LibPaxos Performance Analysis

[SP08: Lab Project]

Marco Primi
University of Lugano
Lugano, Switzerland
marco.primi@lu.unisi.ch

ABSTRACT

In this experiment we analyze the performance of a C implementation of the Paxos algorithm for distributed consensus. Through a series of targeted benchmark we try to gain insight on the dynamical behavior of the system. Our analysis will test different aspects such as bottlenecks and configuration options that will hopefully help us improve performance.

1. MOTIVATION

Paxos[4] is a distributed algorithm for consensus[1], it can be used to provide a total order broadcast primitive to a network of processes. It tolerates failures and message loss while delivering values at high rates.
This algorithm is extremely useful for maintaining consistency (for example among servers or among nodes of a distributed database), and it's fundamental to core techniques in fault tolerance, such as state machine replication.
For this reason we put our implementation, *libpaxos*[2], on the test bench. Our objective is the *understanding* of the performances, since "You cannot control what you cannot measure"[3].

2. PAXOS IN A NUTSHELL

We provide a brief overview of the Paxos algorithm. This explanation can give an idea of the dynamics of execution but it's for sure not enough to understand why the protocol works and how does it provide such strong properties with such weak assumptions.

[1] `http://en.wikipedia.org/wiki/Consensus_(computer_science)`
[2] `http://libpaxos.sourceforge.net`
[3] Tom DeMarco (Software Engineer)

Actors

There are different actors in a Paxos network:
Clients are using Paxos as a reliable communication medium. Some clients propose values to be delivered, some clients listen to that broadcast of values. A client may do both.
Learners are "passive" entities, they monitor the acceptor's *learn* messages and, when a value is unambiguously *decided* for an instance, they can deliver that value to a client waiting on top. Paxos ensures that eventually all Learners will deliver the same values in the same order (so called Atomic Broadcast[4]).
Acceptors processes wait for *prepare* and *accept* messages from the leader and, based on their status, may reply with a *promise* or send a *learn* to the learners. For a Paxos network to be fault-tolerant, acceptors must be able to recover after a crash. For this reason each acceptor synchronously writes his status to disk before answering any message.
Proposers receive values from clients. To ensure progress of execution, a *Leader Election* service is required by Paxos to agree on a single Leader-Proposer. Non-Leader Proposers just forward client values to the current leader.
Leader-Proposer collects client values and submits them in two phases. During the first phase it sends *prepare* messages to acceptors and wait for a majority of *promises*. Once the first phase is completed, the instance is declared *ready*. During phase 2, the leader sends an *accept* message to the acceptors, containing a client value. Proposers know the result of this operations since they are learners too (or use an internal learner).

As a performance optimization, a number of phase 1 instances are pre-executed by the leader, i.e. before the clients start to submit values we can pre-execute phase 1 for instances from 1 to 100. When a client value is submitted, it can be delivered in only two message delays (an *accept* and a *learn*).
In principle, also phase 2 could be executed in parallel to achieve better throughput. Since our application has some strict FIFO requirements, we opt-in for a simpler version that does not implement this optimization.

Example execution

n acceptor processes are started somewhere, clients A and B start as proposers, clients X and Z start as learner. Through the leader election service, A's proposer is nominated leader. The leader pre-executes phase 1 for instances from 1 to 100:

[4] http://en.wikipedia.org/wiki/Atomic_broadcast

it can send a single *prepare* message requesting a *promise* from the acceptors. When a majority of acceptors acknowledge that request, the leader knows that instances from 1 to 100 are ready. If clients are not submitting values, all the actors are idle.
At some point the clients A and B start submitting values v and v' to their respective proposers. As soon as the leader receives v from A or v' from B's proposer, it can start phase 2, it picks a *ready* instance and send an *accept*.
The acceptors receiving such request decide to accept it, so they send a *learn* message to all learners. When a learner sees that a majority of acceptors accepted the same value for instance number 1, it is sure that the value is *chosen*. Learners deliver value v to clients X and Z.
Since each proposer is a learner too, the leader knows that instance 1 was closed successfully. It can now start phase 2 for instance 2, v' will be eventually delivered to X and Z.

3. LITERATURE REVIEW

Paxos has been subject of much studies, the author himself has been working on different version of the algorithm, like Fast Paxos[5], and Cheap Paxos[6].
Google is using Paxos within Chubby[1], they also published a very interesting engineering perspective [2] describing their design choices and various implementation-specific optimizations.
There are some developed methodologies for performance analysis of distributed system that we find interesting, like [9], [8] and [3].

