RAM-ified Part-Time Parliament

Victor Domene

victordomene@college.harvard.edu

Gabriel Guimaraes

gabrielguimaraes@college.harvard.edu

Brian Arroyo

brianarroyo@college.harvard.edu

May 6th, 2016

Abstract

With the rise of persistent memory technologies [2], it is possible that in the near future we will have RAM-speed access with byte-granularity persistence. In this project, we seek to measure the performance differences between a Paxos implementation that uses disks as its *ledger* [1], and another that simply uses RAM. This may provide some insight into new applications of Paxos, where the algorithm may be ignored due to its low performance. This paper describes our implementation of the Paxos algorithm in Python (which, on its own right, deserves a great deal of attention) and presents benchmarking information on several workloads designed to stress the ledgers. Our results indicate that writing to disk is not the biggest bottleneck of the Paxos algorithm, as long as the proposal is reasonably small.

Contents

1 Introduction

In the classical Paxos specification [1], each legislator keeps a ledger with *indelible ink*; furthermore, such ledger must be *consistent*. These assumptions are key to the functionality of the algorithm. Within the context of an actual implementation, the closest to an *indelible ink* is the disk. However, with a new technology of persistent RAM [2], it is possible to conceive an implementation of Paxos that runs with improved performance. This gives rise to new use cases for the Paxos algorithms in settings where it would usually be refused easily, given the performance costs. This project is an attempt to benchmark the differences in performance between the Paxos implementation when the ledgers are backed by the disk (with *disk ink*), and when they are backed by RAM (with *memory ink*) itself.

The main objective of this project was to evaluate whether the performance of Paxos will be severely altered by changing the type of "ink" used in the ledgers. We observed the effects as we scale up the number of machines communicating, and gathered information on how much faster the algorithm can perform. We hypothesized that, by switching the "ink" to be simply RAM, the algorithm will have a significant performance boost, particularly due to the fact that, as the number of machine grows, the number of messages necessary to reach consensus grows, and then the amount of time spent in I/O operations may be large.

We have implemented our own Paxos library in Python. This paper will describe in depth the design choices made in writing this library, and shows the results we have found in benchmarking the different "inks". We also describe all of the different workloads we created in order to perform this benchmarking.

Our implementation can be found at this GitHub repository

2 Paxos Implementation

2.1 Overview

Our Paxos implementation follows Leslie Lamport's second paper on his Paxos algorithm [2], which simplifies the approach given in his original paper [1]. Lamport proposes an implementation based on three base roles: a Proposer, an Acceptor and a Learner.

In order to clarify these roles even further, more so than in the Lamport's second paper, we will use an analogy to the U.S. political system. In this case, we assume that a decree is of form "X will be the next president", and each proposal tries to say that X is some specific person.

Proposers are the main actors in Paxos. They are the machines who actually try to propose some value for a decree. If proposers were in the U.S. political system, they would be the Republican and the Democrat parties. The Republican party will try to pass the decree "Donald Trump is the next president", while the Democrat party will try to pass the decree "Hillary Clinton is the next president".

Acceptors are the voters. They are the machines who either accept, or do not answer a decree. Here the analogy is not perfect (since machines cannot choose between Trump and Clinton, but rather, can only accept a proposal or say nothing at all).

Learners are the people who actually count the votes. They are the machines who will receive the notifications from all of the voters, and check if any of the proposals had a majority of voters saying "Yes" to it. In this case, the learner is whoever counts the election votes and then decides who the next president is.

The role of Paxos is to ensure that, even if some of the votes do not arrive, only one president is elected; and that eventually, some president will be chosen, as long as there are enough people present to vote (for some definition of "enough", such as a simple majority).

2.2 Initial Design: Messaging System

The most obvious approach to implementing Paxos is by using some kind of messaging system. Indeed, even in Lamport's original paper [1], the legislators use their messengers to communicate between each other. Therefore, we define the following classes:

- Proposer;
- Acceptor;
- Learner;
- Messenger (sends messages to other machines);
- Receiver (receives messages from other machines);
- VM (encapsulates the abstraction of a machine).

