Universidade do Minho
Escola de Engenharia
Departamento de Informática

Diogo José Cruz Sobral

Consensus in high performance computing

March 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Diogo José Cruz Sobral

Consensus in high performance computing

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
José Orlando Pereira
Ana Nunes Alonso

March 2022

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor, José Orlando Pereira, for all the support and availability over the last few months. Without him, I wouldn't be able to overcome many of the problems and challenges faced along the way.

The pandemic forced everyone to be on lockdown. A big part of this work was developed at that time creating a need for a good workplace. For that, I would like to thank my parents for helping to keep the motivation high, and for maintaining a healthy work environment at home while the lockdown took place.

Last but not least, I would like to thank my friends. The moments of fun and distraction played a crucial role in recharging my energies and maintaining my focus.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

For a long time, developers and scientists designed consensus solutions that sacrificed thread scalability or load balance to decrease the latency. The appearance of networking and memory with microseconds requests reshaped how high-throughput consensus solutions are designed today. Today's protocols must be multi-threaded to scale and seize the high hardware-level parallelism available. Moreover, their focus should be on not overloading the system instead of decreasing the latency.

In the last decade, storage devices underwent an enormous development in their performance. The NVMe devices bought the request's latency of solid-state devices to the microseconds range. Additionally, the software available to manage and operate these devices kept up with the hardware development. Libraries such as the Storage Performance Tool Kit (SPDK) appeared to ease the development of high-performance storage applications. In this context, this work aimed to answer a fundamental question: can NVMe Devices and SPDK improve the existing consensus-related work?

This dissertation describes a solution for the distributed consensus problem that combines Disk Paxos (a consensus algorithm that relies upon writing and reading in a network of storage devices to achieve a distributed agreement) with NVMe Devices and SPDK. We conceived this solution using C++ and conducted a performance evaluation that, in the end, compared our solution with LibPaxos. Our findings describe the issues and benefits of the usage of these technologies to solve consensus. With our approach, we increase the understanding of the potential of these new technologies in enabling better solutions for the consensus problem in the future.

KEYWORDS Distributed Systems, Distributed Consensus, NVMe, Storage Performance Tool Kit, Network Storage.

RESUMO

Durante bastante tempo, as abordagens adotadas para resolver o consenso distribuído sacrificavam a escalabilidade do número de "threads" ou o balanceamento da carga com o objetivo de diminuir a latência da solução. O aparecimento de redes e latências de memória, com pedidos na ordem dos microssegundos, alterou a forma como os algoritmos de alto desempenho são desenhos, hoje em dia. Os protocolos atuais têm de ser "multi-threaded" para conseguirem escalar e aproveitar o elevado paralelismo existente do hardware disponível. Para além disso, o foco dos protocolos deve residir em evitar a sobrecarga do sistema e não na redução da latência.

Na última década, houve um grande desenvolvimento nos dispositivos de armazenamento. Os discos "NVMe" introduziram latências na gama dos microssegundos aumentando significativamente o desempenho dos discos de armazenamento. Adicionalmente, o "software" disponível, para gerir e operar sobre estes dispositivos, também acompanhou o seu desenvolvimento. Bibliotecas, como o "Storage Performance Tool Kit (SPDK)", apareceram para facilitar o desenvolvimento de aplicações de armazenamento de alto desempenho. Neste contexto, este trabalho teve, como principal objetivo, responder à seguinte questão: Podem os dispositivos NVMe e o SPDK melhorar o trabalho existente sobre o consenso distribuído?

Esta dissertação descreve uma solução para o consenso distribuído que combina o Disk Paxos, um algoritmo que resolve o consenso distribuído, através de leituras e escritas numa rede partilhada de discos, com os dispositivos NVMe e a biblioteca SPDK. Esta solução foi desenvolvida em C++ e foi realizada uma análise de desempenho que, na sua fase final, compara a mesma com outra solução existente, o LibPaxos. A análise desenvolvida descreve os problemas e os benefícios de usar estas tecnologias para resolver o problema de consenso distribuído. Com esta abordagem, foi possível explorar e descobrir o potencial que as tecnologias usadas têm para possibilitar o desenvolvimento de melhores soluções no futuro.

PALAVRAS-CHAVE Sistemas Distribuídos, Consenso Distribuído, NVMe, Storage Performance Tool Kit, Armazenamento em Rede.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1

INTRODUCTION

1.1 MOTIVATION

As the load placed in the modern-day distributed systems grows with the ever-increasing demand of daily users, it is essential to maintain a high quality of service. A user might evaluate a service by metrics such as system uptime, request-response time, and others. These metrics usually require fault-tolerant systems and horizontal scaling to maintain the high standards expected by the end-users. As a result, most solutions require agreement protocols across the system's components to achieve replication.

Distributed consensus is an abstraction of agreement problems between processes, and is a fundamental problem to solve in distributed systems. It allows a group of processes to settle on some specific value; as a result, all participants decide the same way. State machine replication, atomic broadcast, or even blockchains are simple examples that require a solution for this problem to be implemented. When it comes to real-world applications and services, consensus is present in a wide variety of solutions. For instance, the etcd distributed key-value store that provides a reliable way to store data in a distributed system is built on top of the Raft Protocol (a consensus algorithm) [8]. Raft gives etcd a total ordering of events across a system of distributed etcd nodes. Another example is the MySQL Group Replication. This distributed database is known for its multi-master architecture and relies on the Paxos algorithm to implement its built-in coordination mechanisms [9].

There have been multiple approaches to address the consensus problem that vary from using message exchanges to relying on direct memory accesses [10, 15, 12]. Another example of a different solution is the Disk Paxos algorithm [1]. It presents a simple and efficient approach that relies on writing and reading in a network of shared storage devices to solve the consensus problem. Moreover, the reliance on disks makes this algorithm extremely robust since hardware failures are less likely to occur than software ones. Many of these approaches lost relevance with the latest modern hardware and were surpassed by newer solutions. Others were refactored to take advantage of kernel bypass mechanisms such as RDMA or DPDK [13]. The appearance of the NVMe Solid State devices introduced a new era of high-performance storage. This major improvement opened an opportunity to enhance Disk Paxos with this new technology.

In an age of emerging big data problems and cloud solutions, the importance of network storage grew significantly. Every day, the amount of data generated and processed by today's applications goes larger and pushes the load upon data centers further. NAND Flash Technology has made solid-state drives affordable and viable for data centers and is increasingly displacing disk drives due to its higher IO rates [6]. In spite of the

improvement with the substitution of HDD devices for SATA SSDs, NVME devices present a higher performance than its predecessors and introduced a new era of high-performance storage. A performance analysis of NVME SSDs on real-world databases [5] revealed a 3x higher bandwidth while writing and 2x overall speed on MongoDB, and a 3.5x increase in the performance of MySQL when comparing it with SATA SSDs. Nevertheless, the NVME specification states that these state-of-art devices outperform SATA SSDs by more than five times. This technology breakthrough improved storage applications and unlocked new solutions that previously were noncompetitive.

Ultimately, the landscape of storage applications has shifted into brighter days due to better software available to manage and operate devices. Libraries such as the Storage Performance Development Kit (SPDK) appeared to maximize the performance out of the NVMe Devices, outperforming kernel's implementations. All these factors not only may increase the storage applications' performance, but also make environments suitable for Disk Paxos more common.

1.2 PROBLEM STATEMENT

For a long time, the focus while designing consensus solutions was lowering the protocol's latency or reducing the number of messages exchanged. Consequently, properties such as threading or load balancing would often be sacrificed. The technological development of processors, networks, memory, and storage in the last decade and a half reshaped how these solutions are designed. Old solutions may no longer be efficient today, while yesterday's algorithms that performed poorly may be efficient today due to the improved hardware available. With the microseconds latency of networks and memory, the new consensus solutions should prioritize load balance and threading to scale and take advantage of the high hardware-level parallelism of today's components [7]. The focus should remain on not overloading the system to keep the latencies in the microseconds range.

Seizing the enormous improvement around the last generation of solid-state devices (SSD) and the nowadays multi-core CPU architectures, this work aimed to answer a fundamental question: Can NVMe Devices, when combined with state-of-the-art software for driver management, improve the existing consensus-related work? NVMe Devices achieve a request's latency under the microseconds range and, as such, present an opportunity to open up a door to a set of solutions relying upon storage devices. Moreover, the low latency of these devices also requires software capable of keeping up with their performance. The Storage Performance Development Kit (SPDK) is the state-of-the-art software available to build storage applications. It provides a set of libraries and tools to build high-performance storage applications and efficiently operate the SSD disks.

NVMe devices and SPDK seem to us a promising path for a future family of consensus algorithms. These technologies follow the same path as nowadays high-performance consensus solutions enabling the design focus on high hardware-level parallelism and threading, two properties required for high throughput solutions [7].

1.3 OBJECTIVES

The aim of this thesis is to gain an understanding of the potential of NVMe Devices and SPDK as a means to solve the distributed agreement. Both NVMe Devices and SPDK have high-performance properties ideal for developing and implementing a high-throughput system. To meet our goals, we develop a consensus system using Disk Paxos that established a network storage composed of NVMe Devices and relied upon SPDK to establish any communication necessary to those devices. The list below presents a set of goals to achieve to find an answer to our problem.

1. Evaluate the restrictions inherited by the usage of SPDK.
2. Propose an algorithm capable of achieving consensus while allowing multiple decisions simultaneously.
3. Study possible approaches for multi-threading solutions taking advantage of today's multi-core architectures.
4. Design and implement a prototype that addresses consensus by combining NVMe Devices and The Storage Performance Development Kit (SPDK).
5. Evaluate the potential of the produced work and compare it with existing solutions.
6. Propose optimizations to address existing issues.

2

STATE OF THE ART

2.1 STATE MACHINE REPLICATION

A state machine approach is a general method used to ensure the reliability of a distributed system. Fault tolerance requires replication and this procedure enables the system to have multiple processes replicating a service. Moreover, replication is usually expected to be transparent to the end-user. Replica consistency is a fundamental issue to solve to maintain this property.

While using this approach, it is possible to keep multiple replicas at the same state and guarantee the desired consistency. As such, there is a set of rules the system must fulfill. Each state machine should have a set of both inputs and outputs, a distinguished starting state, and should exclusively support deterministic operations. If all replicas apply the same input instructions in the same order to the starting state, they will end up in an equal state. As long as the system keeps the determinism in each transaction, this assumption remains true.

The solution for the consistency problem is an agreement in the order by which inputs are handled [24, 25].

2.2 CLASSIC PAXOS

Classic Paxos [10, 11] is the basis of many algorithms that solve consensus today. It provides a solution to reach an agreement between a set of processes in a distributed system. It is described as the collaboration of: *proposers*, *acceptors*, *learners*.

- *Proposers* - attempt to convince acceptors into accepting requests received by clients
- *Acceptors* - act as the fault-tolerant memory of the system and handle messages from proposers
- *Learners* - keep the replicated state and compute responses to clients

Classic Paxos requires a set of assurances to guarantee it can solve consensus. The distinct agents must be able to communicate, and their messages may take arbitrary time to arrive, may be lost or duplicated, but they can't be corrupted. In addition, the algorithm supports agents running at arbitrary speeds and failures and recoveries. However, it doesn't tolerate byzantine faults.

The algorithm begins by having a proposer send a *prepare* message to a majority of acceptors containing a proposal number *n*. Upon receiving it, each acceptor falls into two distinct situations. In the first case, it has

already promised to another process that it won't be accepting proposal numbers lower than m such that $n < m$ and sends a "Nack" to the proposer. In the second scenario, the proposal number (*n*) received is the highest until that point. The acceptor responds with the value of the highest ballot number accepted if it exists and a promise that it will decline all proposal numbers lower than *n*.

The response from a quorum of acceptors marks the end of phase one. Right after, the proposer chooses the highest ballot number value among the responses or its own value if none has one and starts phase two.

During phase two, the proposer sends an *accept* request with the chosen value to a majority of acceptors. These accept the value only if they have not seen a *prepare* request with a higher proposal number and respond to the proposer with an acknowledgment. After receiving it from a quorum of acceptors, the proposer broadcasts the result to all learners. Alternatively, the acceptors could message all learners directly, providing a more reliable approach at the cost of additional messages.

Lastly, there might be circumstances where two or more proposers are executing simultaneously, canceling each other out and preventing progress. These scenarios are avoided with a selected proposer [11]. This proposer would be the only one issuing proposals, and if able to communicate with a quorum of acceptors, it would eventually complete the algorithm, ensuring progress.

2.3 MULTI-PAXOS

Multidecree Paxos [2], often know as multi-Paxos, allows multiple instances of the classic Paxos algorithm to run simultaneously. Each instance performing the complete algorithm would result in a tremendous overhead. Therefore, multi-Paxos skips Paxos's first phase if the leader is stable. This approach relies on three distinct core entities: replicas, leaders, and acceptors.

A replica is responsible for handling requests from its clients. Since a client broadcasts its requests, different processes receive them in a distinct order requiring an agreement to ensure consistency. Each time a request arrives, the replica chooses a slot number and sends a proposal to the leaders containing a pair with both the slot number and the request. Eventually, the process will receive a decision message from a leader with a slot's outcome. A replica then verifies if its current state is ready to handle the decision and if it has to retransmit an older proposal. If the decision is applied, a response message is sent to the respective client.

An acceptor keeps the fault-tolerant memory of Paxos. Each acceptor maintains a ballot number and a set of triples where each triple has a ballot number, a slot, and a command. This set contains all values accepted yet. This process's type receives two distinct messages:

- p1a, which changes the acceptor's ballot to the maximum between its value and the one received. It responds with the current ballot and the set of accepted triples.
- p2a, which adds the values sent to the accepted set if their ballot is equal to the acceptor's ballot

Each leader receives proposals from replicas and starts the consensus protocol for them. A slot has multiple requests, however, only one should be accepted. This process type maintains a set with all proposals admitted for this purpose. Whenever a leader receives a proposing message, it verifies if it has not seen that slot yet,

adds it to its set, and spawns a thread named commander. A commander performs the classic Paxos's second phase. It starts by broadcasting to acceptors a p2a message with the ballot number, slot, and command passed to it as arguments upon creation. If the ballot number sent differs from the response, then the current value has expired, and the leader is informed with a preempted message. Otherwise, the commander waits until it receives a message with an equal ballot number from a majority of acceptors. In this case, a decision has been accepted and should be broadcasted to every replica.

Multi-Paxos allows many agreements to run simultaneously by using the same ballot number for distinct slots. As long as an acceptor's majority has the same ballot number as the leader, a commander will always be able to decide for a slot. However, if this does not happen, the system won't progress until the leader's ballot and the set of proposals updates. A leading process uses another background thread for this purpose, a scout. A scout computes the first phase of a classic Paxos. It sends a p1b message to acceptors and waits until it either receives a response from a majority of processes with the same ballot number as its own, or a high-numbered ballot response. The first case results in messaging its leader with an adopted message, while the second one delivers a preempted one.

A leader then receives two additional messages. The preempted one means that its ballot has expired. When one of these arrives, a leader changes its state to not active, increases its ballot number, and spawns a new scout. This scout updates every acceptor's state and exits with another preempted message forcing the repetition of this procedure or an adopted one. With the arrival of an adopted message, a leader finds out that its current ballot is valid and should return to the execution state. For such, it updates its set of proposals and spawns a commander for each of them. Eventually, every slot will be decided, and as long as the system's leader remains stable, a ballot number won't expire permitting multi-Paxos to skip the classic Paxos's phase one.

2.4 DISK PAXOS

Disk Paxos is an algorithm for implementing a fault-tolerant distributed system. Like the traditional Paxos, this approach tolerates non-byzantine failures and can assure progress as long as a single process is kept operational and the system is stable. Disk Paxos differs from most solutions by how it maintains consistency and how it reaches an agreement. Instead of relying on an exchange of messages with other processes, this algorithm uses write and read operations in a set of disks. It requires both, a network of processes and shared disks to implement this approach.

Since hardware failures are less likely to occur than software ones, the use of disks enables Disk Paxos to provide an efficient and simple solution to build a reliable system. Classic Paxos [10] might be seen as a variant of Disk Paxos where each process has its disk without having accesses from other machines [1].