4. HYPOTHESES AND EXPERIMENTAL METHODOLOGY

In the following section we present the research questions we want to answer and the corresponding experiment we intend to perform.

Up to which point does the system scale?
Submitting more values increases throughput, however at some point the system will not be able to deliver yet more values per second. We can easily identify that limit by observing the delay between the proposal of a value and it's delivery.
By building a throughput versus latency graph we want to verify up to which point the system scales linearly. Discovering this upper limit will also be useful when choosing parameters for the next experiments.
Setup: We build a special client that proposes values at a given rate, we will increase this until we hit a limit. The client can also keep track of turnaround time of submitted values.

How does the size of values affect the throughput and latency?
Assuming that a client submits values at some fixed rate, we want to know how the size of those values has impact on the throughput.
Setup: We fix a benchmark, for example 100k values and proposal rate of 3000 vps (the rate is under the limit discovered previously). We measure the time required to complete this task varying the size of values proposed.

How much writing to disk slows down the system?
Some of the actors of a Paxos network require stable storage, *libpaxos* uses Berkeley DB[5] for such purpose. Since this is likely to be a critical factor for the overall performance, we test all possible access methods available in BDB: *BTree* (current implementation), *Queue* and *Recno* (reccomended for integer keys), and finally no disk synchronization at all.
Setup: We fix a benchmark (number of values, proposal rate) and we measure the time required to complete with different BDB settings. The values size will vary randomly but using a fixed random seed will always produce the same workload.

What's the cost of adding more acceptors?
In a Paxos network, we can tolerate $n/2 - 1$ failures, where n is the number of *Acceptor* processes. If a majority of them fails, the broadcast of values halts until some process recovers. We want to assess the cost of having more acceptors.
Setup: We fix a benchmark (number of values, proposal rate, values size) and we measure the time penalty of having more acceptors.

What's the cost of adding more learners?
In a Paxos network, learners are impersonated by clients subscribed to the broadcast. We want to assess the cost of having more learners.
Setup: We fix a benchmark (number of values, proposal rate, values size) and we measure the time penalty of having more learners.

Can we do better?
Using what we learned so far, we try to change some other library parameter (which was fixed in the previous cases), for example we lower timeouts or increase the limit on memory usage. By tuning those parameters, we hope to achieve better overall performances.
Setup: We design at least two benchmarks, for example heavy/light load, small/large/mixed packets or combinations of those. We try to change some parameter and compare the result with respect to the default settings.

5. INFRASTRUCTURE

All the tests will be performed within an Apple cluster comprised by 16 nodes connected trough a GigaBit ethernet switch. Each node is a RackMac3,1 with two PowerPC G5, 2.3GHz CPUs.
For the experiments described above, we will run each process on a different node of the cluster (i.e. a node is a proposer, 5 nodes are acceptors, 5 nodes are learners).
The client and a single proposer run on the same machine (the proposer is a thread started by the client).
All messages are relied trough UDP multicast with MTU of around 9kb.

6. EXPERIMENTS

We perform 3 kinds of experiment: *Measurements* to help understanding of the system it as it is, *Cost assessments* to quantify the impact of some external variable on which we have no control and *Improvements* to get better performances by varying some inner library parameter.
For those experiments we make use of the following metrics:

[5] `http://www.oracle.com/technology/products/berkeley-db/index.html`

Time to complete

Given a fixed set of values, we measure the time taken from when the first is submitted to when the last is delivered on the learner running inside the proposer. The granularity of those measurements is seconds.
In this benchmarks the clients is able to auto-regulate his proposal rate: if the queue of values in the proposer becomes too long, the client will temporarily stop submitting new ones.
Despite the reproducibility of the values proposed, even when the sizes are randomly chosen, there is variability in the results. We always present the average of at least 5 execution.

Delivery Rate

The average delivery rate can be measured as the number of values delivered divided by the time taken. If we look at the network as a queuing system, we imagine that we need to stay under this threshold to guarantee timely delivery of values.
The rate alone can be misleading since it does not consider the *quantity* of information delivered. It is however a good comparison metric for two benchmarks using the same values workload.

Throughput

The quantity of data delivered per second can be estimated by multiplying the delivery rate by the average value size. The size of values submitted depends on the application, it is therefore not always possible to maximize throughput.