Each Proposer, Acceptor and Learner must have a messenger in order to communicate. Each Receiver has a Proposer, Acceptor and Learner that it can access, in order to handle requests that it receives properly; it is, under the hood, a server. Finally, a VM class has access to both the receiver and the messenger, and it should be able to start a server, add new machines to the network, do benchmark computations, and so on. For more information on specific implementation details, refer to the documentation in the GitHub repository.

Given this abstract functionality, the next design decision is which communication protocol we were going to use in order to send messages to other machines and receive messages adequately. We chose to explore a protocol we had not used in practice: Protocol Buffers [3] with Google's new gRPC library [4].

2.2.1 Using Google's gRPC

We had decided on a class structure that depended on messages being passed between machines. Therefore, if we wanted to use gRPC, we would need to use it with the purpose of sending messages. We decided on the following service definition, using Protocol Buffers:

```
service VM {
    // Acceptors will receive these
    rpc handle_prepare (PrepareRequest) returns (OKResponse);
    rpc handle_accept (AcceptRequest) returns (OKResponse);

    // Proposers will receive these
    rpc handle_promise (PromiseRequest) returns (OKResponse);
    rpc handle_refuse (RefuseRequest) returns (OKResponse);

    // Learners will receive these
    rpc handle_learn (LearnRequest) returns (OKResponse);
}
```

Notice that each of the remote calls actually simply returns a dummy response; we do not use it in this design. Thus, on a simple sketch of how Paxos would work (assuming everybody promises and accepts without conflicts, so no calls to `handle_refuse` would be issued):

- Proposer calls `handle_prepare` on the Acceptors;
- Acceptors call `handle_promise` on the Proposer that sent the prepare;
- Proposer calls `handle_accept` on the Acceptors that promised;
- Acceptors call `handle_learn` on the Learners;
- Learners count the number of learns for a given proposal, and if appropriate, write it down to the ledger.

All of the `OKResponse` messages are supposed to be ignored (since we are simply passing messages around through gRPC).

We ran into several issues with this approach. First, RPC is supposed to mostly be synchronous: we issue a request, and wait for an answer. This is not ideal for Paxos, since we want to send all the `PrepareRequests` at once, and then handle promises later (ignoring the `OKResponse` results we will get from gRPC). Fortunately, gRPC provides a nice way to introduce asynchrony, by using the abstraction of a `future` [6]. Instead of actually waiting for the result, we can add a callback to one of these future objects. Since we want to simply ignore the results, the callback is a no-op.

We also had delightful hours debugging remote code. Whenever the remote gRPC fails somehow, the Python stack trace is extremely unresponsive, and simply say there was some `RemoteError`, and it refuses to respond to SIGINT signals (from `Ctrl+C`). Due to this difficulty in debugging, we decided to parallelize and try another communication protocol, and come back to gRPC later.

2.2.2 Using our own RDTP (Real Data Transfer Protocol)

In order to debug our code, we decided to implement our messenger and receiver with RDTP [5]. This protocol was implemented by the authors of this paper previously, and it is actually meant for message sending communications (it was written for a chat application).

Therefore, we continued with the same design choices we had for the previous section, with the exception that now we used a simple messaging protocol for communication.

When we want to send a `prepare` request to the Acceptors, we make the following call, which sends a message over the network:

```
rdtp.send(destination_socket, 0, "send_prepare", proposal_number, ...)
```

This sends a message with status 0 to `destination_socket`, with the arguments described.

In general, RDTP worked really well with our initial design, and we decided to keep it for some extra benchmarking. We fixed most of the bugs we had in the proposer code, and then we decided to go back to gRPC.