2.4.1 *The algorithm*

A consensus algorithm allows a set of processes to agree on which command to execute at a given n^{th} operation. This type of algorithm requires each intervenient to have a starting input and is multiphased. For example, the

classic Paxos requires 3 phases to make a decision. In this case, Disk Paxos requires only two. The first one where a process decides if it either can choose its input or must use another's and, the second one responsible for attempting to commit the decision. Both phases write and read in a majority of disks. If the second phase finishes successfully, then a command has been decided and should be broadcasted to the remaining constituents of the system.

To perform an instance of this algorithm, a process *p* starts by numbering its attempt with a ballot number. A ballot number must be a positive integer only used once by a process. After this, phases one and two take place. During their execution, *p* keeps a block with its state in both memory and each shared disk. As such, each disk must have a region dedicated to each process. The paragraphs ahead will be referring to the block kept in memory as *dblock*.

A block contains three different components:

- *mbal*, The current ballot number
- *bal*, The largest ballot number for which p entered phase 2. (Initially, 0)
- *input*, The value p tried to commit in ballot number bal. (Initially, a special value NotAnInput)

In both phases, process *p* initiates each one by, concurrently, attempting to write the current value of *dblock* in each disk. It then reads the blocks from the remaining processes and inserts them in a set with all blocks seen so far (*blocksSeen*). If it finds a block with *mbal* higher than its current ballot number, *p* must call for an abortion, starting again at phase 1 with a higher ballot number. Each phase ends with *p* writing and reading from a majority of disks.

After the writings and readings phase ends, either p ends phase 2 and commits the input value of *dblock*, or it chooses a new value for *dblock.inp*, sets *dblock.bal* to *dblock.mbal* and enters phase 2.

The choice for a new value for *dblock.inp* is accomplished by the analysis of the set with every block read in the previous phase. Therefore, process *p* computes the subset of *blocksSeen* whose records have an *input* field different from *NotAnInput*. Once a process writes a block with an *input* not *NotAnInput*, all subsequent agreement attempts must determine an *input* within one of the values previously wrote. This is required to ensure consistency. So, if the computed subset is empty then *dblock.input* is set to the input value of *p* otherwise it chooses the *input* field of the record with the highest *bal* number within the subset.

2.4.2 *Considerations*

Disk Paxos requires a leader to make progress. Despite the instability of the system and the existence of multiple leaders simultaneously, it ensures consistency. Under these conditions, an instance of this algorithm may not make progress until the system becomes stable again with a single leader elected. This requires a real-time leader election to occur in the background.

To support a dynamic network, Disk Paxos also requires the introduction of other mechanisms. Lamport [1] suggested keeping a directory in each disk listing both the processes and the locations of their blocks in the disks. A process joins the system by adding itself to the directory after acquiring a distributed lock on it.

No further information about this algorithm will be covered. A detailed version of the algorithm is available in the original publication [1].

Algorithm 1 Disk Paxos - Consensus Protocol

```
1:  procedure START BALLOT( )
2:      dblock.mbal ← Value Higher than dblock.mbal
3:      blocksSeen ← {dblock}
4:      Phase1or2(p, 1)
5:  end procedure
6:  procedure PHASE1OR2(p,phase)
7:      repeat concurrently for d ∈ disks
8:          disk[d][p] ← dblock                    ▷ Writes dblock at disk d, block p
9:          for q ∈ processes, q ≠ p do
10:             blockQ ← disk[d][q]                ▷ Reads block q of disk d
11:             if dblock.mbal < blockQ.mbal then
12:                 blocksSeen ← blocksSeen ∪ {blockQ}
13:             else
14:                 Abort() ∨ StartBallot()        ▷ Abort if it thinks it's no longer the leader
15:             end if
16:         end for
17:     until Has completed for a majority of disks
18:     if phase == 1 then
19:         dblock.bal ← dblock.mbal
20:         nonInitBlks ← {bk ∈ blocksSeen : bk.inp ≠ NotAnInput}
21:         if nonInitBlks ≠ ∅ then
22:             maxBk ← max{bk.bal : bk ∈ nonInitBlks}
23:             dblock.inp ← maxBk.inp
24:         else
25:             dblock.inp ← input
26:         end if
27:         Phase1or2(p, 2)
28:     else
29:         Commit dblock.inp
30:     end if
31: end procedure
```

2.5 MODERN APPROACHES TO CONSENSUS

In a world where applications, networks, and memory are at the microseconds latencies, the consensus problem remains an issue in high availability systems. The most common approach to replication, state machine replication, induces an overhead on the hundred of microseconds range [15, 14]. Typically, two main challenges to performance arise, protocol latency and high packet rate. The high protocol latency is mainly due to the software network layers in OS Kernels [16], while the high packet rate comes from the number of messages Paxos roles must handle to reach a high throughput [12]. Modern hardware provides solutions for these issues either by using kernel bypass mechanisms or running specific parts of the algorithms in the network's hardware. The paragraphs above present recent approaches to address the consensus problems using modern hardware to accelerate its performance.

P4xos [12] describes an approach to solve consensus that shows how a distributed system becomes a distributed networked system. In Paxos, a message from a leader to an acceptor travels over Top of Rack, Aggregate, and Spine switches before arriving at its destination. P4xos takes advantage of these programmable switches and deploys the logic of the distinct Paxos's roles in network hardware. The messages used are encoded using a custom packet header. As the packet travels through different switches (each with a specific role in the algorithm), each switch changes the fields inside the packet header before forwarding it to the next stop working as a response for the input messages and completing the different phases of the algorithm. P4xos reduces the end-to-end latency by completing consensus logic as the messages travel over the network. Additionally, the usage of forwarding hardware enables P4xos to avoid the I/O bottleneck of software implementations [12]. The P4xos prototype was implemented with P4 [17] and showed a latency 3x time lower than *libpaxos* [18] (an open-source implementation of Paxos).

Similar to P4xos, more solutions tested network hardware-based approaches to accelerate the consensus problem. Belocchi et al [20]. developed a consensus implementation to run inside smart NIC. Their solution translated Paxos logic into the XFSM based abstraction to run in the hardware using the FlowBlaze [19] engine. In their approach, both proposer and learner roles run on the host while acceptors and the leader are deployed into hardware switches.

Another path to improve consensus is the kernel bypass. **Mu** [15] solves the distributed agreement and implements a replicated state machine leveraging the Remote Direct Memory Access (RDMA) to bypass the OS kernel. RDMA, as the name implies, enables a process to read and write data directly in another process. Mu specifies that a replica is either a leader or a follower. Each replica maintains a consensus log structure keeping all the relevant data. The leader reads and writes on the logs of the followers to execute the algorithm. Seizing the write permission feature of RDMA, the leader only executes the algorithm while it has the write permission in a quorum of followers. At the start of execution, a new leader sends a permission request to all replicas and waits for a quorum of responses meaning that the permission was granted. For each follower, only a single leader has the write permission implying that a permission's majority is exclusively achievable by a unique leader. From there, it executes the protocol with a prepare and acceptance phase. Whenever a leader executes the prepare phase finding out only empty values for a given index, it starts skipping the prepare phase for higher indexes until it aborts due to a write failure. In this scenario, a leader repeats the initial procedure of acquiring the write

permission on a quorum of followers and executing the prepare and acceptance phase in case of still believing it should lead. Mu can outperform the other replications systems by at least 2.7x times [15].

As the last example, **Apus** [16] is presented. Apus follows a viewstamp-based Paxos protocol [21] and combines it with the RDMA technology. As a result, it executes an agreement in a one-round algorithm taking advantage of one-sided RDMA read/write operations to achieve a microsecond latency. Apus splits its system entities into leaders or backups. When a request from a client arrives, a leader assigns a request id, allocates a log entry in its consensus's log, and stores the entry in its local storage. Each log entry contains a timestamp of the current view, an array (*reply*) to keep track of acknowledgments from backups, and it's accessible remotely via RDMA. Additionally, each backup also has its consensus log. The next leader's step is writing its log entry in the remote memory of backups using RDMA. Whenever a backup notices that the latest unagreed entry changed, it compares the entry's timestamp with its own, accepts the value, and stores the entry in its local storage. Then the backup acknowledges the agreement by writing in the leader's *reply* array. While the backups are executing, the leader polls its local *reply* array waiting until a quorum of backups responds to its request. This one-round algorithm is possible due to the viewstamp-based approach. Every time a leader dies, the backups execute the classic two-round Paxos to agree on the next view before returning to the normal execution.

Given the above, many newer approaches are focusing on leveraging hardware improvements and software enabling kernel bypassing to improve consensus. Despite only presenting four solutions, there are many more such as Derecho [23], Dare [22], and others.

2.6 STORAGE PERFORMANCE DEVELOPMENT KIT (SPDK)

The performance of any storage application depends on both hardware and software. Until recently, the bottleneck induced by hardware was the major problem, while the software's overhead was ignored since it was too low when compared with the hardware limitation.

Recent breakthroughs in the technologies used by disks changed this problem. The new NVMe devices replaced the conventional SATA interface used by solid-state drives for a PCI Express interface, which connects the device directly to the CPU. This change allowed a higher data transfer rate that led to an increase of both the supported IO/sec operations and the driver's bandwidth.

As the hardware's performance increased, the available software used to control these devices needed to go along with this improvement. The SPDK toolkit appeared to address this issue.

The Software Performance Development Kit provides a set of libraries for disk access. There is a Linux NVMe driver in the Linux Kernel; however, it does not run in the userspace. The SPDK toolkit, on the other hand, has its own NVMe driver with the same functionalities, but it runs in the userspace. By skipping kernel layers, SPDK can achieve higher performance than Kernel I/Os [3]. Furthermore, it also supports network protocols that came in handy in our prototype, such as NVMe-of which extends the locally connected devices to the network. Their implementation supports many interfaces suitable for distinct environments including RDMA, TCP, and others.

This toolkit enables the development of high-performance, scalable, user-mode storage applications while also reducing the software's overhead.

2.6.1 *SPDK NVMe Driver*

The NVMe driver is the software piece that connects a NVMe device to an application. Its usage relies on three distinct phases: an initial connection, request submission, and close connection. The SPDK NVMe Driver starts the communication by establishing a link between the device and the driver and creating a set of queues (**qpair**) that handle all remaining communication. Every time a request needs to be submitted, it is placed inside those queues and later handled by the device. The driver abstracts from the user any internal communication, and both submissions and completions take place using only readings and writings on the special queues.

The *qpairs* mentioned above contain two internal circular queues, one for submissions and another for completions. To submit an operation, the user places it in the submission queue. It will later be read by the device. As soon as the device processes it, a completion request is added to the completion queue. Since this procedure is asynchronous, the user periodically reads the *qpairs* to discover new completions. Figure 1 provides an illustrative example of the interaction between the driver, a *qpair* and a NVMe device.

At the end of execution, the application frees the allocated resources. For such, the shutdown phase begins. The driver starts by destroying all *qpairs* allocated, closing their connection with the device. Finally, any existing communication left is also concluded.

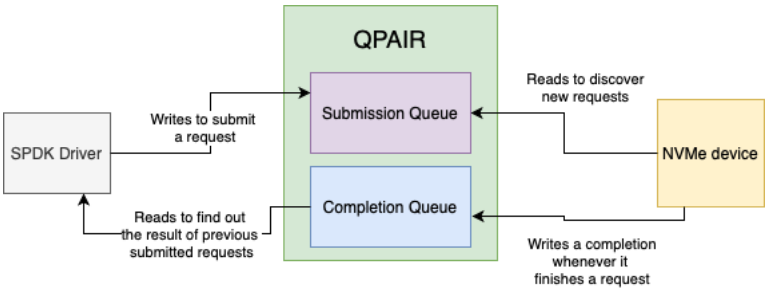


Figure 1: Example of the interaction between the driver and a device

The NVMe Devices support multiple *qpairs* simultaneously and each *qpair* may have a queue depth of up to 65,536 requests. Despite having various *qpairs* concurrently, a queue with 128 depth achieves the same performance as four queues of 32 depth [4].

2.6.2 *Performing Operations*

The supported interface for the execution of a read or write operation can be broken into two groups. The first one enables the request's submission, and the second allows the verification of the request's current state. For this SPDK offers the following functions:

The *spdk_nvme_ns_cmd_XXX* primitives on Table 1 submit a request to *qpairs*. In addition to the arguments required to execute the command, these requests receive a callback function and an object that will later be passed to the callback function when invoked. Whenever a request completes, the testing function *spdk_nvme_qpair_process_completions* invokes the callback function of the completed requests. These primitives

Function	Description
spdk_nvme_ns_cmd_read	Submits a read operation
spdk_nvme_ns_cmd_write	Submits a write operation
spdk_nvme_qpair_process_completions	Checks if there is any operation concluded

Table 1: Some of the SPDK primitives

are the only ones necessary to perform an IO operation. However, when designing a solution, it must take into account that the request submission primitives return immediately without having completed it. It may take some time for a device to execute an operation. Additionally, the testing function is a non-blocking call, and, as a result, it may be necessary to invoke it more than once before the device handles the request.

The example below presents a way to build a blocking read. In other words, the function only returns after the operation has been completed, forcing the current thread to wait.

Algorithm 2 Blocking Read

```
1:  procedure CALLBACK(obj)
2:      obj.status ← true
3:  end procedure
4:  procedure READ
5:      obj.status ← false
6:      spdk_nvme_ns_cmd_read(qpair, callback, obj)        ▷ Submits a request to a device
7:      while obj.status == false do
8:          spdk_nvme_qpair_process_completions(qpair)     ▷ Calls the callback function passed
    upon submission in case of a newly finished completion
9:      end while
10: end procedure
```

3

DESIGN

This chapter describes the conception of a state machine replication using the Disk Paxos consensus to order requests. As with any consensus protocol, the designed approach breaks the system into distinct entities to ease the understanding of the solution and simplify possible implementations. These entities, each with a specific system role, cooperate and interact together to reach the desired system's behavior.

The system is structured as three core entities: replicas, leaders, and Disk Paxos instances. Replicas are responsible for receiving the client's requests, asking leaders to sort them, and applying each decision to its current state. A leader accepts proposals from replicas and launches new consensus instances for different slot numbers. Additionally, a leader also controls the number of consensus instances running simultaneously. Having too many agreements concurrently would eventually delay each decision and compromise the system's performance. To address this issue, a leader restricts the number of consensus instances running, permitting, at any given moment, a maximum of **K** agreements simultaneously. Finally, a Disk Paxos instance is the entity that performs the agreement protocol for a given slot and input. Table 2 presents a summary for each entity.

Type	Function
Replica	• Keeping the application state • Receiving Requests from Clients • Propose Requests to Leaders • Updating the application state after each decision • Sending responses to clients
Leader	• Receiving proposals from replicas and deciding which to accept • Controlling the number of instances of consensus running • Starting DiskPaxos instances for new agreements
DiskPaxos	• Performing the consensus protocol for a given slot and input • Maintaining the current state of consensus • Notifying both leaders and replicas of a slot decision

Table 2: Core entities of the system

3.1 SYSTEM OVERVIEW

As a recap and a closer look into the cooperation between the system's entities, this section will describe how a single agreement is achieved. It will cover each step from a client sending a request until the reception of the respective response. Figure 2 presents an illustration step by step of the procedure to reach a single agreement.