Latency

We use a non-standard idea of latency, defined as *the time take to deliver* $r+1$ *values*, where r is a fixed proposal rate. To understand why this is better than measuring the time taken by a *single* value, we must explain how we perform the measurement.
Since the latency is quantified at a given delivery rate, we create a special client running on top of the proposer. This client takes a rate r as argument and once a second will submit r values to the proposer. Periodically (i.e. every two seconds), the client submits another value which contains an unique identifier together with a timestamp. We call this message a *probe*.
On the other "side" of the network, on top of a learner we create another client that receives the values delivered. This client is only interested in probes, for those it computes the time difference and log it for later analysis.
To avoid complex time-synchronization across cluster nodes, the client/proposer and the client/learner will run on the same machine. Notice that since there's no direct communication between them, this does not represent a bias, in fact it's a disadvantage since the two have to share CPU and network interface.
If we submitted the fixed-size probe *before* the r values, the time taken to deliver it would not change at all with a higher proposal rate or bigger values (assuming that the proposer is able to deliver all values within a second).
Enqueuing the probe after the r values does not return the absolute time required to deliver it, rather the time to deliver the previous r values.
This is much more useful when for example using big values and it's still a reliable metric for systems comparison.

The time taken to deliver a single value can be still estimated as $Latency/r$, i.e. if the proposer sends 2000 values per second and the latency is 240000μs, on average a value is delivered in 120μs.

Ex.1 - Current performance

We start with a really simple test to get an idea of the rate at which values can be delivered. We create a client that submits a fixed number of values to the proposer and we measure the time it takes to have all of them delivered to a learner.
For this experiment, the network is composed by: 1 Proposer (Client), 3 Acceptors, 1 Learner (within the proposer).
All Paxos parameters are set to default, Table 1 summarizes the relevant parameters for the client that proposes. The results for this experiments are presented in Table 2.

Name	**Value**	**Comment**
# of Val.	100000	Number of values proposed
Val. size	Random	Reproducible sequence of mixed-size values
Min Val. size	30 byte	Minimum size of a value proposed
Max Val. Size	3 kbyte	Maximum size of a value proposed
Wait interval	10 ms	Interval for submitting next batch of values
Max queue	300	Threshold after which the client proposer stops submitting values for a while

Table 1: (Ex.1) Client/proposer parameters.

The system takes around 30 seconds to deliver 100k mixed-size values. We observe quite some variability in the measurements despite the fact that the sequence of values is deterministic.

Run	Duration (sec)	Rate (vps)
1	37	2702.7
2	43	2325.5
3	38	2631.5
4	41	2439.0
5	40	2500.0
6	40	2500.0
7	38	2631.5
8	37	2702.7
9	36	2777.7
10	37	2702.7
Average	38.7	2583.9

Table 2: (Ex.1) Execution time and delivery rates in values per seconds.

Ex.2 - Impact of instrumentation

Measuring the system by only means of time to complete a workload is probably too simplistic. There are different runtime events that may reveal a lot about what's going on in the network, for example the number of times the proposer had to restart from phase 1 because a timeout occurred.
For this reason, we decide to lightly instrument our code,

we create a set of macros to be placed in critical areas of the learner and proposer to count specific events.
We repeat the previous experiment with event counting enabled, this will be useful for both assessing the impact of our instrumentation and for getting an idea of the counter values for future comparison.

Table 3 shows the results for this experiment. More details on the meaning of each counter is presented in Appendix A. Two interesting fact emerge from this series of test: the first one is that the system is performing as expected, the few discrepancies can be traced back to packet loss.
The second fact is that when run the experiment different times, we get better values after the fifth run. This is unexpected as every time we restart completely.
We also conclude using a t-test[6] that the instrumentation does not affect the performance in a significant way.

Ex.3 - Scalability

So far our measurements focused on execution time for a fixed-size workload, this tells us just the average rate at which values are delivered. An other essential aspect of the system is the time it takes for a value to be propagated in the network.
In this experiment we measure the time interval from whena set of a values are proposed by a client to when those values are learned by another client, as previously described in the Latency section. In different runs we gradually increase the proposal rate up to the point in which we notice an exponential growth of response time.