2.3 Alternative Designs

2.3.1 Using gRPC in a better way

After some time, we realized that the way we were using gRPC is not ideal. We could, instead, simply have the response from a `handle_prepare` be a `PromiseRequest`. This scheme can be sketched out as follows:

```
service VM {
    // Acceptors will receive these
    rpc handle_prepare (PrepareRequest) returns (PromiseResponse);
    rpc handle_accept (AcceptRequest) returns (OKResponse);

    // Learners will receive these
    rpc handle_learn (LearnRequest) returns (OKResponse);
}
```

The sequence of calls would be:

- Proposer calls `handle_prepare` on the Acceptors;
- Acceptors respond with a promise;
- Proposer calls `handle_accept` on the acceptors;
- Acceptors respond with an "OK" and call `handle_learn` on the learners;
- Learners respond with an "OK".

Here, we save one extra call from the Acceptors to the Proposers, by answering the RPC call with the promise. However, this design would violate our initial class design where messages are passed, and a lot of the design choices would have to be redone (for

instance, we would have to change the way we use Receivers and Messengers, by passing in to them a callback to actually handle the `PromiseRequest` received in the RPC). We chose to not go further with this approach.

2.4 Final Design

Our final implementations supports both gRPC (with our initial messaging design) and RDTP. Also, it is very easy to extend the library to use any other protocol for communication, as long as it presents the structure of the abstract class in `receiver.py` and `messenger.py`. The files are well-documented in order to support further extensions.

3 Workloads

All of the following workloads can be found in our GitHub repository [7], under `/workloads/WORKLOAD_NAME.py`.

3.1 Workload A: Single Proposer

The simplest workload we designed is to have a network of size `NETWORK_SIZE`, where only one of the machines is a proposer. Furthermore, the proposer only makes new proposers once every second.

3.2 Workload B: Two Proposers (surviving forever)

The second type of workload we designed used a network of size `NETWORK_SIZE`, where two of the machines were proposers. The proposers make requests once every second, and one of them (which we call M0) starts slightly earlier than the other one (M1).

3.3 Workload C: Two Proposers (one dies)

The third type of workload designed is very similar to Workload B, but now we require that the first proposer (M0) dies after some time. This is used to check correctness

of Paxos, which should still be able to accept and find new values for decrees using the proposals from M1 (the surviving proposer). We did not use this workload for benchmarking, since it does not provide us with any insight on the main objective of this paper, but it does allow us to check correctness.

3.4 Workload D: Multiple Proposers (surviving forever)

The last workload we implemented can spawn `NUM_PROPOSERS` proposers in the system with a `NETWORK_SIZE` (as long as `NUM_PROPOSERS < NETWORK_SIZE`, of course). We did not use this workload for benchmarking, since we considered the case of two proposers enough for our purposes. However, it was important for testing and debugging.

3.5 Variable Parameters

We can very easily change between RDTP and gRPC, as well as between RAM ink and disk ink. It is also very easy to tweak the `NETWORK_SIZE` to get different behavior for a different number of machines.

Within RDTP and gRPC, we can also add sleeps in the middle of the code, with some random value distributed around 50 milliseconds. This allows us to simulate bad network situations and difficulty in communication.

Finally, our Paxos assumes that the values for decrees are simply integers. Writing integers to disk or to RAM probably will not give a significant difference in performance, since it is a very small write. Because of this, we also vary the amount of bytes we actually write to disk, by simply writing the proposal number several times. This is used only for benchmarking, and should not happen on regular Paxos run, but it simulates the case where proposed values are larger, and can thus provide us with significant insight.