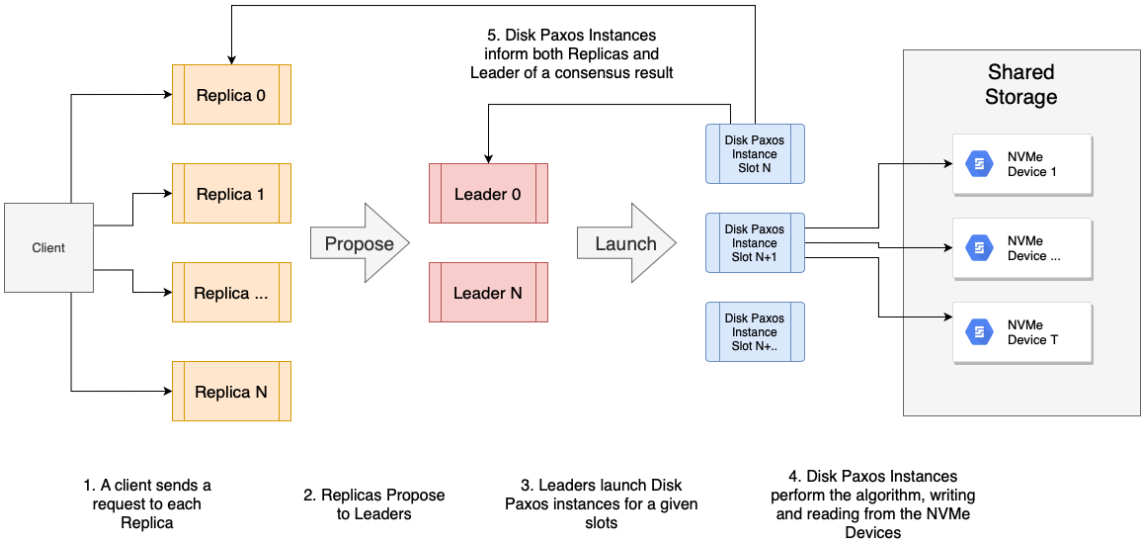


Figure 2: Overview of the entire system

The whole agreement procedure starts with the sending of a request from a client to replicas. Upon reception, replicas assign a number (slot number) to the request and propose it to leaders. As leaders receive proposals for unseen slot numbers, they launch new Disk Paxos Instances for those slots. A Disk Paxos Instance then takes the command and the assigned number and starts the consensus algorithm. To perform the Disk Paxos algorithm, each instance executes multiple writings and readings on each storage device in the network. Whenever the algorithm achieves a consensus for its slot, the instance disseminates the decision across leaders and replicas and stops its execution. The replicas then compute the response and send it to the respective client.

The system's design and Figure 2 do not specify how the distinct actors communicate with each other. They only provide a high-level view of the stages each request goes through. The technologies and software of today offer a wide variety of solutions to enable communication between processes. Some of these solutions are communication through the TCP Protocol, RDMA, or even using shared storage to propagate information across. This system's property is left open, so it doesn't restrict a future system's implementation.

3.2 REPLICA

Replicas are the entry point for any request in the system. They are responsible for handling incoming requests from the system's clients, forwarding them to the leaders, and eventually returning a response to a client. Replicas run applications and compute new applications' states as each new command is decided. Upon a request arrival, a replica process assigns the next available slot and sends a proposal to the leader using both the command to execute and the assigned slot number. Since a proposal does not always end up in a decision with a distinct command, a replica keeps track of its ongoing proposals. It retransmits them if it finds out that a decision for a proposed slot resulted in a distinct command or if they timed out. When a replica receives a decision, it updates its application state and replies to the respective client. Our replica's tasks are similar to those described in [2], however ours do not address system changes.

This approach requires two system properties. Each client must broadcast its requests to each process, which means that different replicas may receive each message in a distinct order, and some proposals may contain the same command for different slots. Whenever a Disk Paxos process broadcasts a decision to replicas, at the reception, each replica must verify if the command decided has already been processed in a previous slot. Replicas ensure that each operation is only applied once by comparing it with the previous ones, which leads to the second property. Each request must be identified by a pair of integers containing the client and the request identifiers. At the arrival to the system, a replica fills the client's id of a request, while the client is responsible for ensuring the command identifier.

3.3 DISK PAXOS INSTANCES

The Disk Paxos Instances are the system's entities responsible for accomplishing the distributed agreement. Each instance performs the *Disk Paxos* algorithm on an assigned slot number and notifies the system when it achieves a decision. The sections ahead describe the disk sections used to reach an agreement as well as changes made to the original *Disk Paxos* to support the proposed features.

3.3.1 *Disk Structures*

Disk Paxos relies on writing and reading storage devices to reach an agreement. In the original description, the algorithm uses a vector of structures kept in disk for each decision. Each vector's entry represents a reserved space for each process in the system. A process only writes on its reserved sector, avoiding concurrent updates.

Figures 3 and 4 present an example of how a single row and a matrix with multiple decisions, respectively, look like in a disk. As multiple decisions are achieved, the structure kept may be seen as a matrix where each row represents a different decision. Whenever the system needs a new agreement, it adds a new row and performs the algorithm on it.

Process Blocks					
0	1	2	3	...	N
inp: bal: mbal:	inp: bal: mbal:	inp: bal: mbal:	inp: bal: mbal:	...	inp: bal: mbal:

Figure 3: An example of a row of blocks.

	Process Blocks					
Slot nº	0	1	2	3	...	N
0	inp: bal: mbal:	inp: bal: mbal:	inp: bal: mbal:	inp: bal: mbal:	...	inp: bal: mbal:
1	...	...	...	...	...	...
2	...	...	...	...	...	...
...	...	...	...	...	...	...
N	inp: bal: mbal:	inp: bal: mbal:	inp: bal: mbal:	inp: bal: mbal:	inp: bal: mbal:	inp: bal: mbal:

Figure 4: Disk Paxos Original block matrix.

As new decisions appear, the structure grows, resulting in an unbounded increasing log in each storage device. In addition to the unnecessary space occupied, an endless log would make it difficult and costly to recover from failures since a process doesn't know where the last row added was. Moreover, new rows would, potentially, require an initialization procedure.

The limitation over the number of consensus executing only requires a total of **K** rows to support the algorithm. A matrix of fixed size **K** enables the required space for consensus, while, at the same time, eases the initialization procedure and eliminates the unbounded log. For such, each row needs to be reused across multiple decisions, raising two concerns.

The first problem is the assurance that different processes attempting to decide the same slot number target the same row. The solution for this issue relies on mapping the Disk Paxos Instance to a row using the remainder after dividing the assigned slot number by **K**. The second issue is the garbage left by previous protocol executions. By adding a fourth field to each block responsible for keeping track of which slot the block refers to, the algorithm can discard the ones that don't refer to the same agreement.

	Process Blocks					
Slot nº	0	1	2	3	...	N
0	inp: bal: mbal: slot:	inp: bal: mbal: slot:	inp: bal: mbal: slot:	inp: bal: mbal: slot:	...	inp: bal: mbal: slot:
1	...	...	...	...	...	...
2	...	...	...	...	...	...
...	...	...	...	...	...	...
K-1	inp: bal: mbal: slot:	inp: bal: mbal: slot:	inp: bal: mbal: slot:	inp: bal: mbal: slot:	inp: bal: mbal: slot:	inp: bal: mbal: slot:

Figure 5: Structure for the designed approach.

Figure 5 shows an example of a structure in a disk. While performing the consensus for a proposal, the algorithm only considers blocks with the same slot number. If it finds a higher slot number then the current proposal already took place and is canceled. Lower slots are ignored since they refer to previous decisions.

3.3.2 *Changes to Disk Paxos*

The disk structure designed in the previous section differs from the one presented in Disk Paxos. The existing difference requires modifications to the original Algorithm 1 (described in Section 2.4) since the used blocks introduced a new field, slot identifier. The slot identifier saves information about the last slot that used that block. This change enables the reuse of the same disk sectors for distinct instances since a process can distinguish the blocks relevant to the current execution from the blocks left from previous executions.

Designing this change required only modifications in the analysis of the sectors read during the writing and reading phase. Algorithm 3 presents a version of Disk Paxos supporting the introduction of the slot number in each block. There are three distinct scenarios. Either a read block has a slot field with a lower, equal, or higher slot number than the assigned slot to the current instance computing (lines 7 to 15). A lower value means that the block refers to a previous instance and should be discarded. If the number is equal, it belongs to the current iteration and must be taken into account. Finally, the last scenario is a higher slot, which indicates that another process has already completed this decision. In this case, the current agreement is outdated and should no longer continue executing. The rest of the algorithm is unchanged.

This change requires that a structure row with index *i* only handles consensus instances with slot numbers computed by $i + K * N$. Additionally, an agreement for slot $i + K * (N + 1)$ can only start after finishing the one with slot $i + K * N$. These two properties ensure that instances with the same assigned slot target the same disk sectors.

Algorithm 3 Changes to Classic Disk Paxos

```
1:  procedure PHASE1OR2(p,phase,s)
2:      index ← s mod K
3:      repeat concurrently for d ∈ disks
4:          disk[d][index][p] ← dblock                ▷ Writes dblock at disk d, row index, block p
5:          for q ∈ processes, q ≠ p do
6:              blockQ ← disk[d][index][q]            ▷ Reads block q in row index of disk d
7:              if blockQ.slot <= dblock.slot then
8:                  if dblock.mbal >= blockQ.mbal ∧ dblock.slot == blockQ.slot then
9:                      blocksSeen ← blocksSeen ∪ {blockQ}
10:                 else if dblock.slot == blockQ.slot then
11:                     Abort() ∨ StartBallot()       ▷ Abort if it thinks it's no longer the leader
12:                 end if
13:             else
14:                 Cancel()
15:             end if
16:         end for
17:     until Has completed for a majority of disks
18:     continue
19: end procedure
```

3.3.3 *Ballot numbers, Leader Election and Abort*

Disk Paxos requires a single leader to make progress. Its safety and liveness properties also enable the algorithm to perform under circumstances where the system has multiple leaders at the cost of performance. The existence of many leaders simultaneously will cause abortions and restarts using higher ballot numbers, delaying an agreement. Usually, multiple leaders are active while a leader election is running. After its conclusion, the system converges to having a single leader. For this to happen, the system requires a leader election mechanism in parallel with the consensus algorithm since *Disk Paxos* does not address this issue. Instead of requiring a leader election in the background, the Disk Paxos algorithm may take advantage of how ballot numbers are generated.

When a Disk Paxos instance is running, it might abort if it finds a block with a higher-numbered ballot. In this case, this means that another process is also trying to decide on the same slot number, and the process has two options. Either it quits in case of no longer being a leader or starts a new attempt, this time with a higher ballot. By taking advantage of how the ballot numbers are generated, it is possible to identify process owning a ballot. The algorithm establishes that the process with the higher identifier should always lead. Through analysis of each ballot upon abortion, a process discovers if it has a higher identifier. By doing so, it figures out if it is still a leading candidate and should keep executing. For such, a Disk Paxos instance generates its ballots using the formula

$ballot = id + N \times i$ where the id, i, and N represent a unique identifier from 0 to N - 1, an increasing integer, and the number of processes, respectively.

The abort function can be written as follows in Algorithm 4. By computing the remaining between the division of the ballot that originated the abortion and the number of processes (line 2), an instance discovers if its identifier is higher than the ballot's one. Assuming that the higher identifier should lead, an instance can then decide if it should restart the algorithm (line 6) or quit due to the existence of a better candidate (line 4). Eventually, there will only be one leader executing.

Algorithm 4 Abort together with leader election

```
1: procedure ABORT(ballot)                ▷ Ballot number that originated the abortion
2:     id ← ballot % N                    ▷ Remaining of Integer Division
3:     if id > pid then                   ▷ Comparison between the id found the instance identifier
4:         exit()                         ▷ Stops executing
5:     else
6:         StartBallot()                  ▷ New attempt with a new ballot
7:     end if
8: end procedure
```

Algorithm 4 explains how a leader gives up leading. New leaders appear when a replica suspects of a failure. It then launches a new leader, which executes until it aborts. The failure detection can be achieved via a proposal that timed out.

3.4 LEADER

Leaders are the entities responsible for managing the consensus instances. A Leader handles proposals and starts new executions of the agreement protocol for distinct slots and inputs. As part of its responsibilities, it must ensure that, at most, there are at most **K** consensus instances running.

Algorithm 5 presents a summary of the leader's behavior using pseudocode. The mapping and queuing system are presented between lines 12 to 19, while the freeing mechanism is on lines 21 to 28.

Upon a proposal arrival, a leader verifies if it has already seen the target slot in a previous proposal (line 9). Each new slot number discovered results in a new agreement execution. A leader may see various proposals targeting the same slot number since each replica proposes once for each slot. The leader only accepts the first to arrive and discards the subsequent ones. For such, each leader maintains a map with the slots seen and their respective commands.

After a proposal acceptance, a leader checks if the system can run a new Disk Paxos Instance. This verification is due to the requirement for not exceeding the maximum **K** instances allowed. Additionally, each Disk Paxos Instance targets a specific disk section, and the system must ensure that distinct leaders running instances for equal slot numbers target the same disk sectors. As mentioned in Section 3.3, each storage device has **K** regions reserved for the consensus computation. A leader maps a consensus instance to a section using the remainder

of the division between the slot number and the value of **K**. This mapping ensures that the same slot number in distinct processes always targets the same sectors. However, a new issue appears. To guarantee that there is always a decision accomplished for each instance launched, each disk region may only have one consensus protocol writing and reading at the same time from each leader.

Instances with different slots but targeting the same sectors inside the same leader need to be queued. In addition, each leader keeps track of the state of the regions reserved. As a new proposal is ready to launch, the leader verifies if the corresponding storage sector is free (line 14). In that case, a new Disk Paxos Instance starts the consensus algorithm; otherwise, it enqueues the proposal until the storage sector is freed (line 18). At the end of execution, each consensus instance notifies the leader freeing the storage sector. Upon being notified, the leader searches for pending proposals queued and starts new instances.

Algorithm 5 Leader Process Workflow

```
1:  procedure LEADER(K)
2:      proposals ← ∅
3:      slots ← [ true  for i ← 0 to K − 1]
4:      queue ← [∅ for i ← 0 to K − 1]
5:      while true do
6:          e ← event()
7:          Switch e
8:              case < PROPOSE, s, c >:
9:                  if s ∈ proposals then
10:                     break
11:                 end if
12:                 proposals[s] ← c
13:                 slot_in ← s mod K
14:                 if slots[slot_in] then
15:                     slots[slot_in] ← false
16:                     spawn(DiskPaxos(s, c))
17:                 else
18:                     queue[slot_in] ← queue[slot_in] ∪ {< s, c >}
19:                 end if
20:             end case
21:             case < FREE, s, c >:
22:                 slot_in ← s mod K
23:                 if queue[slot_in] == ∅ then
24:                     < s1, c1 >← queue[slot_in][0]
25:                     spawn(DiskPaxos(s1, c1))
26:                 else
27:                     slots[slot_in] ← true
28:                 end if
29:             end case
30:         end switch
31:     end while
32: end procedure
```

4

IMPLEMENTATION

The following chapter presents the transition from the design to implementation. First, it introduces the core aspects of the SPDK toolkit that shaped the solution. These aspects dictate how the application handles storage devices. They are key aspects to understanding how to submit requests to the shared storage. Second, the chapter covers all the implementation decisions that influence the interaction between distinct components. Moreover, it presents the architecture of the system and provides a detailed explanation of its components' implementation and behavior. The system was developed using C++17 since it was the language that provided the most features while it made it easier to manage the SPDK's C interface.

4.1 SPDK: CORE CONCEPTS

The SPDK toolkit is the set of libraries to communicate with the storage devices. It provides not only an abstraction to the end-user of the connection between the application and storage devices but also a simple interface to submit and handle IO NVMe requests. The NVMe driver presented the flow required to complete any IO request in Section 2.6.1. Unfortunately, the application couldn't rely on blocking IO operations such as Algorithm 2; otherwise, it would have its performance diminished. Due to the driver's workflow, the workaround is implementing IO requests based on an event approach. This decision requires the introduction of an event framework to support the asynchrony of the NVMe Driver.

4.1.1 *SPDK Event Framework*

SPDK provides an event framework to write fully asynchronous applications. This framework runs entirely in the background and is written in a single-threaded faction. Internally, the application launches multiple threads, all pinned to different CPU cores. It is important to avoid sharing resources between the existing workers otherwise, a thread might block waiting for a resource leading to a decrease in the performance.

Each thread in the framework has its event pool. The user only has to create events and submit them. They will, eventually, be handled, enabling the asynchronous flow expected. Table 3 presents the two primitives available to create and submit events. To create an event, the user invokes *spdk_event_allocate* passing as arguments a function, an object with data for the function and the core where the event should be schedule. After the event

creation, the user calls *spdk_event_call* to submit the event to the framework. Internally, the event framework schedules the event to run in the desired processing unit.