Table 4 and Figure 1 present a summary of the results

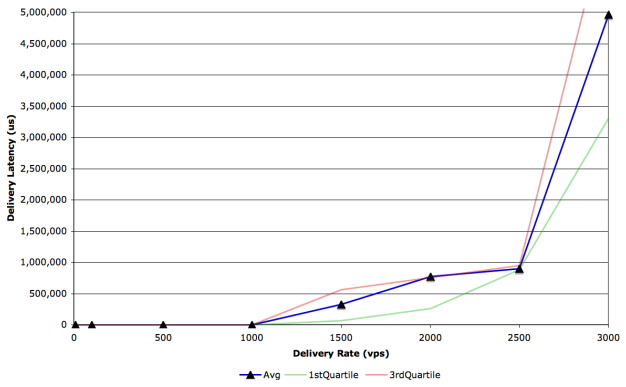

Figure 1: (Ex.3) Exponential increase of latency when submitting more than 2500 values per second.

collected. The system seems to be delivering with delays of less than a millisecond if the proposer sends less than 1000 values per second. From 1000 to 2500 values per seconds, the delay increases, but the system is still able to deliver in less than a second.
The system is clearly not able to digest more than 2500 values per second. When we try to push 3000 the delay of probes increases continuously, moreover we see the queue of client values in the proposer growing bigger and bigger, so we are sure we hit a limit.
Notice that those latency times are aggregated, if the latency

[6]With 95% confidence, the average difference of those runs w.r.t the previous is across zero

Table 3: (Ex.2) Event counters, measured, average and expected.

Run	1	2	3	4	5	6	7	8	9	10	Avg	Expected
seconds	40	39	42	38	37	37	37	37	36	36	37.9	-
Proposer:												
send_prepare	4313	4108	3869	3948	3847	3847	3847	3847	3847	3856	3933	4000
handle_promise	13015	12726	11264	10792	10251	10492	10528	10173	10133	10457	10983	12000
p1_timedout	535	259	43	99	0	0	0	0	0	8	94	0
send_accept	100008	100003	100005	100003	100000	100000	100000	100000	100000	100002	100002	10000
delivering	100008	100005	100007	100005	100003	100003	100003	100003	100003	100003	100004	100000
verify_val	100005	100002	100004	100002	100000	100000	100000	100000	100000	100000	100001	100000
delivering_reserved	3	1	1	1	0	0	0	0	0	1	1	0
instance_stolen	0	0	0	0	0	0	0	0	0	0	0	0
p2_timedout	5	2	4	2	0	0	0	0	0	1	1	0
Learner												
send_lsync	0	0	0	0	0	0	0	0	0	0	0	0
handle_learn	288987	296926	293938	282158	271770	277215	277730	268693	267891	274638	279995	300000
total_holes	0	0	0	0	0	0	0	0	0	0	0	0

Rate (vps)	Min	Max (ms)	Avg
10	0.4	0.5	0.5
100	0.4	0.5	0.5
500	0.3	0.6	0.4
1000	0.3	0.4	0.3
1500	30.3	637.6	328.1
2000	209.0	3364.2	776.2
2500	436.3	1393.9	898.3
3000	1284.2	8368.4	4967.7

Table 4: (Ex.3) Delivery latency with different delivery rates.

is 0.3 ms at 1000 vps, the time to deliver a single value is around 0.3μs on average.

Experiment 4 - Permanent Storage

In all the previous scenarios, we modeled a somehow *ideal* condition for the network, that is we assume no process failure. In real world situation however, crashes cannot be ignored.
To ensure *safety* despite failures, the acceptors in the network must log their state information to disk before answering to any request. In this way if an acceptor crashes and then recovers, it can restore it's state, thus ensuring consistency.
For each instance, the acceptor maintain a key-value pair in the form:

$$< K,\ V > = < Id,\ (Hb, Val, Vb) >$$

where *Id* is the instance number (int), *Hb* holds the highest ballot to which we promised (int), *Val* is a value if we accepted one (char[])), and *Vb* is the ballot corresponding to that value (int).
When configuring the database, we must consider that (i) the structure is dynamic but within a fixed limit (UDP message size), and *(ii)* the structure is essentially identified by an unique integer number.