4 Benchmarks

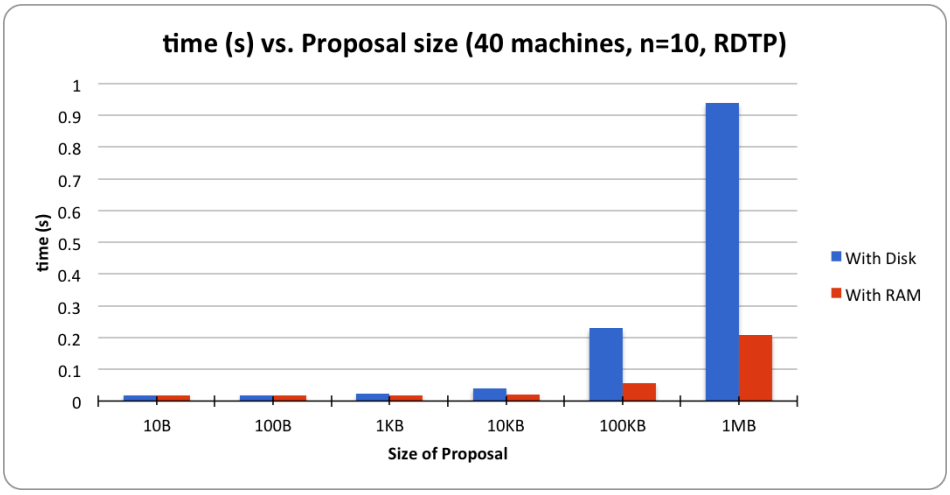
All of the benchmarks were peformed using SSDs, which is quite an important factor into the time it takes to write blocks to disk. SSDs are fairly faster than regular, old

school hard disks. Therefore, we can believe our results to be more significant in a setting of an older server that only has access to a hard disk.

The only benchmark we actually take is the time between when a single VM proposes a value, and when it learns that the value has been chosen by the pool of acceptors. Ideally, we would have more specific benchmarks to isolate where the bottlenecks are, but due a lack of agreement between the authors of what those benchmarks should be (and timing constraints), we omit those.

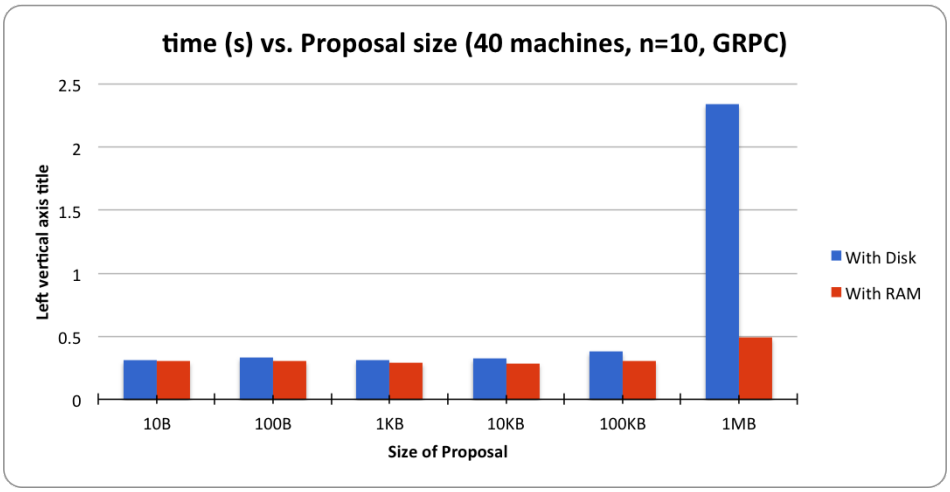
4.1 Workload A

4.1.1 RDTP, RAM vs. Disk, fixed machines, variable proposal, no latency



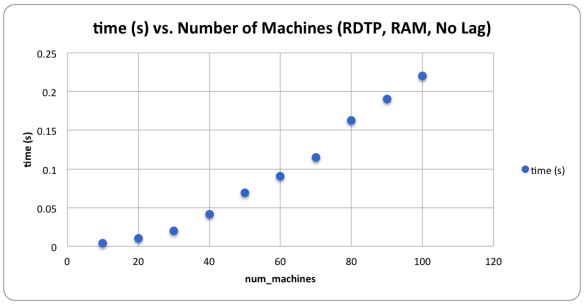
Notice that for small proposal sizes, the RAM vs. disk comparison is not particularly significant. Only after 10KB, we can actually notice some difference in performance.