Function	Description
spdk_event_allocate	Allocated an event for a given core
spdk_event_call	Submits an event

Table 3: Available interface for the Event Framework

The read operation presented at Algorithm 2 blocked the current thread until the IO request was completed. Algorithm 6 provides a read operation using an event approach as an improved alternative. Despite working as an example, this approach was the one adopted to perform the IO requests on the storage devices.

The solution breaks the function into two events: the first one submits the IO request to the NVMe device (line 4); the second one handles the verification of the request's current state (line 10). The user starts by creating an event to submit an IO request (line 19). The event framework then allocates it to a specific core with the usage of *spdk_event_call*. Upon handling it, the respective thread submits an IO request to the NVMe device (line 6). The next step would be to run a while loop until the request has been completed, however, an event cannot block a thread; otherwise, it compromises the application's performance. The while loop is replaced by creating a new verification event (line 10) and submitting it (line 7). This event verifies if the request has been completed just once (line 14) and allocates a new one recursively (line 16) until the IO request is completed. Additionally, any scheduled events regarding the same operation are always allocated to the same core since the qpair structures of NVMe's Driver are not thread-safe. This property enforces each thread to have its non-shareable qpairs for the purpose of avoiding any communication or coordination between them.

This approach allows the execution of other parallel calls without compromising their performance.

Algorithm 6 Event-Based Read

```
1:  procedure callback(obj)
2:      obj.status ← true
3:  end procedure
4:  procedure event_read                          ▷ Runs at Event Framework
5:      obj.status ← false
6:      spdk_nvme_ns_cmd_read(device, callback, obj)      ▷ Submits a request to a device
7:      event ← spdk_event_allocate(core, event_verify, obj)   ▷ Creates an event to verify
    the request state
8:      spdk_event_call(event)                            ▷ Submits an event
9:  end procedure
10: procedure event_verify                        ▷ Runs at Event Framework
11:     if obj.status then
12:         return
```

```
13:     else
14:         spdk_nvme_qpair_process_completions(qpair)
15:         event ← spdk_event_allocate(core, event_verify, obj)   ▷ Creates an event to
    verify the request state
16:         spdk_event_call(event)                                  ▷ Submits an event
17:     end if
18: end procedure
19: procedure read                                                  ▷ Called by the user
20:     event ← spdk_event_allocate(core, event_read, obj)   ▷ Creates an event to submit a
    read
21:     spdk_event_call(event)                                      ▷ Submits an event
22: end procedure
```

4.1.2 *Network Storage*

NVMe devices are connected via the PCI Express interface to the CPU, making them available only locally. The NVMe over Fabrics (NVMe-oF) is an extension to the NVMe protocol that connects hosts to storage devices over the network. It enables data transfer through Ethernet, Fibre Channel (FC), InfiniBand, and others. The network storage is a set of NVMe devices using the NVMe-oF to be accessed remotely.

The SPDK libraries provide an NVMe-oF implementation together with an application to configure devices. The storage device's network is established by running this application in each device and configuring them to listen to a specific port and IP address. The devices would integrate a NVMe-oF Subsystem, and any host wanting to connect would only connect to the NVMe-of's discovery subsystem. This subsystem would reveal every device connected to the system. Any NVMe-oF, regardless of how many devices it has, has a discovery subsystem.

The NVMe-oF is transparent to the end-user, so to connect either to a local device or a device over the network, the only change required is the connection string used upon the initialization of the NMVe driver. In the case of a NVMe-oF, the connection string refers to the network's discovery subsystem.

Given the above, the system's storage is composed of a group of SPDK NVMe-oF applications. Each application connects to a local NVMe Device and enables remote access through the TCP protocol. TCP presents the most versatile and the less hardware-dependent protocol from the ones available. For this reason, it is the best choice among the available options since it is easier to configure and more suitable for cloud environments.

4.2 APPLICATION STRUCTURE

Chapter 3 presented two core entities of the architecture, Replicas, and Leaders. Additionally, Section 4.1.1 introduced the necessity for an event framework to enable IO requests using events. We will refer to this one as "SPDK Event App" (SPDKEA) in the sections ahead. The prototype is composed of these three components, each with a distinct task.

Figure 6 pictures an overview of the application structure for the implementation of each process. As shown, Replica and Leader execute as threads inside the same process. While the Replica is always active, the Leader is only running while a process believes it is leading. The implementation runs the Replica in the process's main thread and spawns additional threads for the Leader. Thus, it is easier to control when the Leader should be active. On the one hand, whenever a process supposes it is no longer leading, the Leader quits and stops executing. On the other hand, if the process believes it should lead, it spawns a new thread and the Leader restarts.

Lastly, the "SPDK Event App" (SPDKEA) is a component that runs in the background and is independent of the others. It has internal threads that run pinned in distinct CPU Cores. Each internal thread has an individual event pool to handle incoming events. All the communication with the storage devices goes through this component. Both Replica and Leader communicate with this component via event allocation. As their workflows generates events, the "SPDK Event App" handles them internally in a asynchronous approach.

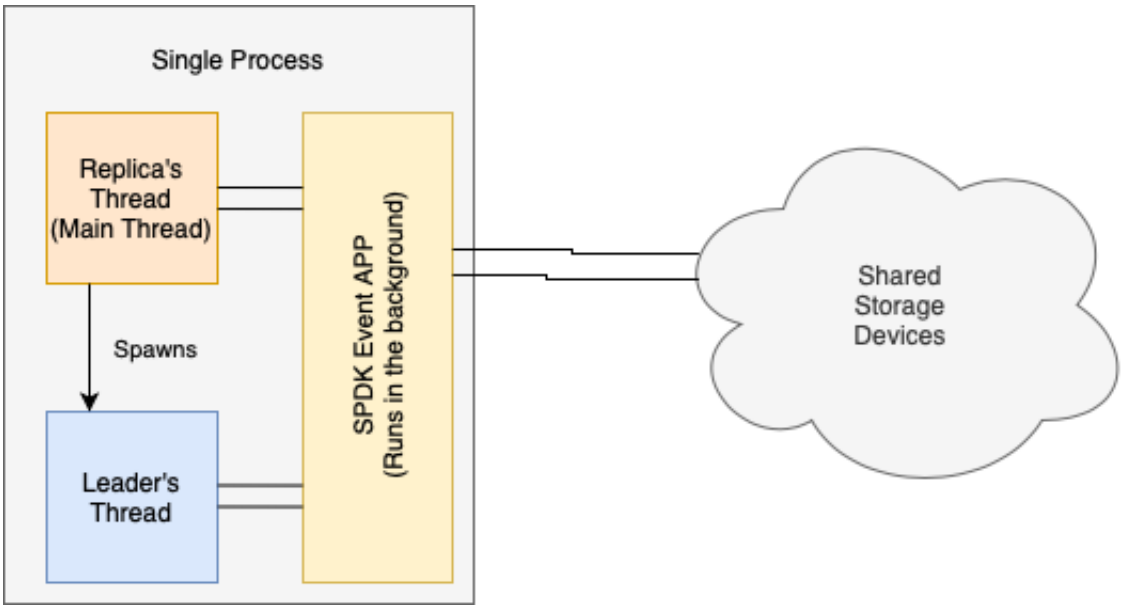


Figure 6: The application structure for a single process

4.2.1 *Communication between Process*

Despite the most common pattern for communication between processes being a message-based approach, we implemented a communication system using the storage devices. Instead of relying upon messages to

communicate, processes submit IO requests to the network storage to share information. These requests operate over specific regions of each storage device, and each region has a distinctive function.

A Disk-based Communication system requires the processes to read and write to the shared storage. The two procedures that require exchanging information between processes are the proposals and the decisions. As such, the procedure of proposing and the propagation of a decision are achieved via writing and reading, respectively. Each time a process wants to propose, it simply writes in the intended section of each disk. On the other hand, to discover a new decision, a process reads the decision segment of each disk. As a result, two distinct sections were added to the storage devices to support this approach, one for proposals another for decisions.

4.2.2 *Structures in Disk*

Any structure kept in disk must take account of any limitations or features available in the SPDK. For instance, an operation can read multiple contiguous logical sectors in a single request. Similarly, an operation can also write in a section of contiguous sectors. Consequently, the structures should be optimized to reduce the IO operations submitted.

Firstly, the consensus algorithm only writes on one sector but reads from many in a single phase. Ideally, it could be accomplished using only two requests, a writing which operates only in an individual region and a reading which retrieves all sectors at once. For such, the blocks must be contiguous in a disk. The properties are fulfilled by mapping each entry of the consensus blocks to a logical sector sequentially.

Secondly, the proposal's region has a different purpose. It must support writing in isolated sectors so a process may propose, but it should also enable a leader to read every existing proposal for a given slot. As a result, the solution was to dedicate, for each slot, a logical sector for each process. Consequently, this requires an unbounded log due to the unlimited slot numbers supported. Nevertheless, a single process can write in a region without having concurrency issues and a leader can read all proposals for a specific slot using just one request.

Lastly, the decision's region only requires a section of each decision completed. Despite the possibility of having multiple leaders writing in the same sector, they will all write the same result due to the algorithm's correctness. Like the solutions presented above, the decision's region ends up being an infinite increasing log where consecutive decisions are contiguous in a storage device. This approach also takes advantage of discovering many decisions with a single request.

Ultimately, these structures are organized in each disk as shown in Figure 7: the consensus blocks start at the logical block address (LBA) 0 and occupy the subsequent N (Nº Processes) $\times$ L (Nº of Maximum Consensus Instances) sectors; the decisions region follows filling the following addresses with decisions, a decision per sector; by last, the proposals region starts right after adding a giant offset to where the consensus blocks end. The offset value is big enough to ensure that the decisions region doesn't overlap the proposal's space. The blocks of Figure 7 are mapped row-wise into the disk sectors.

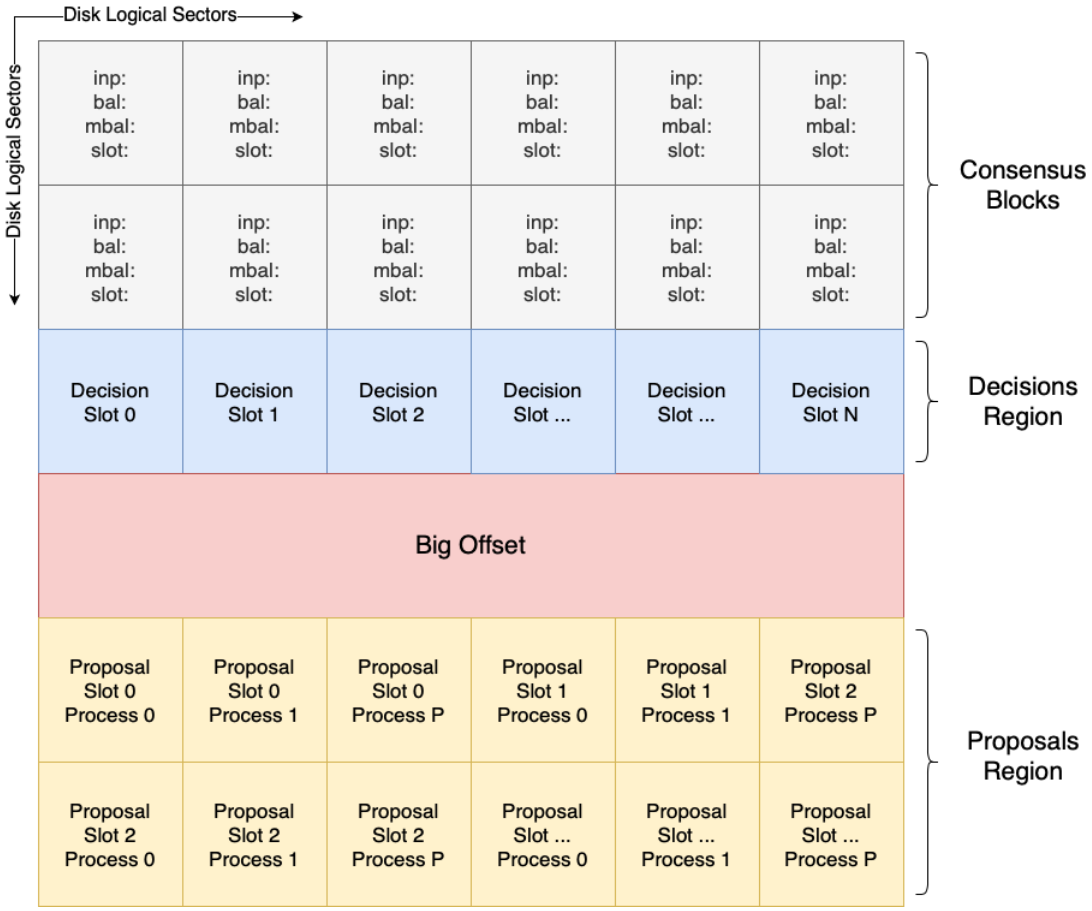


Figure 7: Visualization of disk structures

4.2.3 *Data Serialization*

NVMe devices split their storage into sectors. These sectors can be configured to have higher and lower sizes depending on the application's needs. The most common configuration is 4kiB sectors being this, the selected size for the network devices. This decision meant that each reading or writing operation results in the transfer of data buffers whose size is multiple of 4kiB. In addition, an object to be written or read must be serialized or deserialized.

The library used for the serialization procedure was Cereal, a C++11 library. It provides a fast, minimal, and easy to integrate solution. Its low-level implementation makes it suitable to integrate with the low-level interface provided to communicate with NVMe Devices.

Despite transferring at least 4kiB per request, a serialized object may not occupy the entire data buffer. It would be inefficient for the unmarshalling procedure to use the full-sized buffer in these scenarios. For such, when writing an object into a data stream, a header with the object's size is added together with the serialized object. The unmarshalling process only uses the bytes that are useful using the header's information. Additionally, each disk sector only keeps a single structure at once.

Figure 8 presents a graphical view of the buffer organization for a IO request. To write X consensus blocks, the operation requires an $X \times 4kiB$ size buffer, and each block is serialized and written into a multiple of 4kiB index, starting at zero. For instance, the first block goes onto index 0, the second to 4096, the third to 8192, and so on. When handling the request, the device splits the buffer into 4kiB pieces and maps each to an individual sector. The operation will overwrite the X next contiguous sectors starting at the specified logical block address.

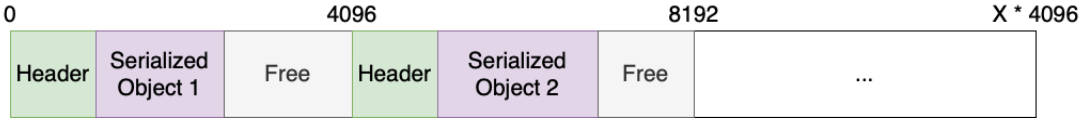


Figure 8: Buffer Organization for a IO request

Every IO request requires a serialization step. If the request is a writing, then the data serialization precedes the request's submission. On the other hand, readings have a deserialization step after the request completion.