In our library, this disk synchronization is implemented using Berkeley DB, a high-performance embedded database[7].
In this experiment we do the following:
1. Try different BSB *access methods* to ensure we are using optimal settings. More details on the BDB options are presented in Appendix B.
2. Turn on disk logging and assess the penalty *w.r.t.* the previous measurements.
With the same settings as before and disk synchronization still turned off, we measure the time taken to deliver a fixed number of values and then run latency benchmarks. The data collected is presented in Table 5.
In all delivery rate tests, Hash and Queue seems to always take a few seconds more than the worst execution times of Btree and Recno. We focus on those.
Since execution times for Btree and Recno are similar, we use a t-test to confirm that the latter is effectively doing a better job[7].

[7]With 95% confidence, the average of the differences is above zero

	BTREE	RECNO	HASH	QUEUE
Time (s)	39.5	37.0	46.7	50.3
Rate (vps)	2531.6	2702.7	2142.9	1986.8
Lat, 1kvps (μs)	326	139,043	431,135	284,015
Lat, 2kvps (μs)	744,345	240,010	1,798,907	4,840,303

Table 5: (Ex.4) Average delivery time for 100k variable-size values, average delivery latency (at 1000 and 2000 values per seconds) for different BDB access methods. Each value is the average of 5 executions.

Also observing latency, Recno seems better, notice how the measurement similar at 1 and 2 kvps (as opposite to all other methods). For the 1kvps Recno is actually able to deliver values in less than a millisecond (like Btree), but this happens only after a few seconds of warmup, so the average is higher.

We discover that turning on disk synchronization has a *catastrophic* effect on performances. Using the previously mentioned event counters we calibrate the timeouts for messages so that acceptors have the time to complete I/O operations. Timeouts for phase 1 and 2 of the protocol were previously set to 10ms, now we need to increase them to respectively 2500ms and 250ms.
The results for this benchmark are presented in Table 6. With this configuration we cannot push more than a mere 8 values per second! To keep the delivery latency under a second, we must stay under that limit.
It is clear that disk synchronization is a bottleneck in our system, if we want to get around this issue we probably need to find a better way to handle I/O. For the rest of this document we focus on tuning of the non crash-tolerant acceptors.

	RECNO	BTREE	
Delivery rate			
1	60	59	s
2	63	59	s
3	62	60	s
4	60	59	s
5	62	60	s
Avg	61.4	59.4	s
Rate	8.1	8.4	vps
Latency at 5vps			
1	698,632	566,636	μs
2	596,725	663,136	μs
Avg	647,678	614,886	μs

Table 6: (Ex.4) Delivery time for 500 values and average latency at 5 values per second. Fault-tolerant configuration.

Experiment 5 - Fault tolerance

In a Paxos network with n acceptors, we can guarantee progress as long as a *majority* of them is alive. In other words we tolerate f failures, where $f = (n/2) - 1$.
To be *more* fault-tolerant we may decide to add more acceptors. This has of course a cost, for example with 3 acceptors an instance in phase 1 is declared ready when 2 promises are

received, with 4 or 5 acceptors instead we need to wait for 3 promises.
Increasing n has a direct impact on both network traffic generated by the acceptors and computation required by proposer and learners. In this experiment we try to measure this impact.
The network is composed by a proposer, a learner and a variable number of acceptors. The rest of the parameters are the same as the previous experiments (i.e. Ex. 1, 2, 3). Figure 2 clearly shows that the delivery rate decreases

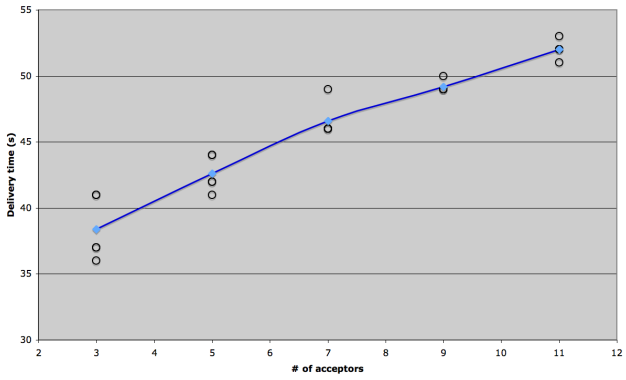

Figure 2: (Ex.5) Linear growth of completion time to deliver 100k values when adding more acceptors.

linearly with respect to the number of acceptors and consequently f.
This is quite good, it means that up to the point where we measured, we were not able to overload the system. This is probably strictly related to the capacity of the network and in fact extracting an estimate of the package loss rate confirms that the network is still able to deal with the load.
The package loss rate (in Table 7 along with average completion time and delivery rate) is measured on the number of learn messages received by the learner (inside the proposer). If the proposer sent i accept messages, the learner should receive $n * i$ learn messages.

n	3	5	7	9	11
f	1	2	3	4	5
Avg Time (s)	38.4	42.6	46.6	49.2	52
Delivery rate (vps)	2604	2347	2146	2033	1923
Packet loss	3.5%	5.5%	7.9%	7.4%	8.0%

Table 7: (Ex.5) Time to deliver 100k values when adding more acceptors and corresponding packet loss rate estimate.

