fault tolerance,” Ph.D. dissertation, Laboratory for Computer Science, MIT, 2001.

[15] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: speculative byzantine fault tolerance,” in *Proceedings of twenty-first ACM SIGOPS SOSP*, NY, USA, 2007.

[16] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, “Throughput optimal total order broadcast for cluster environments,” *ACM Trans. Comput. Syst.*, vol. 28, no. 2, 2010.