4.3 OVERVIEW OF THE SYSTEM'S IMPLEMENTATION

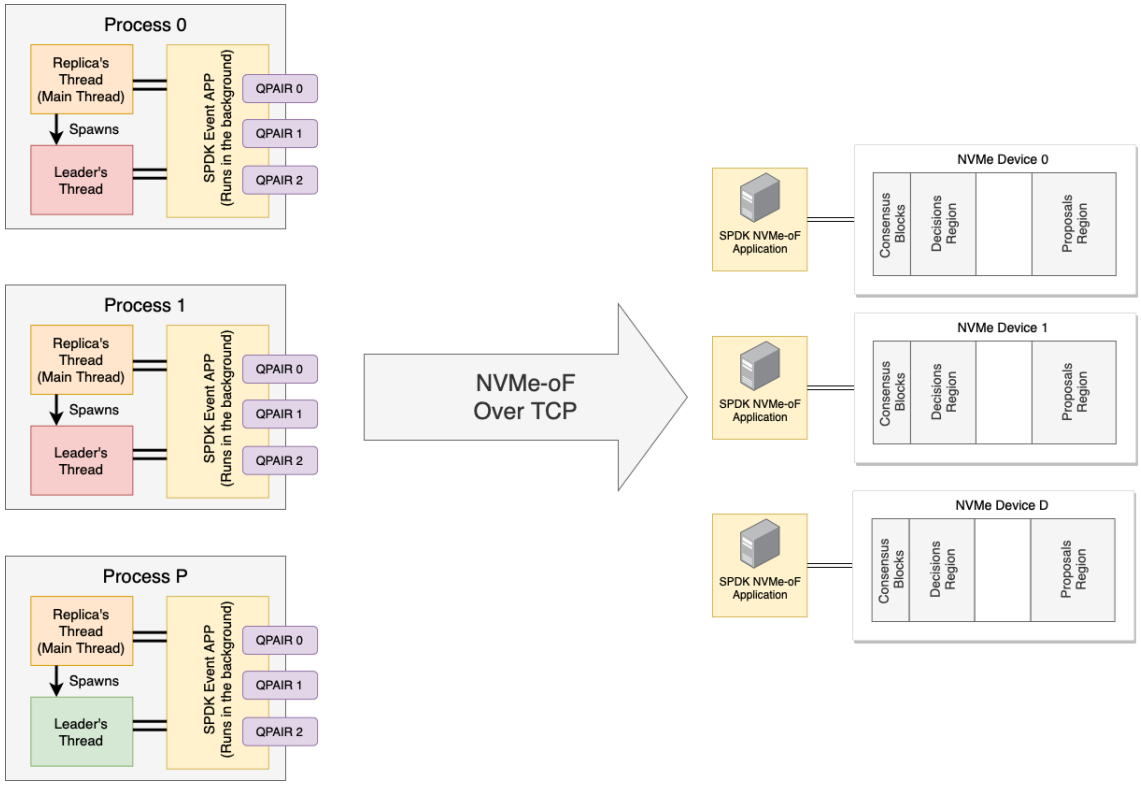


Figure 9: Overview of the implemented system

The overall view of the system splits it into two, the system's processes and the network storage. As depicted in Figure 9, the network storage is composed of multiple SPDK NVMe-oF Applications connected to NVMe devices and accessible remotely via the TCP Protocol. The system's processes play the distinct roles required to reach an agreement. Each process has a main thread performing the Replica's task, a thread for the Leader's job that is only active in the leading processes, and a background event application (SPDKEA) to communicate with the storage using the SPDK NVMe Driver.

4.3.1 *Overview of an agreement*

Figure 10 presents the steps and the interactions between the distinct components required to reach a decision.

An agreement starts with a process having a command to propose. For such, its main thread assigns a slot number available and allocates an event to write the command in the storage (step 1). Inside the same process, the SPDKEA component handles the event. For each device, the SPDKEA component writes the command in the proposal space in the reserved sector for the assigned slot and process (step 2). Simultaneously, the leader's thread inside the system's leading processes actively reads the proposal space of each disk (steps 4,5). Upon discovering a proposal for an unseen slot number, the thread dedicated to the leader tasks allocates an event to compute the consensus algorithm. The consensus algorithm runs entirely inside the SPDKEA component using the chaining of events via the callback of IO operations. In the end, the decision accomplished is written in the respective sector over the decisions reserved space. Similar to the proposal's discovery mechanism, the main thread discovers new agreement results by reading the decisions space in each device (steps 10, 11, 12).

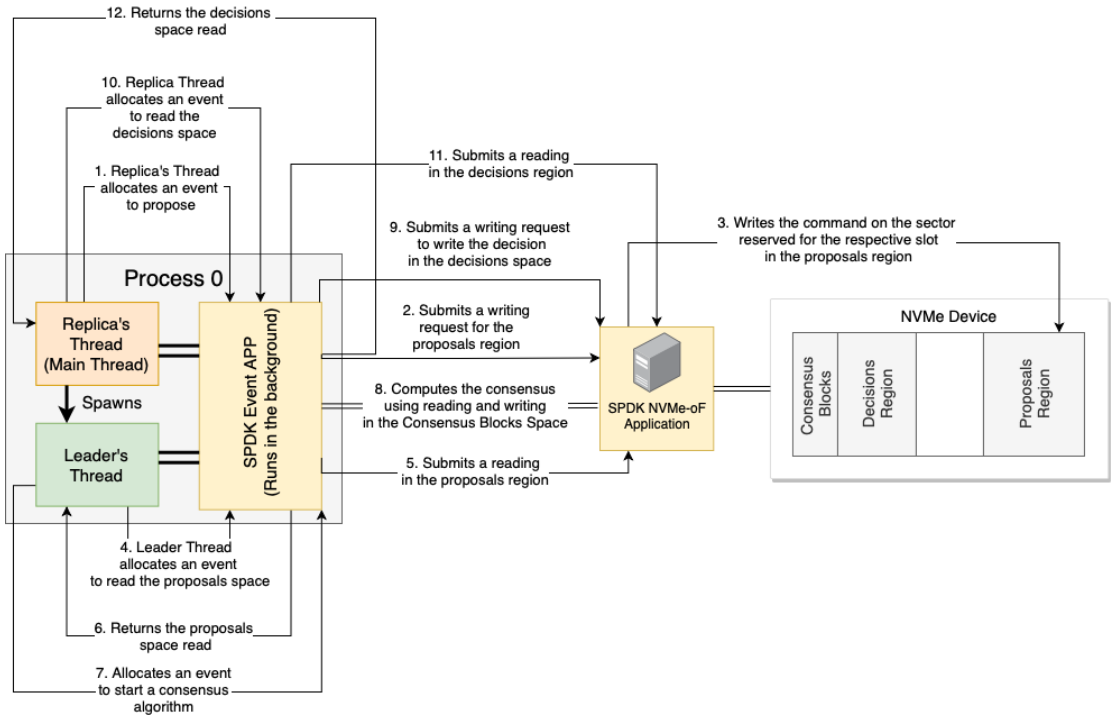


Figure 10: Steps needed to achieve a single agreement

Any IO request is always accomplished via the SPDKEA component. Both the replica and leader threads allocate events to perform operations in the network storage. The SPDKEA handles these events and submits the operations for them. Eventually, it returns a response.

4.4 COMPONENTS

4.4.1 *Replica*

The Replicas receive requests from clients, propose commands to a Leader, and inform clients of the decisions. Due to the experimental environment where we implemented our system and taking into account the purpose of this work, we decided not to implement the reception of requests for clients. Instead, a mechanism to simulate a client request is used. At the start of execution, each process has a list of instructions in a distinct order which it uses to propose. The replica ends its execution when it receives a decision for each proposed command.

The two core concepts important to understand about a Replica are the proposal and discovery mechanisms. Algorithms 7 and 8 illustrate how a Replica uses events to communicate with the SPDKEA component in order to propose to Leaders and discover new decisions.

Firstly, to propose, a replica periodically reads its list of commands and submits proposal events to the SPDKEA. These events will write on each storage device (lines 14 to 20 of Algorithm 9). Each proposal results in a single IO request per device, and it is completed using the same method as Algorithm 6. The replica's thread creates an event and allocates it to one of the SPDKEA's threads (procedure at 14). For each disk, the handling thread submits a writing request and launches a verification event (lines 4 to 13 of Algorithm 7). The request's callback function only controls if the request has finished. Given a proposal with a slot number N, the logical block address (LBA) where the SPDKEA writes is computed using the formula below:

$$LBA = P \times LANES + OFFSET + N \times P + Pid$$

P , number of processes
LANES , number of consensus instances
Pid , process id

Algorithm 7 Proposal Mechanism

```
1:  procedure WRITE_CALLBACK(obj)                          ▷ Callback function of IO request
2:      obj.status ← true
3:  end procedure
4:  procedure HANDLEPROPOSAL(obj)                          ▷ Internal Proposal Event
5:      lba ← P × LANES + OFFSET + obj.slot × P + pid
6:      for d ∈ disks do
7:          qpair ← qpairs[core][d]
8:          opt ← {obj ← obj, disk ← d, status ← false }
9:          spdk_nvme_ns_cmd_write(qpair, serialize(obj), lba, 1, write_callback, opt)
10:         e ← event(event_verify, opt, core)
11:         submit(e)
12:     end for
13: end procedure
14: procedure PROPOSE(cmd,slot)                            ▷ Interface for Replica's Thread
15:     core ← nextCoreReplicas()
16:     p ← Proposal(cmd, slot)
17:     e ← event(handleProposal, p, core)
18:     submit(e)
19: end procedure
```

Secondly, the disk-based proposal system also requires a discovery mechanism to find out decisions as they are decided. This mechanism is always running and uses the Future API to share data between components (procedure *Search*, line 1). It starts with the replica's thread creating an event to read the decision's region and submitting it to the SPDKEA component (prodecude at line 27). This event has a promise object that points to a Future object owned by the main thread. When the reading finishes, the SPDKEA's internal threads complete the promise sharing data between the replica's and the event's components. The shared data is the result of merging the readings from a majority of disks into a map of slot numbers and their respective decision (*read_callback*, line 1). This routine discovers multiple decisions at once, and whenever it finishes, a new one starts again from the last known decision. The formula below computes the starting index of the logical block where the read operation starts. Since the decisions are contiguous in a disk, the read operation reads the following K sectors after the starting index to discover K decisions.

$$LBA = P \times LANES + N$$

P , number of processes

$LANES$, number of consensus instances

Algorithm 8 Decision Discovery Mechanism

```
1:  procedure READ_CALLBACK(opt)                          ▷ Callback function of IO request
2:      opt.status ← true
3:      if opt.dec.status == false then
4:          dec ← opt.dec
5:          dec.diskSeen ← dec.diskSeen ∪ {opt.disk}
6:          for i = 0; i < dec.size; i = i + 1 do
7:              if slot + i ∉ dec.data & valid(opt.buf[i]) then
8:                  dec.data[slot + i] ← deserialize(opt.buf[i])
9:              end if
10:         end for
11:         if |dec.diskSeen| > |disks|/2 then
12:             dec.status ← true
13:             dec.promise(dec.data)
14:         end if
15:     end if
16: end procedure
17: procedure HANDLEDECISIONS(dec)                        ▷ Internal Decision Discovery Event
18:     lba ← P × LANES + dec.slot
19:     for d ∈ disks do
20:         qpair ← qpairs[core][d]
21:         opt ← {dec ← dec, disk ← d, status ← false }
22:         spdk_nvme_ns_cmd_read(qpair, opt.buf, lba, dec.size, read_callback, opt)
23:         e ← event(event_verify, opt, core)
24:         submit(e)
25:     end for
26: end procedure
27: procedure DISCOVERY(slot,size)                        ▷ Interface for Replica's Thread
28:     core ← nextCoreReplicas()
29:     dec ← Decision(slot, size)
30:     fut ← Future()
31:     dec.promise ← fut.promise()
32:     e ← event(handleDecisions, dec, core)            ▷ Creates an event for SPDKEA
33:     submit(e)                                        ▷ Submits a event
34:     return fut
35: end procedure
```

To summarise, Algorithm 9 covers a general view of the replica's work. The replica's thread simulates the reception of a request by reading from a local list. Then, it proposes to a leader at a fixed rate using writings on disk. Moreover, it has a background mechanism to discover decisions that relies on Futures to share data across components. The process quits when there is a decision for every proposal submitted.

Algorithm 9 Replica's Behaviour

```
1:  procedure SEARCH(fut,decisions)                    ▷ Procedure to discover new decisions
2:      if fut == null then
3:          fut ← discovery(decisions.size(), T)       ▷ Submits an event to read the decisions sectors
4:      else if fut.state == ready then
5:          map < Slot, Decision > ← fut.data
6:          update_decisions(decisions, map)           ▷ Updates the structure saving the information about
    the known decisions
7:          fut ← discovery(decisions.size(), T)       ▷ Submits an event to discover new decisions
    again
8:      end if
9:  end procedure
10: procedure REPLICA(Lanes)
11:     future ← null
12:     decisions ← ∅
13:     slot ← 0
14:     while true do
15:         p ← readfile(i)                            ▷ Reads local file to find a new proposal to propose
16:         propose(p, slot)                           ▷ Submits an event to write p in proposal sector i
17:         Search(future, decisions)                  ▷ Discovers new decisions
18:         wait(Rate)
19:         slot ← slot + 1
20:     end while
21:     while slot ! = decisions.length do
22:         Search(future, decisions)                  ▷ Discovers new decisions
23:     end while
24: end procedure
```

4.4.2 *SPDK Event App*

The SPDK Event APP (SPDKEA) is a background application to support asynchronous routines, handling any communication with the storage and it is built on top of the event framework mentioned on Section 4.1.1. Its usage requires an initial call to launch the application and an exit call to release the allocated resources. The process's main thread invokes these two calls right before starting and when ending its execution. After its initialization, any communication between other components and SPDKEA is done by allocating and submitting events to a specific core. Internally, the SPDKE will schedule the events to the thread pinned on the intended processing unit.

Most events require the submission of IO operations to a device. The SPDKE establishes and manages the communication with the storage devices. It operates in a multi-threaded environment, and as a result, its usage should avoid sharing resources between threads. Additionally, the NVMe qpairs mentioned in Section 2.6.1 are not thread-safe and only support point-to-point communication. In other words, a qpair can only be connected to a single device at any given moment. This restriction allows the usage of three distinct approaches to communicate with the remote disks:

1. There is only a main thread managing the qpairs and submitting requests to the storage devices. Whenever a thread wants to submit a request, it messages the main one, which will take the request from that point (Message-based Approach)

2. A global structure that is shared among threads using a lock mechanism

3. Each thread has a qpair connected to each disk

Both approaches 1 and 2 rely on fewer qpairs to communicate. However, a message-based solution increases the load on a single thread and has a single point of failure. A lock-based solution introduces shared resources and may cause some threads to block waiting for a resource to be released. The third approach relies on neither locks nor messages, each thread is independent of the others.

Given the reasons above and since the SPDK specification states that a big qpair achieves the same performance as multiple smaller qpairs, this component implemented the third approach. Each thread has a qpair dedicated to each storage device in the network and handles its submitted requests. Consequently, if an event generates child events, they must be computed by the same thread as the parent was for the only purpose of avoiding sharing resources.

Finally, this approach might be seen as an application with multiple workers since each thread can perform the same work as the others. Therefore, the load across processing units needs to be balanced. This issue is addressed using cyclic mapping to map an event to a core.

In short, the SPDKE implements an event handling application using multiple threads. Events are mapped to a core using a cyclic mapping upon allocation, and the SPDKE will, later, schedule them on the specific processing unit. Any child events generated by an event run at the same thread as the parent event. Each thread has its channel to communicate with the storage devices allowing it to run independently.

4.4.3 *Leader*

The most relevant aspects of the leader's implementation are the proposal discovery mechanism and the management of the Disk Paxos instances computing. Algorithm 10 covers a high-level view of the proposal discovery mechanism which it described in detail bellow.

With the decision of a proposal system based on disks, a leader finds new proposals by performing readings in a restricted region of each disk. The leader's thread discovers new proposals by actively reading the disk sections where proposals appear that proceed the last proposal found (procedure *Search*, line 1). The leader accomplishes this by submitting events to the SPDKEA (procedure *Discovery*, line 30). Those events will result in readings on a majority of disks (line 20) and merging the results into a key-value map where the keys are slot numbers, and the values are the respective proposal. To find multiple proposals (N) with one request, the SPDKEA reads $N \times NProcesses$ sectors, then it iterates over them and builds the map using the first proposal found to each slot number (line 6). Upon completion of a request, the SPDKEA notifies the leader using the Future mechanism (line 16). There is only one discovery request active at the same time and, to actively discover new proposals, a leader starts a new search right after the results of the previous one arrive (lines 2 to 8, algorithm 11). Any proposal for a unseen slot number is added into the lane queues.