Experiment 6 - Cost of Learners

While the number of acceptors in a network is a design decision, the number of learners is more likely to be an external constraints, i.e. the number of database replicas or number of clients subscribed to the broadcast.
From an abstract point of view, adding more learners comes for free, the acceptor still sends a single learn message when accepting a value. However, at lower level, this message has to be switched to multiple hosts thus increasing network usage.
In the worst case the package loss rate grows and the learners don't receive all learn message. So they start to send synchronization requests to the acceptors and the situation gets worst.
Figure 3 depicts the time taken to push 100k mixed-size

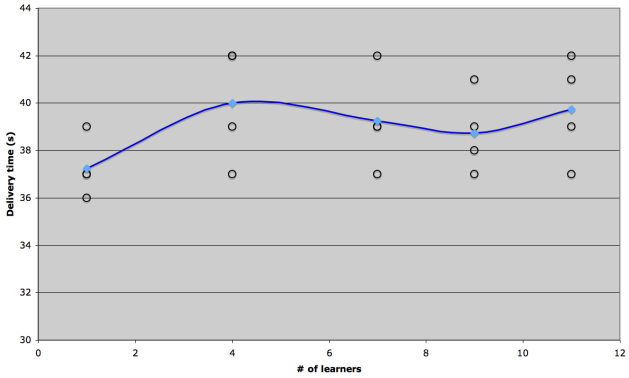

Figure 3: (Ex.6) The number of acceptors does not seem to affect the time taken to deliver 100k values.

values with a varying number of learners. Although the variance of values is quite high, the change don't seem to affect the outcome very much. Looking at latency measurements (Table 8) is even more surprising: the system seems to work *better* with more learners. Which is a little counterintuitive. The network is not *immune* from the change, in fact to get those results we have to increase the timeout parameters in the proposers since most of the phase 1 were expiring. We

Delivery Rate		
Learners	(s)	(vps)
1+0	37.3	2685
1+3	40.0	2500
1+6	39.3	2548
1+8	38.8	2581
1+10	39.8	2516
Latency at 2kvps		
Learners	(μs)	
2+0	1,017,583	
2+2	1,348,107	
2+3	1,348,107	
2+6	710,626	
2+8	585,312	

Table 8: (Ex.6) Average delivery rate and average delivery latency with different number of learners.

conclude that the cost of more learners is indeed handled by the network switch in this case, so from the protocol point of view they come almost for free.

Experiment 7 - Cost of packet size

Our network uses IP multicast to communicate. Since Paxos the protocol does not deal explicitly with fragmentation, the maximum size of a value to be delivered must fit in a UDP datagram, together with some header we need to attach. In

the current scenario, we can deliver messages up to 9kb (and Paxos values up to 8kb).
Starting from few bytes and then increasing, we measure the average delivery rate in a network composed of a proposer, three acceptors and a learner.
Figure 4 shows the significant drop in delivery rate when increasing the size of values. However the rate decays gracefully, sending big values does not make the system less stable, as we notice from the small variance of the rightmost set of values.

A decreasing rate does not imply worst performance in all

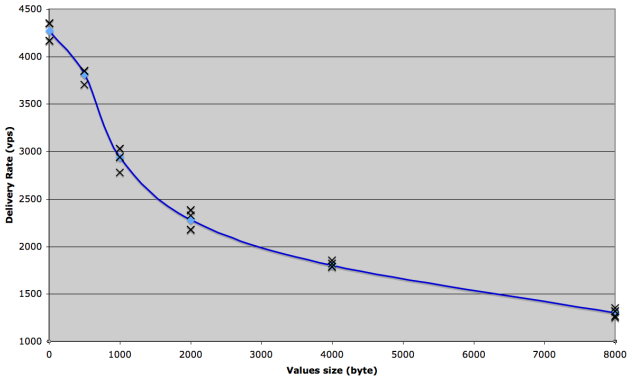

Figure 4: (Ex.7) Deacrease of delivery rate when increasing the size of values.