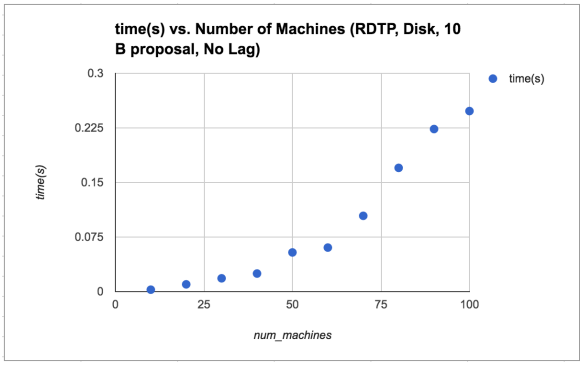
4.1.2 gRPC, RAM vs. Disk, fixed machines, variable proposal, no latency



A similar trend from the previous benchmark appears here, except that gRPC itself introduces much more overhead. The disk vs. RAM comparison is dwarfed by the effect of the communication protocol, until we get to around 100KB.

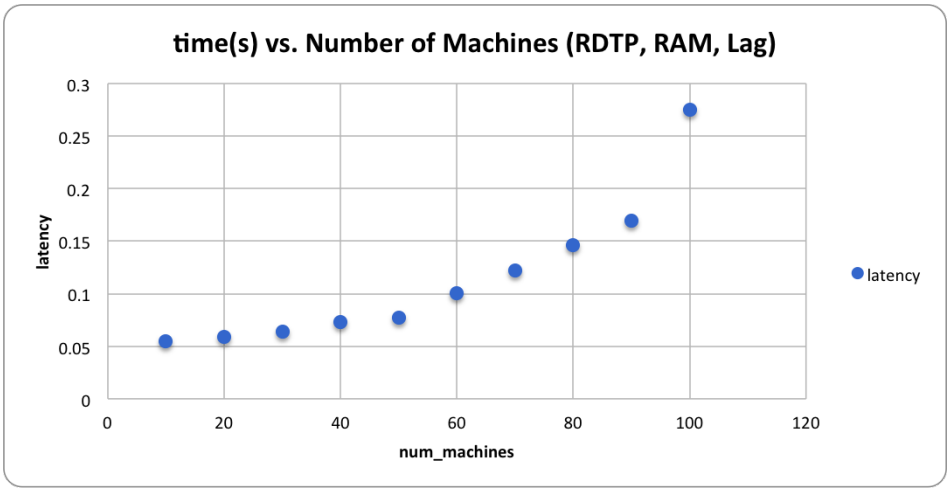
4.1.3 RDTP, RAM vs. Disk, variable machines, fixed proposal, no latency





For this set of graphs, we varied the number of machines. We did not notice any differences between RAM and disk for this comparison, probably because the proposal size was small (1KB).

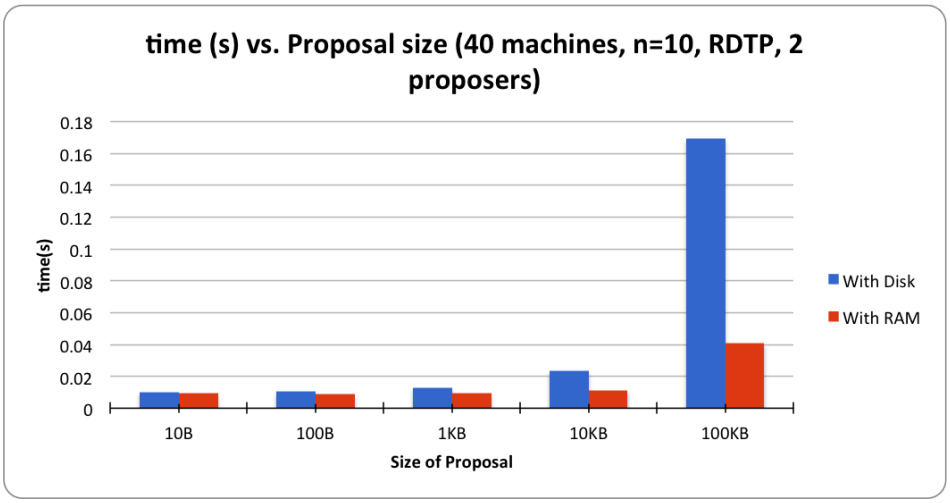
4.1.4 RDTP, only RAM, variable machines, fixed proposal, latency



Here, latency can be seen to have a much more profound effect in the performance when compared to the performance with the disk.