Algorithm 10 Proposal Discovery Mechanism

```
1:  procedure read_callback(opt)                          ▷ Callback function of IO request
2:      opt.status ← true
3:      if opt.dec.status == false then
4:          prop ← opt.prop
5:          prop.diskSeen ← prop.diskSeen ∪ {opt.disk}
6:          for i = 0; i < prop.size; i = i + 1 do
7:              for j = 0; j < Processes; j = j + 1 do
8:                  if slot + i ∉ prop.data & valid(opt.buf[i × Processes + j]) then
9:                      prop.data[slot + i] ← deserialize(opt.buf[i × Processes + j])
10:                     break
11:                 end if
12:             end for
13:         end for
14:         if |prop.diskSeen| > |disks|/2 then
15:             prop.status ← true
16:             prop.promise(prop.data)
17:         end if
18:     end if
19: end procedure
20: procedure handleProposalDiscovery(prop)               ▷ Internal Proposal Discovery Event
21:     lba ← P × LANES + OFFSET + P × prop.slot
```

```
22:     for d ∈ disks do
23:         qpair ← qpairs[core][d]
24:         opt ← { prop ← prop, disk ← d, status ← false }
25:         spdk_nvme_ns_cmd_read(qpair, opt.buf, lba, P × dec.size, read_callback, opt)
26:         e ← event(event_verify, opt, core)
27:         submit(e)
28:     end for
29: end procedure
30: procedure discovery(slot,size)                    ▷ Interface for Leaders's Thread
31:     core ← nextCoreLeader()
32:     prop ← LeaderProposal(slot, size)
33:     fut ← Future()
34:     prop.promise ← fut.promise()
35:     e ← event(handleProposalDiscovery, prop, core)    ▷ Creates an event for SPDKEA
36:     submit(e)                                         ▷ Submits a event
37:     return fut
38: end procedure
```

The other leader's function is the management of the consensus instances. Algorithm 11 describes the main concepts required for that management as well as details how all leader's functions are put together.

The leader delegates the consensus computation to the SPDKEA and only controls which instances will run. The leader maintains both the current state of each consensus lane and a list of queues, where each entry represents the list of proposals targeting a specific lane. To find out when to start new instances, the leader periodically iterates over the state of the consensus running at the moment (line 16). Since it is the SPDKEA component that computes the consensus protocol, the leader and the SPDKEA must implement a signaling mechanism. This property could be achieved using a global structure with a lock mechanism, but it would be too expensive for the application. Instead, the solution we came up with relies on the use of atomic variables.

In every consensus instance, there is an object maintaining the consensus state. The leader's thread and the thread that handles the algorithm share the object between them. Despite the object containing the entire information about the consensus, the leader only needs to access the data about its current state. Therefore, the signaling is done via an atomic read in the state variable inside the object (line 18). Consequently, any change on that variable is also done atomically to avoid data inconsistency. Whenever the leader discovers an end-state consensus instance, it spawns a new instance for a new slot (line 20).

To summarize, the leader's behaviour consists on checking the current state of the discovery for new proposals and iterating over the launched consensus to launch new ones. Whenever a discovery ends, the leader submits a new one to find out proposals as soon as possible.

Algorithm 11 Leader's Behaviour

```
1:  procedure SEARCH(fut,queues)                          ▷ Procedure to discover new proposals
2:      if fut == null then
3:          fut ← discovery(last_proposal, T)             ▷ Submits an event to read the proposals sectors
4:      else if fut.state == ready then
5:          map < Slot, Proposal >← fut.data
6:          update_queues(queues, map)                    ▷ Updates the structure saving the information about the
    known proposals
7:          fut ← discovery(last_proposal, T)             ▷ Submits an event to discovery new proposals
    again
8:      end if
9:  end procedure
10: procedure LEADER(Lanes)
11:     future ← null
12:     queues ← [∅ for i ← 0 to Lanes − 1]
13:     slots ← [null for i ← 0 to Lanes − 1]
14:     while true do
15:         Search(future, queues)                        ▷ Discovers new proposals
16:         for i = 0 to Lanes − 1 do
17:             s ← slots[i]
18:             if s.state == END then                    ▷ Atomic Read on the consensus instance state
19:                 p ← queues[i].pop()
20:                 slots[i] ← launch(p)                  ▷ Submits an event to start consensus for proposal i
21:             end if
22:         end for
23:     end while
24: end procedure
```

4.4.4 *Consensus Implementation*

The consensus algorithm can be computed in multiple ways. The solution might be having a process or a thread performing it. As a new slot number appeared, the application would launch one of these, and its purpose would only be the protocol's execution. For instance, a threaded model would be a better decision since it would enable the introduction of a thread pool, lowering the cost of creation and destruction of threads for short-lived tasks. The need for an event framework presented a viable alternative to this problem. Therefore, we implemented the consensus using an event approach.

Since an event is simply a function that runs in the near future, the algorithm was implemented using chained events. By doing so, the protocol runs entirely inside the SPDKEA component, and it is easier to handle the communication with the devices while the algorithm is executing. Additionally, the callback functions of the disk operations are used to chain the distinct phases of the algorithm, making its implementation cleaner.

Algorithms 12 and 13 describe how the implementation chains events to reach an agreement. The event-based implementation of the protocol starts with the submission of an event to create a new Disk Paxos instance for a given slot (line 32). Upon handling it, the thread responsible writes the current Disk Paxos Block kept in memory in each system's disks (WriteAndRead Procedure, line 23). Due to the asynchronicity of the driver, as soon as the thread finds out that its write operation was completed for a given device, it immediately submits the read request for the same disk. This is done via the callback function (line 13) passed upon submission of the first write request. When the read operation completes, the read callback function iterates over the data read and adds the blocks to the object maintaining the consensus instance state. Additionally, it verifies the current ballot number and checks for issues regarding protocol abortion or cancelation. This writing and reading procedure has to be completed for more than half of the devices to move forward in the protocol. For this reason, the read callback (line 1) function also adds the current disk identifier to the set of confirmed devices and tests if the condition above is valid. Eventually, there will be a read callback function where the verification will be true, and the analysis phase will start. The analysis phase (line 8, procedure 1) either chooses a value for the next stage or commits the protocol's output for phases one and two, respectively. After selecting a value on phase 1, the protocol moves to phase 2 and repeats the write and read steps by resubmitting the *WriteAndRead* (23) procedure.

Despite submitting requests in parallel, any future events related to the same consensus instance are allocated to the same thread as the initial was. Thereby, there are no issues such as inappropriate access and data inconsistency. The only problem that might appear is the completion of out-of-time IO requests. In other words, the algorithm only requires a majority in the write and read phase, but the thread submits requests for each disk. These requests may be completed at a time where they are no longer needed and should be discarded. To address this issue, the object containing the consensus state keeps an incremental id. Every time a request finishes, the callback function simply compares the current value with the one passed upon submission. If the values don't match, the callback function discards the operation. The value only increments if a new write and read phase starts or ends. Lines 2, 7 and 14 present the usage of this solution.

Algorithm 12 Chained read and write using events for the consensus protocol

```
1:  procedure READ_CALLBACK(opt)                ▷ Callback function after reading from a disk
2:      if opt.tick == opt.dp.tick then
3:          dp ← opt.dp
4:          iterate(dp, opt.data)               ▷ Iterates over data read and verifies each block read
5:          dp.diskSeen ← dp.diskSeen ∪ {opt.disk}
6:          if |dp.diskSeen| > |disks|/2 then
7:              dp.tick ← dp.tick + 1
8:              analysisPhase(dp)
9:          end if
10:     end if
11:     opt.status ← true
12: end procedure
13: procedure WRITE_CALLBACK(opt)               ▷ Callback function after writing on a disk
14:     if opt.tick == opt.dp.tick then
15:         qpair ← qpairs[core][opt.disk]
16:         opt2 ← opt
17:         spdk_nvme_ns_cmd_read(qpair, read_callback, opt2)   ▷ Submits a reading request
18:         e ← event(event_verify, opt2, core)
19:         submit(e)
20:     end if
21:     opt.status ← true
22: end procedure
23: procedure WRITEANDREAD(dp)                  ▷ Event that starts the chain described above
24:     for d ∈ disks do
25:         qpair ← qpairs[core][d]
26:         opt ← {dp ← dp, tick ← dp.tick, disk ← d}
27:         spdk_nvme_ns_cmd_write(qpair, write_callback, opt)   ▷ Submits a writing request
28:         e ← event(event_verify, opt, core)
29:         submit(e)
30:     end for
31: end procedure
32: procedure LAUNCH(Slot,Cmd,Pid)              ▷ Function called by leader to spawn a new instance
33:     dp ← DiskPaxos(slot, cmd)
34:     e ← event(WriteAndRead, dp, core)                                   ▷ Event allocation
35:     submit(e)                                                ▷ Submits an event to SPDKEA
36:     return dp
37: end procedure
```

Lastly, similar to Leader's and Replica's components, the consensus algorithm also had implications due to communication between processes using the shared disks. In the Disk Paxos's original version, the algorithm would commit the decision over the network. The authors suggested that it could be a simple broadcast to every process in the system. With the communication decision, the algorithm writes the protocol's output on every disk in the system instead of broadcasting (line 11).

Algorithm 13 Analysis Phase and Commit Step

```
1:  procedure ANALYSISPHASE(dp)
2:      if dp.phase == 1 then
3:          dp.chooseValue()
4:          dp.phase ← 2
5:          e ← event(WriteAndRead, dp, core)
6:          submit(e)
7:      else
8:          Commit(dp.value, dp.slot)
9:      end if
10: end procedure
11: procedure COMMIT(dp,slot)
12:     for d ∈ disks do
13:         qpair ← qpairs[core][d]
14:         opt ← {disk ← d}
15:         spdk_nvme_ns_cmd_write(qpair, commit_callback, opt)   ▷ Submits a writing request
16:         e ← event(event_verify, opt, core)
17:         submit(e)
18:     end for
19: end procedure
```

From all the shown procedures, only the launch runs on the leader's thread while the others run on the SPDKEA component. Additionally, the information omits not only the buffers where the read and written data goes through but also the indexes regarding the logical sectors where these operations take place.

5

EVALUATION

The following chapter evaluates our implementation. Firstly, it introduces a tuning phase to find a base configuration for a single process since various parameters influence the performance. Secondly, it presents a series of experiments that address multiple aspects about the system's performance. These experiments covered different setups both for the system and the storage network and aimed to discover and study possible issues and improvements. Lastly, we present an evaluation of the solution's scalability and compare the system with an existing open-source implementation of Paxos, LibPaxos [18].

5.1 CONFIGURATION OF PARAMETERS

The parameters that influence the prototype's performance are broken into two groups. The first one refers to parameters related to the network of storage devices and the number of processes in the system. The second group configures a single process.

Table 4 breaks down the parameters for the network and the system. The number of processes and disks establishes the maximum number of processes and disks active, respectively. The lanes parameter sets a maximum for the number of consensus instances computing simultaneously. It is required to ensure that all processes know the space occupied by the disk structures and correctly map their IO requests to the corresponding disk sector. Additionally, it also guarantees that the number of lanes allowed does not vary among leaders.

Name	Description
Nº Processes	Number of processes in the system
Nº Disks	Number of Disks in the system
Lanes	Maximum number of consensus instances simultaneously

Table 4: Parameters that influence the network of processes and disks

Table 5 presents the parameters required to configure a single process. The read decisions and proposals parameters are used by replicas and leaders, respectively. The read decisions establish how many sectors a replica reads from each disk every time it uses the decision discovery mechanism. Similarly, the read proposals parameter sets the number of proposals that a leader attempts to discovery when it uses the proposal discovery mechanism. The remaining parameters are the proposal frequency which establishes how many proposals each

replica issues in each second, and the number of threads that configures the number of threads used by the SPDKEA component inside each process.

Name	Description
Read Decisions	Number of Decisions read each time a replica goes to a disk
Read Proposals	Number of Proposals read each time a leader goes to a disk
Freq. Proposal	Number of Proposals per second
Nº Threads	Number of threads used by the SPDKEA component

Table 5: Parameters to configure a single process

The tuning phase has the objective of finding a base configuration for the process parameters. From the parameters presented in Table 5, this phase only addresses the proposal frequency and the number of slots read by a leader in each request. These two are the ones that have the highest impact on the algorithm since they influence the leader's performance.

To run adequate tests, the Google Cloud Platform was used. The tests used a proposal frequency ranging from 10000 to 40000 proposals per second and captured the system's throughput with distinct configurations. The environment was composed of three N2 instances, each with 8 CPUs and 32 GB of memory. Each machine had a disk application running on 4 processing units.

Table 6 presents a summary of the combinations tested. Each combination consisted in a pair with a proposal frequency value and a read proposals value. The remaining parameters did not change across the tests and were configured using the values on Table 7.

Parameters	
Proposal Frequency (Proposals / Sec)	[10000,13333,20000, 40000]
Read Proposals	[4,8,16]

Table 6: Test Parameters

Parameter	Value
Lanes	32
Threads	2
NºProcesses	3
Read Decisions	8
Nº Disks	3

Table 7: Fixed Parameters

Figure 11 shows the average results of 3 runs using the correspondent configuration. During the tests, the system was configured for a network of three processes. The results were collected after the system converged to having a single leader. Most of the time, the system will only have a leader. Consequently, the prototype should be optimized for such circumstances which are more common to appear.

Heatmap for the process throughput		Read Proposals		
		4	**8**	**16**
Proposal Frequency (Proposals / sec)	**10000**	5142	5626	3601
	13333	4614	6279	4860
	20000	5037	6745	4634
	40000	4893	6360	5230
		Decisions/sec		

Figure 11: Heat map for the throughput of the application while tuning

The chosen configuration for the following tests was a proposal frequency of 20000 proposals per second and a value of 8 for the amount read by a leader. At first sight, if the system is achieving a maximum of 6700 decisions per sec, it would require a lower proposal frequency to match the observed output. Nonetheless, a write operation takes a certain amount of time to finish, and it may be placed in a queue delaying it further. Besides, it is essential to avoid starving situations in a leader; otherwise, the performance may be affected. A high-frequency ratio helps evade these issues; however, the value should not be too high to avoid affecting the system with overload. For these reasons, the chosen configuration outperforms the others.

5.2 IMPACT OF NUMBER OF LANES AND THREADS

Differently from the tuning phase, the performance evaluation test fixed the proposal frequency and the number of slots read by leaders. At the start of execution, each process running in a different machine started as the leader and, eventually, the system converged into having a single leader active. The hardware was again N2 instances, but this time configured to have 16 CPUs and 64 GB of memory. This change was due to increasing the number of processing units available for the NVMe-oF application in each storage device.

For these tests, the focus was on evaluating the throughput using distinct values for the maximum number of consensus instances (Lanes) as well as a varied number of threads for the SPDKEA component. Each combination tested used the base configuration of the previous section, Table 9, and a unique pair obtained by combining the values on Table 8.