senses, in fact Figure 5 shows that with larger values we make a much better use of the network. The proposer can deliver data at 10MBps, while with very small values it can't even reach a tenth of megabyte.
This is an essential aspect to consider when using Paxos as a delivery service. A client application built on top of the proposer should try to batch values and submit them in chunks of 8kB to maximize throughput.
This optimization is sometimes made transparent to the client by implementing it directly in the proposer and learner. When creating *libpaxos* we decided not to have this feature since the application on top was already proposing large values in most of the cases. However after seeing how it can improve throughput in the general case, we surely consider it for the next version.

If the most important metric is latency it's still preferable to send small values, as visible from the results in Table 9.

Val.Size byte	Avg time s	Rate vps	TP MB/s	Lat. 1kvps μs
10	23.4	4274	0.04	
500	26.2	3817	1.82	369
1000	34.0	2941	2.80	
2000	43.8	2283	4.35	
4000	55.4	1805	6.89	508,965
8000	77.0	1299	9.91	421,987
mix 500/3000	39.2	2551	4.26	
mix 500/8000	61.2	1634	6.62	

Table 9: (Ex.7) Average time to deliver 100k fixed-size values, corresponding rate, throughput. Latency when delivering 1000 values per second.

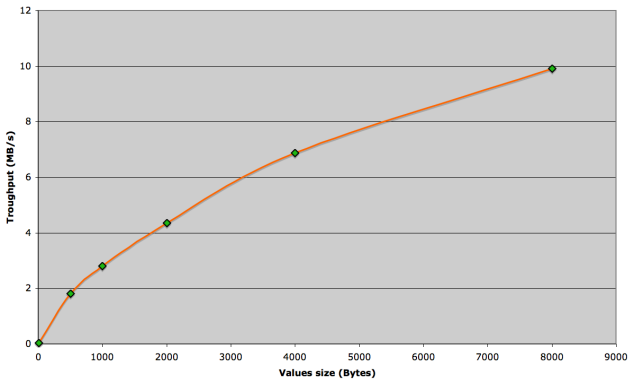

Figure 5: (Ex.7) Increase of throughput when increasing the size of values.

Experiment 8 - Library parameters

There are few compile-time parameters in our library which may directly affect performances. None of those was properly tuned and tested previously, they were chosen as "resonable defaults" during development.
Those parameters are:
Pre-execution window (**PEW**): the number of instances for which the leader proposer pre-executes phase 1. The current policy is to start this procedure if at least half of the previously prepared were consumed (i.e. closed with a value).
Proposer array size (**PAS**): entries in the in-memory structure of the proposer. It must be a few times bigger than the pre-execution window.
Acceptor array size (**AAS**): entries in the in-memory structure of the acceptor, it's used as a cache but all modifications needs to be write back immediately to permanent storage too.
Learner array size (**LAS**): entries in the in-memory structure of the learner.
Learner lsync interval (**LLI**): how frequently the learner checks to be in synchrony with delivered values (i.e. there are no holes).
The default values used so far are presented in Table 10 together with the other configurations tested in this experiment. We run with a proposer, 3 acceptors and 3 learners.

As shown by the results in Table 11, our choice of default values is not so bad.
Increasing the PEW (A) increases the workload of the proposer and the probability of phase 1 timeouts. Decreasing the LLI makes the learner more responsive (B), however if this change is combined with an increased LAS, the effect is the opposite (C). As expected, increasing the cache of the acceptor makes it a little faster (E).

Experiment 9 - Profiling

As a last experiment, we decide to dig deeper into the runtime behavior of each actor. We use the time profiling analysis of *Shark* for this purpose. We profile the execution with the proposer submitting values as fast as possible.
One thing that we notice in common for proposer, acceptor and learner is that most of their time (50% or more) is spent in system calls, specifically sent and receive operations on sockets. Unfortunately Shark does not build a tree that

	default	A	B	C	D	E
PEW	50	**300**	50	50	**300**	50
PAS	512	**1024**	512	512	**1024**	512
AAS	1024	1024	1024	1024	1024	**4096**
LAS	512	512	512	**2048**	**2048**	512
LLI	5	5	**2**	**2**	**2**	5

Table 10: (Ex.8) Configurations of library parameters.