4.2 Workload B

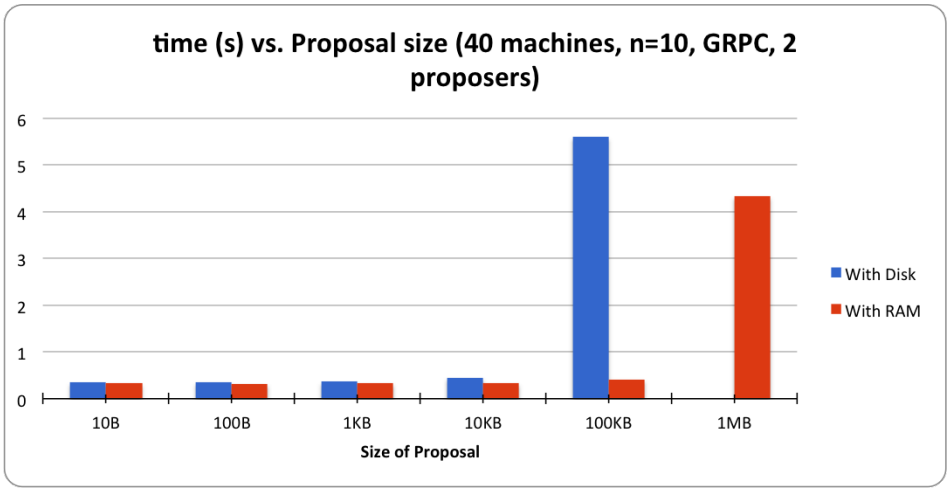
4.2.1 RDTP, RAM vs. Disk, fixed machines, variable proposal, no latency



The same kind of trend from Workload A happened again: we only see significant increase after 10KB. Notice, however, that there are many more messages being sent

(due to the nature of having more proposals), so that we see a more significant effect.

4.2.2 gRPC, RAM vs. Disk, fixed machines, variable proposal, no latency



The same trend from the previous benchmark can be seen, except that, as before, gRPC is much slower than RDTP. In fact, we did not even run gRPC with 1MB, since it overloaded our machines.

5 Conclusions

Paxos was an incredibly interesting algorithm to implement (and quite hard). There were several parameters that we could vary in our experiments:

- RDTP vs. gRPC;
- Proposal Size;
- Number of Machines;
- RAM vs Disk;
- Number of Proposers;
- Latency.

We arrived at the following conclusions:

- Google's gRPC works very reliably. However, it incurs a high performance cost when compared to simpler, lower-level message-passing schemes. Furthermore, it can prove challenging to debug.
- With small proposals, writing to disk or to RAM did not affect performance, since the network latency and the general overhead of the Paxos algorithm will dominate the measurements. We currently wonder whether an implementation of Paxos in C++ or C could present less amount of overhead as our Python implementation, in order to highlight the differences between disk and RAM more reliably.
- Obviously, increasing the number of machines in the network will increase the number of required messages, affecting performance and increasing runtime.
- With more proposers, we usually saw a higher number of messages sent, and therefore a slightly higher runtime.
- Introducing higher network latency will greatly affect performance, since the whole Paxos algorithm relies on the speed of messages.

In general, our hypothesis that writing to disk can incur extra overhead is proven correct, as long as the proposal is big enough (roughly, more than 100KB, which seems unlikely for general purposes nowadays).

6 References

[1] LAMPORT, Leslie. "The Part-Time Parliament".

http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf

[2] LAMPORT, Leslie. "Paxos Made Simple".

http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf

[3] Google's Protocol Buffers.

https://developers.google.com/protocol-buffers/

[4] Google's gRPC Library.

http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf

[5] RDTP GitHub repository.

https://github.com/gablg1/rdtp/

[6] gRPC Future Interface.

https://github.com/grpc/grpc/blob/master/src/python/grpcio/grpc/framework/foundation/future.py

[7] RAM Paxos GitHub repository.

https://github.com/victordomene/ram-paxos