Parameters	
Lanes	[16,32,64,128]
Threads	[1,2,4,6]

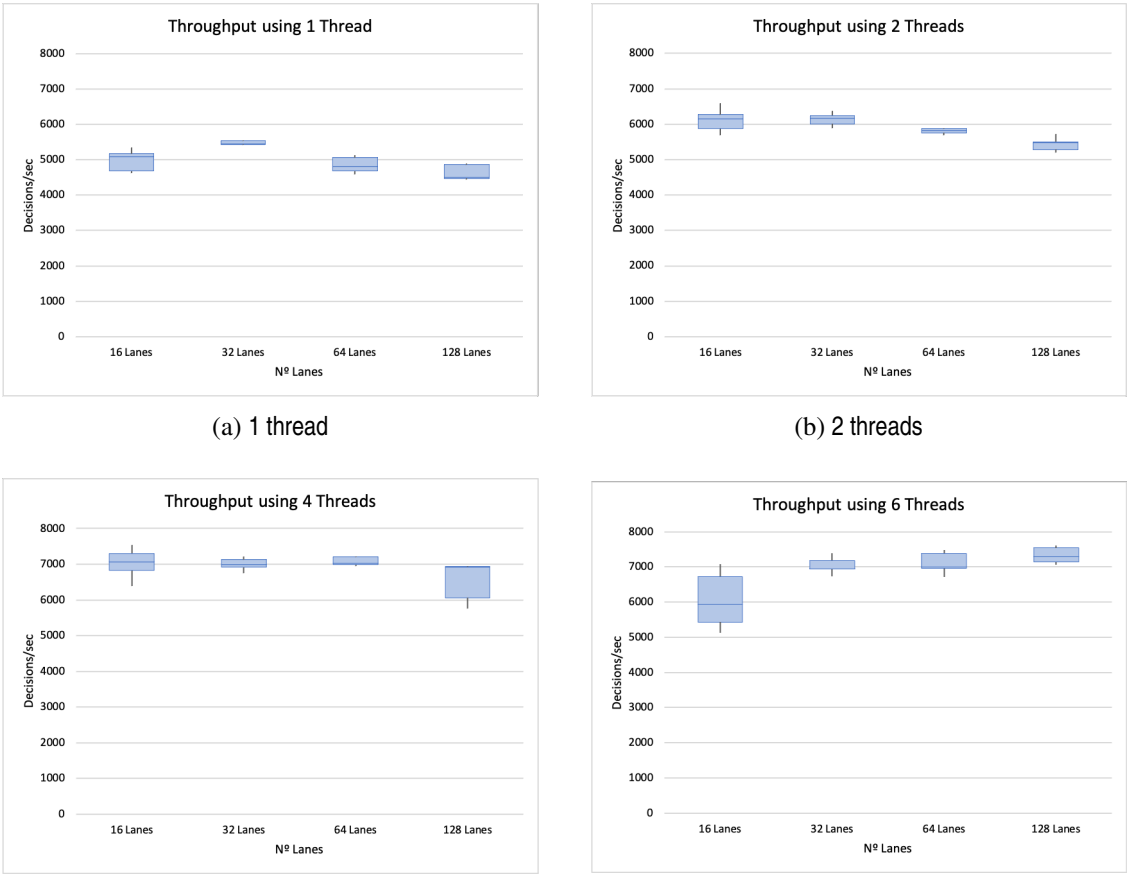
Table 8: Testing Parameters

Parameter	Value
Nº Disks	3
NºProcesses	3
Read Decisions	8
Proposal Frequency	20000 / sec
Read Proposals	8

Table 9: Fixed Parameters

With the NVMe-oF application using 8 CPUs, each machine only had eight cores left to run the application. Ideally, the replica and leader threads inside each process should be on a processing unit during the entire execution. As a result, the maximum number of threads available for the SPDKEA component tested was six.

Figure 12 exhibits the results of running three tests using each configuration. The chart type chosen was a quartile chart since the test measured the throughput in each process, and the quartile chart makes it easier to understand and analyze the system's throughput stability. A wide bar means that the variability of the values is higher, while a short bar means steadier results. Each graph compares the performance of different values for the variable lanes with a fixed number of threads.



(a) 1 thread

(b) 2 threads

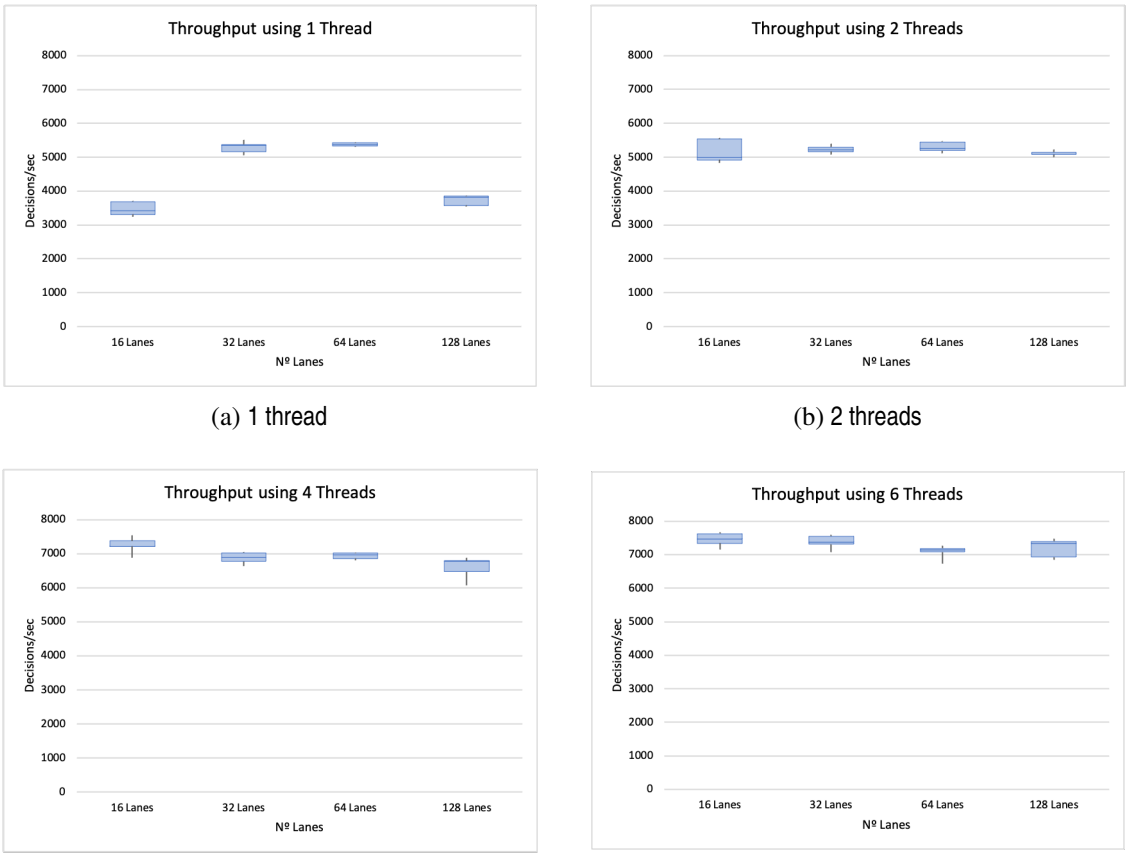
(c) 4 threads

(d) 6 threads

Figure 12: Throughput using different values of lanes and threads

The results of Figure 12 show that as the number of threads and lanes grows, the throughput also increases. The experiments that perform the worst are using a single thread. The disk-based communication system generates a high number of events. These events, when combined with the ones generated by the consensus protocol, overload the *SPDKEA* component. The configurations using smaller thread pools are the most harmed by this problem. This issue is the main reason for the discrepancy between the throughput using different thread numbers.

As said previously, wide quartile bars mean that the results are scattered and are not very stable. Most bar charts have a slight variation. The ones with a higher instability are due to how the system addresses the leader election. When the system takes too much time to converge, it decreases its performance. Since the benchmark can't start every process exactly at the same time, sometimes the processes take too much time to realize who is going to lead. This problem appears most frequently when the expected leader is the last to launch, and the first starting process accomplishes various agreements in a short period. To draw more conclusions, an equivalent test but with a single and a fixed leader was conducted. The results are shown in Figure 13 and revealed a reduction in the instability of each configuration. A comparison between the results from 6 threads and 16 lanes or 4 threads and 128 lanes of both tests shows that the first one presents wider quartile bars meaning the results are more scattered. Nevertheless both tests presented a top throughput of over 7500 decisions per sec while using six threads configurations.



(a) 1 thread

(b) 2 threads

(c) 4 threads

(d) 6 threads

Figure 13: Throughput using different values of lanes and threads and a fixed leader

In conclusion, a configuration with a good throughput requires a thread pool with at least four threads. In addition, the leader election should be improved since the expected leader may be in situations where there is another leader working and it is ahead of the expected one. Since the leaders only quit if they find a collision, it may take some time until, the system converges to a single leader.

5.3 IMPACT OF THE DISK-BASED COMMUNICATION SYSTEM AND THE NUMBER OF LANES

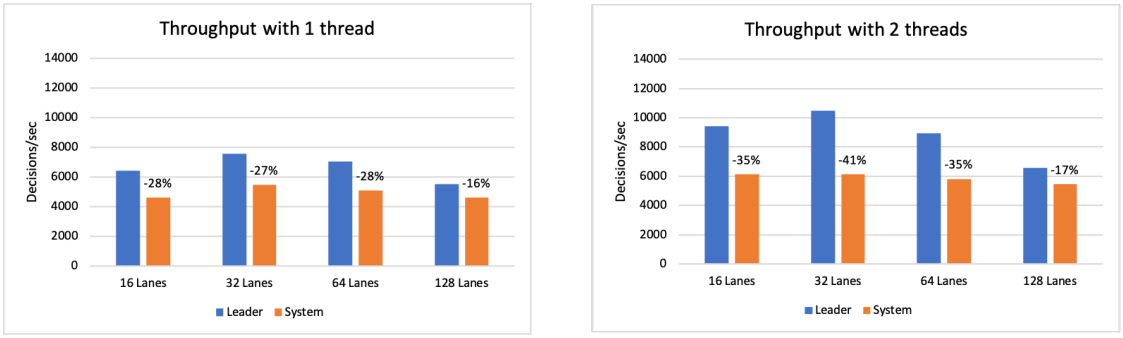
One of the core decisions while implementing the system was the communication between processes. A disk-based communication system required a special mechanism for each process to propose and detect new decisions. As a result, each NMVe Device in the storage network had an increased load. Since this implementation decision affects the most critical system's part, the consensus algorithm, and it needs to be as efficient as possible, it is crucial to understand the impact of the mechanisms for proposing and propagating decisions.

The impact can be measured by splitting the system into two parts: the mechanisms to propose and discover new decisions, and the execution of the Disk Paxos protocol. By evaluating the Disk Paxos execution separately, it is possible to collect results without the interference of the communication system. Inherently, comparing those results with the ones from Section 5.2 (collected using both parts) gives an idea about the communication's influence on the entire system.

For such, an environment identical to the previous test was set up; however, this time, the system had no replicas active, and there was only one process simulating a leader's behavior. The existing leader already had all the proposals in memory before starting the execution, and its only role was to manage the consensus execution. Despite only having a single process executing, from a computational point of view, the load of the consensus protocol upon each NVMe Device was the same as before since each consensus instance reads and writes the same amount of data. Using an environment as described, the test measured the system's throughput under ideal conditions (similar to when the system converges to have a single leader) without the additional effort required for the proposal or decision mechanism.

Likewise, in Section 5.2, the hardware chosen was three N2 instances, each with 16 CPUs and 64 GB of memory. Each N2 instance had an NVMe-oF application using 8 of the available processing units. The system configuration was the same as before (Tables 8 and 9).

Figure 14 shows the values using different configurations and compares, side by side, the results of the previous test (System) with the experiment above described (Leader). The throughput of the Leader's trial represents the average of three runs, while the system's value corresponds to the average of all processes among also three runs. Additionally, the percentage value above the system's bars expresses how much smaller the result was when compared with the Leader's one. The second chart (Figure 15) shows the average time an agreement took to complete and helps understanding if the test is suffering from saturation for having a high value for the number of lanes.



(a) 1 thread (b) 2 threads

(c) 4 threads (d) 6 threads

Figure 14: Comparison between throughput measured in the replicas vs leaders

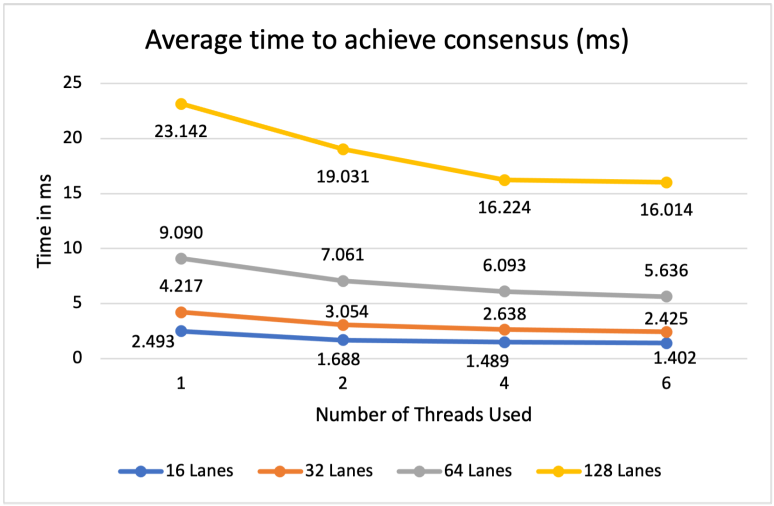


Figure 15: Average time to reach an agreement using three disks

After a look into the data of Figure 15, the charts helped concluding about the impact of the number of lanes. As the number of lanes increased, the time to achieve consensus also grew for every configuration tested; however, it did not increase proportionally to the lanes values. While the rise from 16 to 32 lanes increased the consensus time by 1.75x, an improvement considering that the number of consensuses also doubled, the change from 64

to 128 lanes resulted in a diminishment since it raised the agreement time by more than two (2.7x). Figure 16 shows another view on the issue mentioned. It took the data from Figure 15 and compared the increase in the consensus time over the 16 lanes version as the number of lanes increased.

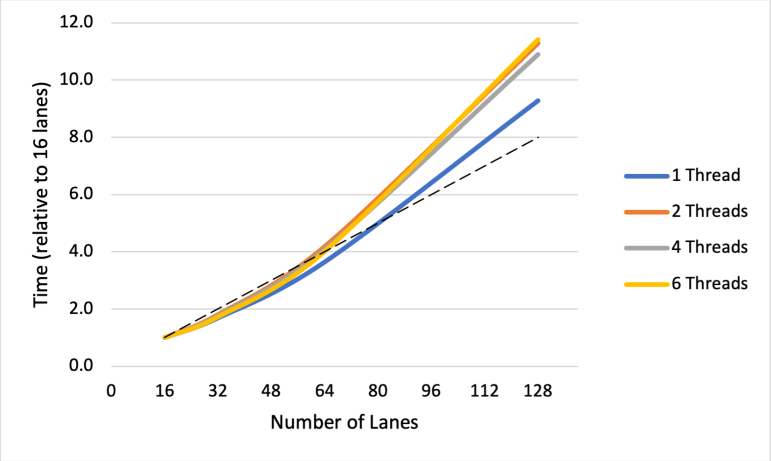


Figure 16: Consensus time increase over the 16 lanes version as the number of lanes upraises

Looking at Figure 16, the dashed line represents the expected rise. Any option resulting in a value above the line means its benefits to the algorithm are not worth since the delay on the time to achieve consensus is higher than the benefit of having more consensus instances simultaneously. In other words, the maximum number of consensus instances is too high, overloading the storage and delaying way too much the agreement time. As the Figure 16 shows, the configurations using both 64 and 128 lanes already overload the system despite presenting the best results in Figure 12.

When analyzing the throughput captured, it can be seen that the leader's test has a significantly higher performance. This suggests that the system is not taking the full potential of Disk Paxos with SPDK. The reason behind the chart plotting a decrease in the leader's throughput between lanes 32 and 64 or 128 has been explained before. The 128-lanes results show the lowest difference in percentage since the configuration already decreases the leader's performance. On the other setups, the main reason for the high discrepancy between the leader's and the system's performance is the communication system. The disk-based communication beyond increasing the charge in each NVMe device takes too much time to propagate proposals and disseminate decisions. As a result, the decisions get delayed and leaders can't exclusively use the disk's performance for the consensus computation. Consequently, the full potential of the system diminishes.

These conclusions were possible because the only difference between Leader and System tests above is the existence of the communication system. On average, the communication system is responsible for a 40% decrease in the overall performance in the best configurations. This 40% includes the propagation of proposals and decisions, and the increased charge on NVMe devices which increases their latency. In addition, it also contains possible starvation situations on leaders since if proposals take too long to be discovered, starvation is likely to occur often.

Overall, the test shows a clear look into a major issue on the system. The negative impact of disk-based communication affects the system's performance. Our findings on the system's performance at least hint that there is room for improvement. A different communication system may shorten the gap between the leader's and the system's throughput.