	Timeouts	Avg.Time	Rate
default	10, 10	37.8	2645
A	100, 10	41.4	2415
B	100, 10	37.6	2659
C	10, 10	38.8	2577
D	150, 10	38	2631
E	10, 10	37	2702

Table 11: (Ex.8) Results for different configurations.

attributes the cost of those calls to the callers, rather it creates separate trees for system and library calls. Therefore is not given to know from where the expensive calls originate, this could be interesting especially with respect to the proposer.
This feature quickly allows us to determine that our part of the code is fine, making it faster would only produce a small speedup. We should focus instead on better I/O utilization, this could mean trying different policies or different calls to achieve better performances.
Another interesting observation regards locking. All the actors are multithreaded, generally there is a thread that sits waiting for packets, a thread that periodically wakes up for timeout checks and so on. In the extreme case the proposer has 3 threads, plus 2 for the learner that it starts internally plus at least one client thread. Nevertheless the time spent in locking is barely visible in the profiling. This should mean that our locking policy is not "getting in the way".
Shark makes easy to observe different threads in isolation, in the proposer for example, the CPU is used mostly by the internal learner. This is not unexpected since it receives quite some traffic from the acceptors and periodically must traverse a memory structure checking for holes.

7. CONCLUSION

In this set of experiments we tried to analyze the performances of *libpaxos* under different perspectives. In most of the cases, the system matched well our expectations but we made some very interesting findings.
When designing an application on top of *libpaxos* we need to keep in mind that larger values results in a better throughput but they have a cost in terms of latency. Adding more learner or acceptors is fine but may require prior verification of the underlying network capacity.
An essential parameter is the timeout interval, which needs to be calibrated carefully, this is made easy by enabling our event counters.
Discovering problems is also very useful, we can certainly say that at the moment disk logging is our Achilles' heel. Also adding transparent batching of values in the proposer may increase a lot performances when submitting a lot of small values.

8. REFERENCES

[1] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[2] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.

[3] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic monitoring of high-performance distributed applications. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 163, Washington, DC, USA, 2002. IEEE Computer Society.

[4] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[5] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[6] L. Lamport and M. Massa. Cheap paxos. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, Washington, DC, USA, 2004. IEEE Computer Society.

[7] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[8] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, page 260, Washington, DC, USA, 1998. IEEE Computer Society.

[9] X. Zhang, J. L. Freschl, and J. M. Schopf. A performance study of monitoring and information services for distributed systems. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, page 270, Washington, DC, USA, 2003. IEEE Computer Society.

APPENDIX

A. EVENT COUNTERS

verify_val: proposer receives a value from his own learner and matches it against the value sent.
delivering: proposer starts phase 2 sending an accept containing a client value.
delivering_reserved: proposer starts phase 2 but it's forced by the protocol to send a value from some acceptor.
instance_stolen: proposer cannot start phase 2 since another value has already been chosen.
p1_timedout: proposer times out while waiting phase 1 promises from acceptors.
p2_timedout: proposer times out while waiting his learner to deliver the last submitted values.
handle_promise: a promise message was received containing promises for multiple instances.

send_prepare: proposer broadcasts a prepare message containing multiple promise requests.
send_accept: proposer broadcasts a phase 2 accept message.
send_lsync: learner broadcasts a message to acceptors asking to retransmit some value.
handle_learn: learner receives a notification from an acceptor that it accepted some value.
total_holes: learner detects that it missed some learn message (i.e. it knows chosen values for instance 2 and 3, but not for 1).

B. BDB ACCESS METHODS

Paxos acceptors in libpaxos use Berkeley DB to implement permanent storage. We have the option to leave to the DB to decide when to write something on disk, or we can enforce disk synchronization, thus achieving fault tolerance. The access method that we use are:[8]:
Queue: Data is stored in a queue as fixed-length records. Each record uses a logical record number as its key. The page size of the database must be set to a multiple of the system page size that fits the data.
Recno: Data is stored in either fixed or variable-length records. Like Queue, Recno records use logical record numbers as keys.
BTree:Data is stored in a sorted, balanced tree structure. Both the key and the data for BTree records can be arbitrarily complex.
Hash: similar to BTree but uses hash-table. It exploits less the locality of data and works better for large working sets.

[8] parts from: http://www.oracle.com/technology/documentation/berkeley-db/db/gsg/C/accessmethods.html