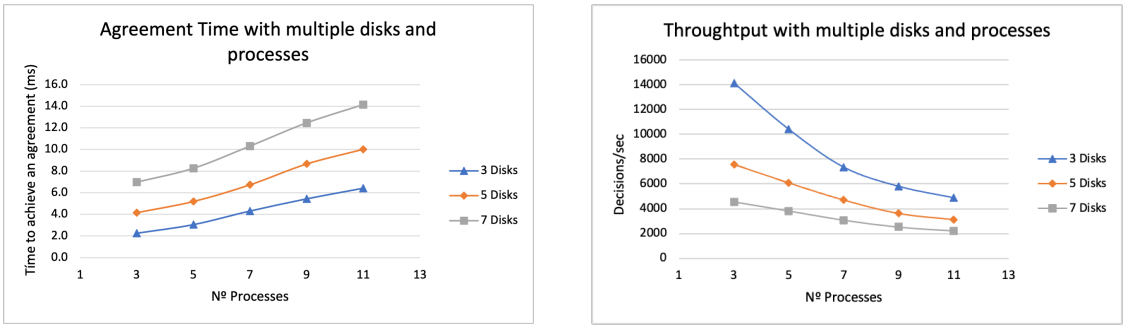
5.4 SCALABILITY OF CONSENSUS ALGORITHM

This section evaluated the algorithm's scalability as the number of processes and disks increased. Since it had the highest impact on the algorithm's computation, the same approach as in Section 5.3 was followed. The test ran using only a single leader already with the proposals in memory and measured the leader's throughput and the agreement required time. The configuration for each process was the one that held the best results previously, 32 lanes and 6 threads.

The environment required an N2 instance with 16 CPUs and 64 GB of memory for each device. Despite only having a process running, each disk needed to be in a distinct machine to simulate a realistic environment. Similar to the previous setup, each device ran the NVMe-oF application on 8 processing units.

For the purpose of studying the algorithm's scalability, the test recorded the output with a 3, 5, and 7 disks and a configured system with the number of processes varying between 3 to 11.

The results of Figures 17a and 17b demonstrate two things. First, the average time to reach an agreement increases linearly with the number of processes. Second, the leader's performance decreases slower and slower as the number of processes rises.



(a) Average Agreement time

(b) Leader's Throughput

Figure 17: Results of running the leader test using a 32 lanes and a 6 threads configuration

Taking into account how Disk Paxos operates, we conclude that the linear increase is due to the amount of data transferred during the consensus execution. Each consensus instance writes in a single disk sector and reads *N* sectors in each phase completed. While a single leader is executing, the algorithm only performs two phases per instance, resulting in a total transfer of $(2 + 2 \times N) \times Sector\_Size$ bytes. Data transfered increases with the number of processes. Consequently, the disk operations become heavier and take more time to complete delaying the average agreement time. These findings hint that prioritizing smaller-sized sectors may help mitigate this issue. The storage had each device configured to 4096-byte sized sectors, but these could be lowered to 512

according to the NVMe specification. Unfortunately, the provided hardware didn't support this feature making any further testing impossible.

Ultimately, the number of disks also impacts the algorithm's performance negatively. With the requirement for executing the write and read phase on a majority of devices, if the number of devices grows, the time to achieve the majority also increases. As expected, the experiment confirmed this increase, and it is an inevitable consequence of using additional disks.

5.5 EVALUATION OF DISK PAXOS AGAINST OTHER APPROACHES

This section compares the system developed (Disk Paxos) with an implementation of Paxos Protocol. The library used for this purpose was LibPaxos. LibPaxos [18] provides a set of open-source implementations of distinct Paxos algorithms that have been extensively tested. The algorithm chosen was the classic Paxos approach.

Comparing both approaches required an evaluation performed under the same settings. For such, the benchmarking used an environment composed of three N2 Google Cloud instances configured with 16 processing units and 64 GB of memory each. Additionally, each machine had an NVMe device attached to it.

In order to be comparable, the results were collected while the LibPaxos and Disk Paxos were configured to use three acceptors and three disks, respectively. Table 10 describes the remaining configuration for Disk Paxos, which corresponds to the setup that held the best results in Section 5.2. Under these circumstances, the tests evaluated the throughput of both approaches as the input size varied.

Parameter	Value
Nº Disks	3
NºProcesses	3
Read Decisions	8
Proposal Frequency	20000 / sec
Slots Read by Leaders	8
Lanes	128
Threads	6

Table 10: Remaining configuration used for Disk Paxos

Figure 18 depicts the observed results. The LibPaxos outperforms the work developed. As the input size increases, the difference between both approaches decreases, having its lowest point at 4096 bytes. The Disk Paxos approach always writes and reads sectors of 4096 bytes despite executing the consensus using substantially smaller inputs. This makes it harder to achieve high throughput when the input requests are small. Nevertheless, the results show that if the work developed improves its communication system and achieves values closer to the ones observed in the leader's maximum throughput of Section 5.3 the Disk Paxos approach might be able to surpass the LibPaxos performance on large inputs. These results also hint that a Disk Paxos approach may be more suitable for environments where the applications always have large requests and is not indicated for applications where the size of its requests has a high variation.

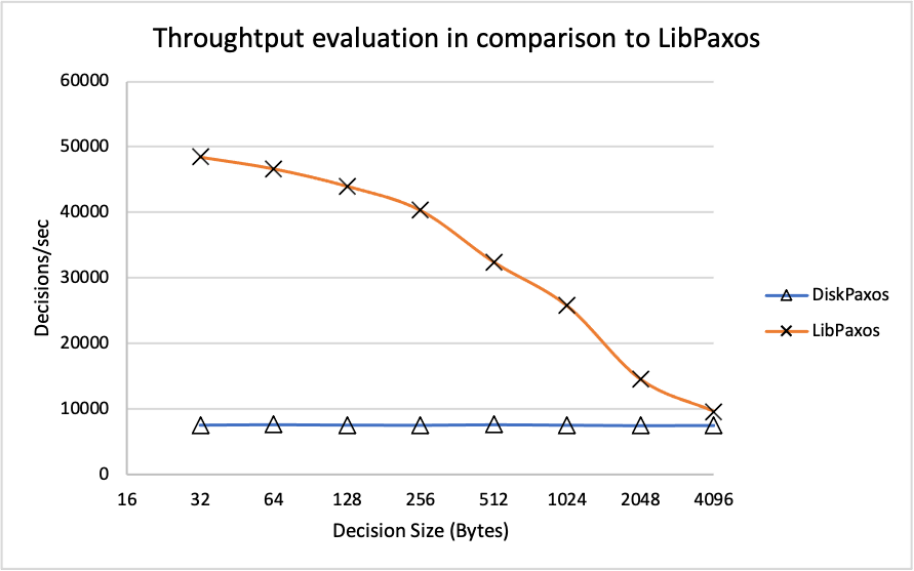


Figure 18: Evaluation using 3 accepts and 3 disks for LibPaxos and DiskPaxos, repectively

6

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSION

In this thesis, we aimed to tackle the consensus problem. Today's available software and hardware opens up new opportunities to solve the consensus problem efficiently. Seizing the significant improvement upon the storage devices and the increasing importance of network storage, this work addressed the consensus problem by developing an agreement system to evaluate the potential of Disk Paxos with the latest generation solid-state devices.

With this work, we designed and implemented a multi-threaded consensus system that restricts the number of instances running simultaneously and takes advantage of the microseconds latency of NVMe Devices. The system integrated Disk Paxos, the consensus protocol, with a disk-based communication system to deliver proposals for future agreements and disseminate decisions across the network. The implementation resorted to the Storage Performance Development Kit for the connection drivers to the NVMe Devices. This toolkit enabled a fully asynchronous approach for the consensus protocol. It also provided an extension of the NVMe-oF protocol using a TCP connection to access the storage remotely.

Our findings indicate that the approach developed does not have much potential while the values to agree on are small-sized. When the size of the operations increases and gets closer to the disk sector size used, the Disk Paxos protocol starts being competitive. This finding indicates that our approach is suitable for environments where the type of operations performed require at least 2KB to be stored. Additionally, our study on the algorithm's scalability revealed that the number of processes in the system increases the agreement's time despite only having a unique leader executing when the system is stable.

In summary, the latest solid-state devices (NVMe devices) may help to improve the available solutions that address the consensus problem. However, their benefits may only play a vital role under specific circumstances, limiting their usage.

6.2 PROSPECT FOR FUTURE WORK

There is work to be done that could improve or provide additional hints about the algorithm. Bellow are presented the most relevant aspects to cover in future work that either address existing issues or provide new guidance for future optimizations:

- **Improve the communication system using SGL Operations:** SGL operations allow the handling of requests that act upon discontiguous blocks. Hence, enhancing the communication system by enabling a batch system where a replica may, with a single request, propose multiple times simultaneously. SGL achieves this by taking a large segment of blocks and labeling parts of that segment. These labels specify which segment's sections are writable, readable, skippable, and others. Using only these three, it is already possible to read and write in sectors that are not contiguous. Despite its flexibility and usage, the SGL instructions are only available in high-end devices, making them unavailable in most NVMe devices.

- **Address the high discrepancy between the system and leader results on Section 5.3:** The high discrepancy results are the result of a slow communication system. Instead of attempting to improve it, the path to pursuit may be on replacing the system with another method of communication. A message sending and delivering in a fast network only incurs an overhead of a hundred-microsecond latency. A message-based system may significantly improve the system throughput and presents an alternative and viable solution to the existing disk-based communication.

- **Evaluate the impact of changing the NVMe-oF over TCP to RDMA:** NVMe over Fabrics, the protocol that extends the NVMe protocol to the network, can be set up using multiple network protocols. TCP presented the most versatile choice without adding any hardware requirement however, the usage of TCP may come with a cost since it is not the fastest protocol. As seen in [15, 16, 23, 22], RDMA improves the protocol's latency, therefore, presenting a possible solution to speed up the submission and completion of NVMe requests, further increasing the storage's performance.

- **Implement an optimization to enable a one-round algorithm:** Many approaches to consensus make assumptions and decisions that enable the algorithm to be performed in one round. Since the implementation always performs the algorithm in two rounds, enabling a one-round version under specific circumstances will boost the system performance. For instance, the Mu's approach [15] performs the agreement in one round when the leader executes the prepare phase and does not find any proposal resulting in choosing its value. Apus [16] also uses a viewstamp-based approach that relies upon agreeing on a view and performing the one-round algorithm until the view changes. Such a mechanism can also be implemented in the system.

- **Study possible improvements to mitigate the large sector sizes of storage devices:** The main obstacle to the algorithm's scalability is the data that each new process adds to the total data transferred during the executing of Disk Paxos. Despite not being involved in the consensus, new processes increase

the computation time of the consensus. Finding a solution for this issue is fundamental to improving scalability.

BIBLIOGRAPHY

[1] Eli Gafni and Leslie Lamport. *Disk Paxos*. Maurice Herlihy, 2002

[2] Robbert Van Renesse and Deniz Altinbuken. *Paxos Made Moderately Complex*. ACM Comput. Surv, 2015

[3] Ziye Yang, James Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma and Luse Paul. *SPDK: A Development Kit to Build High Performance Storage* Applications. IEEE, 2017

[4] Intel. *Storage Performance Development Kit, Documentation, NVMe Driver* (https://spdk.io/doc/nvme.html). 2021

[5] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh and Vijay Balakrishnan. *Performance Analysis of NVMe SSDs and their Implication on Real World Databases*. SYSTOR '15: Proceedings of the 8th ACM International Systems and Storage Conference. 2015

[6] Luiz Barroso, Jimmy Clidaras and Urs Hölzle. *The Datacenter as a Computer, An introduction to the Design of Warehouse-Scale Machines, Second Edition*. Morgan and Claypool Publishers. 2013

[7] Vasilis Gavrielatos, Antonios Katsarakis and Vijay Nagarajan. *Odyssey: The Impact of Modern Hardware on Strongly-Consistent Replication Protocols*. EuroSys '21. ACM Comput. 2021

[8] Diego Ongaro and John Ousterhout. *In search of an understandable consensus algorithm*. Proc ATC'14, USENIX Annual Technical Conference. USENIX. 2014

[9] Oracle. *Guide to MySQL High Availability*. A MySQL White Paper. 2018

[10] Leslie Lamport. *The part-time parliament*. ACM Transactions on Computer Systems, 16(2):133–169. 1998.

[11] Leslie Lamport. *Paxos made simple*. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001). 2001.

[12] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. *P4xos: Consensus as a Network Service*. EEE/ACM Transactions on Networking, vol. 28, no. 4, pp. 1726-1738. Aug. 2020

[13] Data Plane Development Kit (https://www.dpdk.org)

[14] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira and Benjamin Reed. *ZooKeeper: wait-free coordination for internet-scale systems*. In USENIX Annual Technical Conference (ATC). June 2010.

[15] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, Igor Zablotchi *Microsecond Consensus for Microsecond Applications*. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 2020

[16] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. *APUS: Fast and Scalable Paxos on RDMA*. In Proceedings of SoCC '17. 2017

[17] P4.org. `http://p4.org`. 2015

[18] DanieleSciascia. *libpaxos*. Bitbucket Source. 2013

[19] S. Pontarelli et al.. *FlowBlaze: Stateful packet processing in hard-ware* Proc. of 16th USENIX Symp, on Networked Systems Design and Implementation, ser, NSDI '19. 2019.

[20] Giacomo Belocchi, Valeria Cardellini, Aniello Cammarano, Giuseppe Bianchi. *Paxos in the NIC: Hardware Acceleration of Distributed Consensus Protocols* 16th International Conference on the Design of Reliable Communication Networks DRCN 2020. 2020

[21] David Mazieres. *Paxos made practical*. Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers. 2017

[22] Marius Poke and Torsten Hoefler. *DARE: High-performance state machine replication on RDMA networks*. In Symposium on High-Performance Parallel and Distributed Computing (HPDC), pages 107–118. ACM. June 2015.

[23] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. *Derecho: Fast state machine replication for cloud services*. ACM Transactions on Computer Systems (TOCS), 36(2). April 2019

[24] Fred B. Schneider.*Implementing fault-tolerant services using the state machine approach: a tutorial*. ACM Comput. Surv. 22, 4, 299–319. Dec. 1990

[25] Leslie Lamport. *Time, clocks, and the ordering of events in a distributed system*. Commun. ACM 21, 7 (July 1978), 558–565. 1978.

Part I

APPENDICES

A

SUPPORT WORK

A.1 AUXILIARY RESULTS OF THE TUNING TESTS, (SECTION 5.1)

The images below present the results that supported the values presented in the tuning phase. The measurements were performed using a network of three processes and three disks on the Google Cloud Platform. The instances used were N2 instances configured with 8 CPUs and 32 GB of memory.

The units for the proposal frequency are microseconds, while the output is measured in operations per second. The Hop-stop values were collected after discarding the initial and ending 10% of proposals. Every value shown in the heat map represents the average of 3 runs using the same configuration.

		Runs							
Freq	Leader Read	1	1-HS	2	2-HS	3	3-HS	AVG	AVG-HS
10000	4	5128	5081	4908	4824	5389	5395	5142	5100
13333	4	4950	4806	4193	3973	4698	4524	4614	4434
20000	4	5343	5133	5061	4843	4708	4462	5037	4813
40000	4	5195	4895	4622	4287	4863	4561	4893	4581
10000	8	5459	5458	5546	5544	5873	5868	5626	5623
13333	8	6192	6139	6454	6452	6191	6068	6279	6220
20000	8	6578	6443	7287	7214	6369	6218	6745	6625
40000	8	6487	6203	6348	6021	6245	5903	6360	6042
10000	16	3654	3346	3362	3075	3786	3452	3601	3291
13333	16	5434	5185	4732	4390	4413	4066	4860	4547
20000	16	4636	4180	4936	4465	4329	3939	4634	4195
40000	16	4934	4486	5201	4704	5554	5081	5230	4757

Figure 19: Values collected to build the heap maps

Heatmap for the process throughput		Read Proposals		
		4	**8**	**16**
Proposal Frequency (Proposals / sec)	**10000**	5100	5623	3291
	13333	4434	6220	4547
	20000	4813	6625	4195
	40000	4581	6042	4757
		Decisions/sec		

Figure 20: Heat map in the hot-spot area